KAPITEL 8

Kryptographische Hash-Funktionen

1. Kryptographische Hash-Funktionen

Hash-Funktionen sind Funktionen, die beliebig lange Zeichenketten in Zeichenketten einer fest vorgegebenen Länge n umwandeln. Wir geben eine mathematische Definition: Ist Σ ein Alphabet, so ist

$$\Sigma^m = \{a_1 a_2 \dots a_m : a_i \in \Sigma\}$$

die Menge der Wörter/Strings/Zeichenketten der Länge m und

$$\Sigma^* = \{a_1 a_2 \dots a_m : a_i \in \Sigma, m \geq 0\} = \bigcup_{m \geq 0} \Sigma^m$$

die Menge aller möglichen Wörter/Strings/Zeichenketten.
Eine **Hash-Funktion** h ist zunächst eine Abbildung

$$h : \Sigma^* \to \Sigma^n,$$

sie bildet also beliebig lange Zeichenketten auf Zeichenketten fest vorgegebener Länge n ab. Für unsere Zwecke wird $\Sigma = \{0, 1\}$ sein, wir deuten Σ^* als Menge aller möglichen Bitfolgen/Bytefolgen/Nachrichten. Zur Motivation geben wir zwei Beispiele:

Beispiele:

(1) Will man testen, ob man beim Abschreiben von Zahlen einen Fehler gemacht hat, kann man die Quersummen bilden: sind sie verschieden, ist ein Fehler passiert.

(2) Beim Komprimieren einer Datei datei mit gzip wird eine 32-Bit-Prüfsumme (crc - cyclic redundancy code) gespeichert, die man mit '`gzip -l -v datei.gz`' erhält. Bei folgendem Beispiel wurde eine mit 10000 Zufallsbytes gefüllte Datei komprimiert:

```
method  crc     date  time  compressed  uncompressed  ratio uncompressed_name
defla 72601657 Jul  3 15:01      10027         10000  -0.1% abc
```

Wurde die gzip-komprimierte Datei verändert/beschädigt, erhält man bei Anwendung von gunzip (meistens) die Meldung '`invalid compressed data--crc error`' und man weiß, dass etwas nicht stimmt.

Für kryptographische Anwendungen braucht man noch weitere Eigenschaften. Wir sprechen von einer **kryptographischen Hash-Funktion**, wenn folgende Forderungen erfüllt sind:

(1) Für jedes $a \in \Sigma^*$ soll sich der Hashwert $h(a)$ schnell und effektiv berechnen lassen.

(2) Zu (allgemeinem) b im Bild von h kann man praktisch kein $a \in \Sigma^*$ finden mit $b = h(a)$. Mit dieser Eigenschaft nennt man h eine Einwegfunktion.

(3) Man kann praktisch keine $a \neq a'$ finden mit $h(a) = h(a')$. Man sagt, h ist (stark) kollisionsresistent.

Man trifft dann auch auf Namen wie Kompressionsfunktion, message digest (MD), kryptographische Prüfsumme, Fingerabdruck, message integrity check (MIC), manipulation detection code (MDC).

Bemerkungen:

(1) Starke Kollisionsresistenz impliziert schwache Kollisionsresistenz: Findet man kein Paar (a, a') mit $h(a) = h(a')$ und $a \neq a'$, so findet natürlich auch zu vorgegebenem a kein $a' \neq a$ mit $h(a) = h(a')$.

Datei: ki_hash.tex. Version vom 22.1.2020

(2) Da Σ^* unendlich, Σ^n endlich ist, ist eine Hash-Funktion $h : \Sigma^* \to \Sigma^n$ nie injektiv. Hat man eine Folge $a_1, a_2, \ldots, a_k$ mit $k > |\Sigma|^n$, so gibt es Indizes $i < j$ mit $h(a_i) = h(a_j)$. Kollisionsresistenz heißt, dass man praktisch kein Gegenbeispiel zur Injektivität von h finden kann. Die Eigenschaft Kollisionsresistenz hängt also wesentlich von den augenblicklichen Rechnermöglichkeiten ab.

