Hashfunktionen und MACS

11

Verschlüsselung sorgt für Vertraulichkeit. Vertraulichkeit ist aber nicht das einzige Schutzziel, das mit Kryptographie erreicht wird. Andere wichtige Schutzziele sind *Integrität* und *Authentizität*. Integrität bedeutet, dass Daten nicht verändert wurden. Authentizität garantiert, dass der Ursprung von Daten ermittelt werden kann. Wir geben zwei Beispiele, die diese beiden Schutzziele illustrieren:

Beispiel 11.1 Internet Browser speichern viele Zertifikate, die sichere Kommunikation zwischen dem Browser und Web-Anwendungen ermöglichen. Dies ist in Kap. 16 beschrieben. Die Zertifikate müssen nicht geheimgehalten werden. Die Sicherheit der Kommunikation hängt aber davon ab, dass die Zertifikate nicht verändert werden. Ihre Integrität muss also gewährleistet sein.

Beispiel 11.2 Bei *Phishing-Angriffen* schickt der Angreifer Nachrichten an Kundinnen und Kunden einer Bank, die aussehen wie Nachrichten der Bank selbst, und fordert die Kundinnen und Kunden darin auf, ihre geheimen Passwörter preiszugeben. Es ist darum für Empfängerinnen und Empfänger einer solchen Nachricht wichtig, festzustellen, ob die Nachricht *authentisch* ist, also ob die Nachricht wirklich von der Bank kommt oder nicht.

In diesem Kapitel behandeln wir *kryptographische Kompressionsfunktionen*, *Hashfunktionen* und *Message-Authentication-Codes*, die Integrität und Authentizität von Daten ermöglichen. Darüber hinaus haben diese kryptographischen Komponenten noch zahlreiche Anwendungen. Im gesamten Kapitel ist Σ ein Alphabet.

J. Buchmann, *Einführung in die Kryptographie*, Springer-Lehrbuch,
DOI 10.1007/978-3-642-39775-2_11

11.1 Hashfunktionen und Kompressionsfunktionen

Wir definieren zuerst Hashfunktionen.

Definition 11.1 Eine *Hashfunktion* ist eine Abbildung

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

Hashfunktionen bilden also beliebig lange Strings auf Strings fester Länge ab. Sie sind nie injektiv.

Beispiel 11.3 Die Abbildung, die jedem Wort $b_1 b_2 \dots b_k$ aus $\mathbb{Z}_2^*$ die Zahl $b_1 \oplus b_2 \oplus b_3 \oplus \dots \oplus b_k$ zuordnet, ist eine Hashfunktion. Sie bildet z. B. 01101 auf 1 ab. Allgemein bildet sie einen String b auf 1 ab, wenn die Anzahl der Einsen in b ungerade ist und auf 0 andernfalls.

Hashfunktionen können mit Hilfe von *Kompressionsfunktionen* erzeugt werden.

Definition 11.2 Eine Kompressionsfunktion ist eine Abbildung

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

Kompressionsfunktionen bilden also Strings einer festen Länge auf Strings kürzerer Länge ab.

Beispiel 11.4 Die Abbildung, die jedem Wort $b_1 b_2 \dots b_m$ aus $\mathbb{Z}_2^m$ die Zahl $b_1 \oplus b_2 \oplus b_3 \oplus \dots \oplus b_n$ zuordnet, ist eine Kompressionsfunktion, solange $1 \leq n < m$ ist.

Hashfunktionen und Kompressionsfunktionen werden für viele Zwecke gebraucht, z. B. um die Suche in Wörterbüchern zu unterstützen. Auch in der Kryptographie spielen sie eine wichtige Rolle. Kryptographische Hash- und Kompressionsfunktionen müssen Eigenschaften haben, die ihre sichere Verwendbarkeit garantieren. Diese Eigenschaften werden jetzt beschrieben.

Dabei ist $h : \Sigma^* \to \Sigma^n$ eine Hashfunktion oder $h : \Sigma^m \to \Sigma^n$ eine Kompressionsfunktion. Den Definitionsbereich von h bezeichnen wir mit D. Es ist also $D = \Sigma^*$, wenn h eine Hashfunktion ist, und es ist $D = \Sigma^m$, wenn h eine Kompressionsfunktion ist.

Um h in der Kryptographie verwenden zu können, verlangt man, dass der Wert $h(x)$ für alle $x \in D$ effizient berechenbar ist. Wir setzen dies im Folgenden voraus.

Die Funktion h heißt *Einwegfunktion*, wenn es praktisch unmöglich ist, zu einem $s \in \Sigma^n$ ein $x \in D$ mit $h(x) = s$ zu finden. In Abschn. 1.5 wurde erläutert, was „praktisch unmöglich“ heißt.

Es ist nicht bekannt, ob es Einwegfunktionen gibt. Es gibt aber Funktionen, die nach heutiger Kenntnis Einwegfunktionen sind. Ihre Funktionswerte sind leicht zu berechnen, aber es ist kein Algorithmus bekannt, der die Funktion schnell genug umkehren kann.

Beispiel 11.5 Ist p eine gemäß Tab. 8.4 zufällig gewählte Primzahl und g eine Primitivwurzel mod p, dann ist nach heutiger Kenntnis die Funktion $f : \{0, 2, \ldots, p-2\} \to \{1, 2, \ldots, p-1\}$, $x \mapsto g^x \bmod p$ eine Einwegfunktion, weil kein effizientes Verfahren zur Berechnung diskreter Logarithmen bekannt ist (siehe Kap. 10).

Eine *Kollision* von h ist ein Paar $(x, x') \in D^2$ von Strings, für die $x \neq x'$ und $h(x) = h(x')$ gilt. Alle Hashfunktionen und Kompressionsfunktionen besitzen Kollisionen, weil sie nicht injektiv sind.

Beispiel 11.6 Eine Kollision der Hashfunktion aus Beispiel 11.3 ist ein Paar verschiedener Strings, die beide eine ungerade Anzahl von Einsen haben, also z. B. $(111, 001)$.

Die Funktion h heißt *schwach kollisionsresistent* (Englisch: *second preimage resistant*), wenn es praktisch unmöglich ist, für ein vorgegebenes $x \in D$ eine Kollision (x, x') zu finden. Nachfolgend findet sich ein Beispiel für die Verwendung einer schwach kollisionsresistenten Hashfunktion.

Beispiel 11.7 Alice möchte ein Verschlüsselungsprogramm auf ihrer Festplatte gegen unerlaubte Änderung schützen. Mit einer Hashfunktion $h : \Sigma^* \to \Sigma^n$ berechnet sie den Hashwert $y = h(x)$ dieses Programms x und speichert den Hashwert y auf ihrer persönlichen Chipkarte. Abends geht Alice nach Hause und nimmt ihre Chipkarte mit. Am Morgen kommt sie wieder in ihr Büro. Sie schaltet ihren Computer ein und will das Verschlüsselungsprogramm benutzen. Erst prüft sie aber, ob das Programm nicht geändert wurde. Sie berechnet den Hashwert $h(x)$ erneut und vergleicht das Resultat y' mit dem Hashwert y, der auf ihrer Chipkarte gespeichert ist. Wenn beide Werte übereinstimmen, wurde das Programm x nicht geändert. Falls h schwach kollisionsresistent ist, kann niemand ein verändertes Programm x' erzeugen mit $y' = h(x') = h(x) = y$.

Beispiel 11.7 zeigt eine typische Verwendung von kollisionsresistenten Hashfunktionen. Sie erlauben die Überprüfung der *Integrität* eines Textes, Programms, etc., also die Übereinstimmung mit dem Original. Mit der Hashfunktion kann die Integrität der Originaldaten auf die Integrität eines viel kleineren Hashwertes zurückgeführt werden. Dieser kleinere Hashwert kann an einem sicheren Ort, z. B. auf einer Chipkarte, gespeichert werden.

Die Funktion h heißt *(stark) kollisionsresistent*, wenn es praktisch unmöglich ist, irgendeine Kollision (x, x') von h zu finden. In manchen Anwendungen muss man Hashfunktionen benutzen, die stark kollisionsresistent sind. Ein wichtiges Beispiel sind elektronische Signaturen, die im nächsten Kapitel beschrieben werden. Jede stark kollisionsresistente Hashfunktion ist auch schwach kollisionsresistent. Außerdem ist jede schwach kollisionsresistente Hashfunktion auch eine Einwegfunktionen. Die Beweisidee ist folgende: Angenommen, es gibt einen Invertierungsalgorithmus, der zu einem Bild y ein Urbild x ausrechnen kann mit $y = h(x)$. Dann können wir einen Algorithmus konstruieren, der

ein zweites Urbild eines Hashwertes $y = h(x), x \in \Sigma^*$ berechnet. Dazu wendet unser Algorithmus den Invertierungsalgorithmus auf y an und erhält das Inverse x'. Ist $x' \neq x$, so ist x' offensichtlich ein zweites Urbild von y. Andernfalls hat unser Algorithmus versagt. Das ist aber sehr unwahrscheinlich, weil Hashwerte sehr viele Urbilder haben.

11.2 Geburtstagsangriff

In diesem Abschnitt beschreiben wir einen Algorithmus zur Berechnung von Kollisionen einer Kompressionsfunktion

$$h : \Sigma^m \to \Sigma^n.$$

Der Algorithmus heißt *Geburtstagsangriff*. Er kann leicht auf Hashfunktionen übertragen werden.

Der Geburtstagsangriff berechnet und speichert so viele Paare $(x, y = h(x)), x \in \Sigma^*$, wie es die zur Verfügung stehende Rechenzeit und der vorhandene Speicher zuläßt. Dabei werden die Urbilder x zufällig und gleichverteilt gewählt. Diese Paare werden nach der zweiten Komponente sortiert. Werden dabei zwei verschiedene Paare (x, y), (x', y) mit derselben zweiten Komponente gefunden, so ist eine Kollision (x, x') von h gefunden.

Das Geburtstagsparadox (siehe Abschn. 1.3.1) erlaubt die Analyse dieses Verfahrens. Die Hashwerte entsprechen den Geburtstagen. In Abschn. 1.3.1 wurde folgendes gezeigt: Wählt man k Argumente $x \in \Sigma^*$ zufällig aus, wobei

$$k \geq (1 + \sqrt{1 + (8 \ln 2)|\Sigma|^n})/2$$

ist, dann ist die Wahrscheinlichkeit dafür, dass zwei dieser Argumente denselben Hashwert haben, größer als $1/2$. Der Einfachheit halber nehmen wir an, dass $\Sigma = \mathbb{Z}_2$ ist. Dann brauchen wir

$$k \geq f(n) = (1 + \sqrt{1 + (8 \ln 2)2^n})/2.$$

Folgende Tabelle zeigt einige Werte von $\log_2(f(n))$.

n	50	100	150	200
$\log_2(f(n))$	25.24	50.24	75.24	100.24

Wenn man also etwas mehr als $2^{n/2}$ viele Hashwerte bildet, findet die Geburtstagsattacke mit Wahrscheinlichkeit $\geq 1/2$ eine Kollision. Um die Geburtstagsattacke zu verhindern, muss man n so groß wählen, dass es unmöglich ist, $2^{n/2}$ Hashwerte zu berechnen und zu speichern. Tab. 11.1 zeigt entsprechende Werte für n. Sie ergeben sich aus [73].

Tab. 11.1 Mindest-Hashlängen

Schutz bis zum Jahr a	Mindest-Hashlänge n_a
2020	2^{192}
2030	2^{224}
2040	2^{256}
für absehbare Zukunft	2^{512}

11.3 Kompressionsfunktionen aus Verschlüsselungsfunktionen

Genauso wenig wie es bekannt ist, ob es effiziente und sichere Blockchiffren gibt, weiß man, ob es kollisionsresistente Kompressionsfunktionen gibt. In der Praxis werden Kompressionsverfahren verwendet, deren Kollisionsresistenz bis jetzt nicht widerlegt wurde. Man kann Kompressionsfunktionen z. B. aus Verschlüsselungsfunktionen konstruieren. Dies wird nun beschrieben.

Wir benötigen eine Blockchiffre mit Blocklänge n und Klartextraum, Schlüsselraum und Schlüsseltextraum $\mathbb{Z}_2^n$. Die Verschlüsslungsfunktionen bezeichnen wir mit $\mathbf{Enc} : \mathbb{Z}_2^n \times \mathbb{Z}_2^n \to \mathbb{Z}_2^n$. Die Hashwerte haben die Länge n. Mit Hilfe von **Enc** kann man auf folgende Weise Kompressionsfunktionen

$$h : \mathbb{Z}_2^n \times \mathbb{Z}_2^n \to \mathbb{Z}_2^n$$

definieren:

$$\begin{aligned} h(x_1, x_2) &= \mathbf{Enc}(x_1, x_2) \oplus x_2, \\ h(x_1, x_2) &= \mathbf{Enc}(x_1, x_2) \oplus x_1 \oplus x_2, \\ h(x_1, x_2) &= \mathbf{Enc}(x_1, x_2 \oplus x_1) \oplus x_2, \\ h(x_1, x_2) &= \mathbf{Enc}(x_1, x_2 \oplus x_1) \oplus x_1 \oplus x_2. \end{aligned}$$

11.4 Hashfunktionen aus Kompressionsfunktionen

Wenn es kollisionsresistente Kompressionsfunktionen gibt, dann gibt es auch kollisionsresistente Hashfunktionen. R. Merkle hat nämlich ein Verfahren beschrieben, wie man aus einer kollisionsresistenten Kompressionsfunktion eine kollisionsresistente Hashfunktion machen kann. Dieses Verfahren beschreiben wir.

Sei

$$g : \mathbb{Z}_2^m \to \mathbb{Z}_2^n$$

eine Kompressionsfunktion und sei

$$r = m - n.$$

Weil g eine Kompressionsfunktion ist, gilt $r > 0$. Eine typische Situation ist die, dass $n = 128$ und $r = 512$ ist. Wir erläutern die Konstruktion von g für $r \geq 2$. Der Fall $r = 1$ bleibt dem Leser als Übung überlassen. Aus g will man eine Hashfunktion

$$h : \mathbb{Z}_2^* \to \mathbb{Z}_2^n$$

konstruieren. Sei also $x \in \mathbb{Z}_2^*$. Vor x wird eine minimale Anzahl von Nullen geschrieben, so dass die neue Länge durch r teilbar ist. An diesen String werden nochmal r Nullen angehängt. Jetzt wird die Binärentwicklung der Länge des originalen Strings x bestimmt.

Ihr werden so viele Nullen vorangestellt, dass ihre Länge durch $r-1$ teilbar ist. Vor jedes $(r-1)*j$-te Zeichen, $j=1,2,3,4\ldots$, dieser Binärentwicklung wird eine Eins geschrieben. Dieser neue String wird wiederum an den Vorigen angehängt. Der Gesamtstring wird in eine Folge

$$x = x_1 x_2 \ldots x_t, \quad x_i \in \mathbb{Z}_2^r, \quad 1 \le i \le t.$$

von Wörtern der Länge r zerlegt. Man beachte, dass alle Wörter, die aus der Binärentwicklung der Länge von x stammen, mit einer Eins beginnen.

Beispiel 11.8 Sei $r=4$, $x=111011$. Zuerst wird x in 0011 1011 verwandelt, damit die Länge durch 4 teilbar ist. An diesen String wird 0000 angehängt. Man erhält 0011 1011 0000. Die Originallänge von x ist 6. Die Binärentwicklung von 6 ist 110. Sie ist schon durch $r-1=3$ teilbar. Wir stellen ihr eine 1 voran, hängen sie an das aktuelle x an und erhalten 0011 1011 0000 1110.

Der Hashwert $h(x)$ wird iterativ berechnet. Man setzt

$$H_0 = 0^n.$$

Das ist der String, der aus n Nullen besteht. Dann bestimmt man

$$H_i = g(H_{i-1} \circ x_i), \quad 1 \le i \le t.$$

Schließlich setzt man

$$h(x) = H_t.$$

Wir zeigen, dass h kollisionsresistent ist, wenn g kollisionsresistent ist. Wir beweisen dazu, dass man aus einer Kollision von h eine Kollision von g bestimmen kann.

Sei also (x, x') eine Kollision von h. Ferner seien $x_1, \ldots, x_t$, $x'_1, \ldots, x'_{t'}$ die zugehörigen Folgen von Blöcken der Länge r, die so wie oben beschrieben konstruiert sind. Die entsprechenden Folgen von Hashwerten seien $H_0, \ldots, H_t$, $H'_0, \ldots, H'_{t'}$.

Weil (x, x') eine Kollision ist, gilt $H_t = H'_{t'}$. Sei $t \le t'$. Wir vergleichen jetzt H_{t-i} mit $H'_{t'-i}$ für $i = 1, 2, \ldots, t-1$. Angenommen, wir finden ein $i < t$ mit

$$H_{t-i} = H'_{t'-i}$$

und

$$H_{t-i-1} \ne H'_{t'-i-1}.$$

Dann gilt

$$H_{t-i-1} \circ x_{t-i} \ne H'_{t'-i-1} \circ x'_{t'-i}$$

und

$$g(H_{t-i-1} \circ x_{t-i}) = H_{t-i} = H'_{t'-i} = g(H'_{t'-i-1} \circ x'_{t'-i}).$$

Dies ist eine Kollision von g.

Sei nun angenommen, dass

$$H_{t-i} = H'_{t'-i} \quad 0 \leq i \leq t.$$

Unten zeigen wir, dass es einen Index i gibt mit $0 \leq i \leq t-1$ und

$$x_{t-i} \neq x'_{t'-i}.$$

Daraus folgt

$$H_{t-i-1} \circ x_{t-i} \neq H'_{t'-i-1} \circ x'_{t'-i}$$

und

$$g(H_{t-i-1} \circ x_{t-i}) = H_{t-i} = H'_{t'-i} = g(H'_{t'-i-1} \circ x'_{t'-i}).$$

Also ist wieder eine Kollision von g gefunden.

Wir zeigen jetzt, dass es einen Index i gibt mit $0 \leq i \leq t-1$ und

$$x_{t-i} \neq x'_{t'-i}.$$

Werden für die Darstellung der Länge von x weniger Wörter gebraucht als für die Darstellung der Länge von x', dann gibt es einen Index i, für den x_{t-i} nur aus Nullen besteht (das ist der String, der zwischen die Darstellung von x und seiner Länge geschrieben wurde) und für den $x'_{t'-i}$ eine führende 1 enthält (weil alle Wörter, die in der Darstellung der Länge von x' vorkommen, mit 1 beginnen).

Werden für die Darstellung der Längen von x und x' gleich viele Wörter gebraucht, aber sind diese Längen verschieden, dann gibt es einen Index i derart, dass x_{t-i} und $x'_{t'-i}$ in der Darstellung der Länge von x bzw. x' vorkommen und verschieden sind.

Sind die Längen von x und x' aber gleich, dann gibt es einen Index i mit $x_{t-i} \neq x'_{t'-i}$, weil x und x' verschieden sind.

Wir haben also gezeigt, wie man eine Kollision der Kompressionsfunktion aus einer Kollision der Hashfunktion gewinnt. Weil der Begriff „kollisionsresistent“ aber nicht formal definiert wurde, haben wir dieses Ergebnis auch nicht als mathematischen Satz formuliert.

11.5 SHA-3

Wie schon erwähnt, gibt es keine nachweislich sicheren kryptographischen Hashfunktionen. Wir skizzieren die Funktionsweise von Keccak, eine Hashfunktion, die im Jahr 2012 als Secure Hash Algorithm 3 (SHA-3) von NIST standardisiert wurde und heute als sicher gilt.

Keccak verwendet einen mit 0 initialisierten Zustandsvektor aus 25 Wörtern mit je $w = 2^l$ Bits. Der Wert $l \in \{0, 1, \ldots 6\}$ ist ein Parameter des Verfahrens. Die Länge des

Zustandsvektors ist also $b = 25w$. Der zweite Parameter ist die Bitlänge n des gewünschten Hash-Wertes. Die Hashlänge liegt zwischen 224 und 512.

Die Eingabe von Keccak ist ein Bitstring x. Er wird zunächst so durch die Bitfolge $100\ldots0$ ergänzt, dass seine Länge durch $r = b - c$ teilbar ist, wobei $c = 2n$ ist.

Keccak verarbeitet seine Eingabe schrittweise. In jedem Schritt wird ein r-Bit-Abschnitt von x mit den ersten r Bits des Zustandsvektors mittels XOR verknüpft und der Zustandsvektor auf diese Weise modifiziert. Danach wird auf den gesamten Zustandsvektor eine Rundenfunktion f angewendet. Die Anzahl der Runden ist $12 + 2l$. Nach der letzten Runde werden die ersten n Bits des Zustandsvektors als Hash-Wert verwendet, falls $n \leq r$ ist. Anderenfalls werden die Bits des Hash-Wertes in mehreren Schritten entnommen und zwar jedesmal maximal r Bit. Zwischen den einzelnen Entnahmeschritten wird wieder die Rundenfunktion angewendet.

11.6 Eine arithmetische Kompressionsfunktion

Wir haben bereits erwähnt, dass nachweislich kollisionsresistente Kompressionsfunktionen nicht bekannt sind. Es gibt aber eine Kompressionsfunktion, die jedenfalls dann kollisionsresistent ist, wenn es schwer ist, diskrete Logarithmen in $(\mathbb{Z}/p\mathbb{Z})^*$ zu berechnen. Sie wurde von Chaum, van Heijst und Pfitzmann erfunden. Wir erläutern diese Kompressionsfunktion.

Sei p eine Primzahl, $q = (p-1)/2$ ebenfalls eine Primzahl, a eine Primitivwurzel mod p und b zufällig gewählt in $\{1, 2, \ldots, p-1\}$. Betrachte folgende Funktion:

$$h : \{0, 1, \ldots, q-1\}^2 \to \{1, \ldots, p-1\}, \quad (x_1, x_2) \mapsto a^{x_1} b^{x_2} \bmod p. \tag{11.1}$$

Dies ist zwar keine Kompressionsfunktion wie in Abschn. 11.1. Aber weil $q = (p-1)/2$ ist, bildet die Funktion Bitstrings (x_1, x_2), deren binäre Länge ungefähr doppelt so groß ist wie die binäre Länge von p, auf Bitstrings ab, die nicht länger sind als die binäre Länge von p. Man kann aus h leicht eine Kompressionsfunktion im Sinne von Abschn. 11.1 konstruieren.

Beispiel 11.9 Sei $q = 11$, $p = 23$, $a = 5$, $b = 4$. Dann ist $h(5, 10) = 5^5 \cdot 4^{10} \bmod 23 = 20 \cdot 6 \bmod 23 = 5$.

Eine Kollision von h ist ein Paar $(x, x') \in \{0, 1, \ldots, q-1\}^2 \times \{0, 1, \ldots, q-1\}^2$ mit $x \neq x'$ und $h(x) = h(x')$. Wir zeigen nun, dass jeder, der leicht eine Kollision von h finden kann, genauso leicht den diskreten Logarithmus von b zur Basis a modulo p ausrechnen kann.

Sei also (x, x') eine Kollision von h, $x = (x_1, x_2)$, $x' = (x_3, x_4)$, $x_i \in \{0, 1, \ldots, q-1\}$, $1 \leq i \leq 4$. Dann gilt

$$a^{x_1} b^{x_2} \equiv a^{x_3} b^{x_4} \bmod p$$

woraus

$$a^{x_1-x_3} \equiv b^{x_4-x_2} \bmod p$$

folgt. Bezeichne mit y den diskreten Logarithmus von b zur Basis a modulo p. Dann hat man also

$$a^{x_1-x_3} \equiv a^{y(x_4-x_2)} \bmod p.$$

Da a eine Primitivwurzel modulo p ist, impliziert diese Kongruenz

$$x_1 - x_3 \equiv y(x_4 - x_2) \bmod (p-1) = 2q. \tag{11.2}$$

Diese Kongruenz hat eine Lösung y, nämlich den diskreten Logarithmus von b zur Basis a. Das ist nur möglich, wenn $d = \gcd(x_4 - x_2, p-1)$ ein Teiler von $x_1 - x_3$ ist (siehe Übung 2.11). Nach Wahl von x_2 und x_4 ist $|x_4 - x_2| < q$. Da $p - 1 = 2q$ ist, folgt daraus

$$d \in \{1, 2\}.$$

Ist $d = 1$, so hat (11.2) eine eindeutige Lösung modulo $p-1$. Man kann also sofort den diskreten Logarithmus als kleinste nicht negative Lösung dieser Kongruenz bestimmen. Ist $d = 2$, so hat die Kongruenz zwei verschiedene Lösungen mod $p-1$ und man kann durch Ausprobieren herausfinden, welche die richtige ist.

Die Kompressionsfunktion aus (11.1) ist also kollisionsresistent, solange die Berechnung diskreter Logarithmen schwierig ist. Die Kollisionsresistenz wurde damit auf ein bekanntes Problem der Zahlentheorie reduziert. Man kann daher der Meinung sein, dass h sicherer ist als andere Kompressionsfunktionen. Weil die Auswertung von h aber modulare Exponentiationen erfordert, ist h auch sehr ineffizient und daher nur von theoretischem Interesse.

11.7 Message Authentication Codes

Kryptographische Hashfunktionen erlauben es zu überprüfen, ob eine Datei verändert wurde. Sie erlauben es aber nicht festzustellen, von wem eine Nachricht kommt. Ein *Message-Authentication-Code (MAC)* macht die Authentizität einer Nachricht überprüfbar.

Definition 11.3 Ein *Message-Authentication-Code-Algorithmus* oder *MAC-Algorithmus* ist ein Tupel (**K**, **M**, **S**, **KeyGen**, **Mac**, **Ver**) mit folgenden Eigenschaften:

1. **K** ist eine Menge. Sie heißt *Schlüsselraum*. Ihre Elemente heißen *Schlüssel*.
2. **M** ist eine Menge. Sie heißt *Nachrichtenraum*. Ihre Elemente heißen *Nachrichten*.
3. **S** ist eine Menge. Sie heißt *MAC-Raum*. Ihre Elemente heißen *MACs*.
4. **KeyGen** ist ein probabilistischer Algorithmus. Er heißt *Schlüsselerzeugungsalgorithmus*.

5. **Mac** ist ein probabilistischer Algorithmus, der zustandsbehaftet sein kann. Er heißt *MAC-Erzeugungsalgorithmus*. Bei Eingabe eines Schlüssels $K \in \mathbf{K}$ und einer Nachricht $M \in \mathbf{M}$ gibt er einen MAC $S \in \mathbf{S}$ zurück.
6. **Ver** ist ein deterministischer Algorithmus. Er heißt *Verifikationsalgorithmus*. Er gibt bei Eingabe eines Schlüssels $K \in \mathbf{K}$, einer Nachricht $M \in \mathbf{M}$ und eines MAC $S \in \mathbf{S}$ einen der Werte $b \in \mathbb{Z}_2$ aus. Der MAC S heißt *gültig*, wenn $b = 1$ ist und andernfalls *ungültig*.
7. Der MAC-Algorithmus verifiziert korrekt: für alle Schlüssel $K \in \mathbf{K}$, jede Nachricht $M \in \mathbf{M}$ und jeden MAC $S \in \mathbf{S}$ gilt $\mathbf{Ver}(K, M, S) = 1$ genau dann, wenn S ein Rückgabewert von $\mathbf{Mac}(K, M)$ ist.

Beispiel 11.10 Eine bekannte Klasse von MACs sind die *Keyed-Hash Message Authentication Codes (HMAC)*. Solche MACs werden aus kryptographischen Hashfunktionen konstruiert. Die Konstruktion wurde von Mihir Bellare, Ran Canetti und Hugo Krawczyk in [9] vorgeschlagen. Sei dazu

$$h : \mathbb{Z}_2^* \to \mathbb{Z}_2^n$$

eine kryptographische Hashfunktion. Dann kann man daraus folgendermaßen einen MAC mit endlichem Schlüsselraum $\mathbf{K} \subset \mathbb{Z}_2^*$, Nachrichtenraum $\mathbb{Z}_2^*$ und MAC-Raum $\mathbb{Z}_2^n$ machen. Der Schlüsselerzeugungsalgorithmus wählt zufällig und gleichverteilt einen Schlüssel $K \in \mathbf{K}$. Der MAC-Erzeugungsalgorithmus erhält als Eingabe einen Schlüssel $K \in \mathbf{K}$ und eine Nachricht $M \in \mathbb{Z}_2^*$. Ist der Schlüssel länger als n so wird er durch $h(K)$ ersetzt. Ist er kürzer als n, wird er so mit Nullen ergänzt, dass seine Länge n ist. Der Ausgabewert ist

$$h((K \oplus \text{opad}) \circ (K \oplus \text{ipad}) \circ M). \tag{11.3}$$

Hierbei ist opad ein String, der binär die Länge n hat und hexadezimal von der Form $0x5c5c5c \ldots 5c5c$ ist. Außerdem ist ipad ein String, der binär die Länge n hat und hexadezimal von der Form $0x363636 \ldots 3636$ ist.

Das folgende Beispiel zeigt, wie man MACs benutzen kann.

Beispiel 11.11 Die Professorin Alice sendet per E-Mail an das Prüfungsamt eine Liste M der Matrikelnummern aller Studenten, die den Schein zur Vorlesung „Einführung in die Kryptographie“ erhalten haben. Das Prüfungsamt muss sicher sein, dass die Nachricht tatsächlich von Alice kommt. Dazu benutzt Alice einen MAC-Algorithmus $(\mathbf{K}, \mathbf{M}, \mathbf{S}, \mathbf{KeyGen}, \mathbf{Mac}, VA)$. Alice und das Prüfungsamt tauschen einen geheimen Schlüssel $K \in \mathbf{K}$ aus. Mit ihrer Liste M schickt Alice auch den MAC $S = \mathbf{E}(K, M)$ an das Prüfungsamt. Bob, der Sachbearbeiter im Prüfungsamt, kann seinerseits den MAC $S' = \mathbf{Mac}(K, M')$ der erhaltenen Nachricht M' berechnen. Er akzeptiert die Nachricht, wenn $S = S'$ ist.

Beispiel 11.12 MACs können auch mit Hilfe von Blockchiffren konstruiert werden. Dabei wird die Nachricht im CBC-Mode verschlüsselt und der letzte Schlüsseltextblock als MAC verwendet.

MACs sind *sicher*, wenn es Angreifern nicht möglich ist, *existentielle Fälschungen* zu konstruieren. Eine existentielle Fälschung ist ein Paar (M, S), das aus einer Nachricht M und einem gültigen MAC S für M besteht. Dieses Paar muss der Angreifer konstruieren ohne den entsprechenden geheimen Schlüssel zu kennen. Sind existentielle Fälschungen unmöglich, so können Angreifer ohne Kenntnis des geheimen Schlüssels auch keine MACs für vorgegebene Nachrichten erzeugen.

Beispiel 11.13 Selbst wenn eine kryptographische Hashfunktion $h : \mathbb{Z}_2^* \to \mathbb{Z}_2^n$ nicht kollisionsresistent ist, kann der entsprechende MAC sicher sein. Das gilt zum Beispiel für den MD5 MAC.

Ein fomales Sicherheitsmodell für MACs kann in Analogie zum formalen Sicherheitsmodell für digitale Signaturen in Abschn. 12.10 formuliert werden.

11.8 Übungen

Übung 11.1 Konstruieren Sie eine Funktion, die eine Einwegfunktion ist, falls das Faktorisierungsproblem für natürliche Zahlen schwer zu lösen ist.

Übung 11.2 Für eine Permutation π in S_3 sei e_π die Bitpermutation für Bitstrings der Länge 3. Bestimmen Sie für jedes $\pi \in S_3$ die Anzahl der Kollisionen der Kompressionsfunktion $h_\pi(x) = e_\pi(x) \oplus x$.

Übung 11.3 Betrachten Sie die Hashfunktion $h : \mathbb{Z}_2^* \to \mathbb{Z}_2^*$, $k \mapsto \lfloor 10000(k(1 + \sqrt{5})/2) \bmod 1) \rfloor$, wobei die Strings k mit den durch sie dargestellten natürlichen Zahlen identifiziert werden und $r \bmod 1 = r - \lfloor r \rfloor$ ist für eine positive reelle Zahl r. Außerdem werden die Bilder durch Voranstellen von Nullen auf die maximal mögliche Länge aller Bilder verlängert.

1. Bestimmen Sie die maximale Länge der Bilder.
2. Geben Sie eine Kollision dieser Hashfunktion an.

Übung 11.4 Erläutern Sie die Konstruktion einer Hashfunktion aus einer Kompressionsfunktion aus Abschn. 11.4 für $r = 1$.