Anwendungsbeispiel: Wir nehmen an, A sendet eine Nachricht a an B, B empfängt a'. Wie kann B sicher sein, dass $a' = a$ ist? A sendet ebenfalls den Hashwert $h(a)$, B berechnet $h(a')$. Ist jetzt $h(a) \neq h(a')$, so stimmt etwas nicht. Ist $h(a) = h(a')$, so kann B wegen der Kollisionsresistenz von h ziemlich sicher sein, dass $a = a'$ gilt. Mit kryptographischen Hash-Funktionen kann man also überprüfen, ob eine Nachricht verändert wurde oder nicht.

Wir geben einige aktuelle Hash-Funktionen an, die auch an den Rechnern des Departments Mathematik und unter aktuellen Linux-Versionen verfügbar sind. Man wendet sie auf Dateien an und erhält dann eine Hexadezimalzahl (mit den Ziffern 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F oder 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f) als Hashwert.

Name	Länge des Hashwerts	Hexadezimalstellenzahl	Aufruf
MD5	128 Bits	32	`md5sum`
SHA-1	160 Bits	40	`sha1sum`
SHA-1	160 Bits	40	`sha1sum`
SHA-224	224 Bits	56	`sha224sum`
SHA-256	256 Bits	64	`sha256sum`
SHA-384	384 Bits	96	`sha384sum`
SHA-512	512 Bits	128	`sha512sum`

Beispiel: Wir haben „`Kryptographie I`“ mit folgenden Python3-Befehlen in eine Datei `beispiel` geschrieben:

```
f=open("beispiel","wb")
f.write(b'Kryptographie I')
f.close()
```

Die Datei besteht dann aus der Bytefolge 75, 114, 121, 112, 116, 111, 103, 114, 97, 112, 104, 105, 101, 32, 73, wie man beispielsweise mit `[x for x in b'Kryptographie I']` sehen kann. Anwendung der verschiedenen Hashfunktionen liefert folgende Werte:

```
   md5sum    2d6e7fc83bc0aab53035a4a61b923710
  sha1sum    5544f38d3534f55d6f6facdd038105aaa8633a86
sha224sum    ad3641b036d5eb6ca0237be8ef00b06a9c63c5b39e142b4eba86a9d5
sha256sum    f6e5dda4aabbe9e4fa5cf22399498a7e2c32706d19dcafddb8b3e3b4d6834831
sha384sum    4471a7a6a6d83e60b5605be053444a950a4e0f65afb26f6fb59da8b447e73f34
             e5ecc6c910800a2ff9ed2c19b437293d
sha512sum    7ede53df601b5bf0b67cae83d5730bb69468b62ce0989c2e0cacf3ea1ac78fa3
             eab56621228db307995f6b8c42bad96851612cc1d57b351297fba97fce2bcf3b
```

(Der `sha384sum`- und der `sha512`-Wert wurde jeweils in zwei Zeilen aufgeteilt.)

Python3 hat ein Modul `hashlib`, das verschiedene Hash-Funktionen nach `import hashlib` zur Verfügung stellt. Mit `hashlib.algorithms_available` sieht man diese. Beispiele sind:

`md5, sha1, sha224, sha256, sha384, sha3_224, sha3_256, sha3_384, sha3_512`

Den Hashwert als Hexadezimalzahl erhält man beispielsweise mit diesen Befehlen:

```
hashlib.sha1(b'Kryptographie I').hexdigest()
hashlib.sha1('Kryptographie I'.encode()).hexdigest()
```

(Wendet man dann `int(_,16)` darauf an, erhält man die zugehörige Dezimalzahl.)

Das folgende Beispiel zeigt Eigenschaften, die man bei einer kryptographischen Hash-Funktion gerade nicht haben will.

Beispiel: Wir definieren eine Funktion h, die als Eingabe Zeichenketten $a_1a_2a_3\dots a_n$ annimmt, die aus Blank, A, B, C, …, Z, a, b, c, … z und dem Punkt . bestehen, die als Ausgabe eine 128-Bit-Zahl liefert: Wir identifizieren Blank mit 0, A mit 1, B mit 2, …, Z mit 26, a mit 27, b mit 28, …, z mit 52 und den Punkt . mit 53. Dann bilden wir

$$h(a_1\dots a_n)=\sum_{i=1}^{n} a_i\cdot 54^{n-i} \bmod N$$

mit

$$N=243202555041942261112061453118568326114=(\texttt{B6F71BBB8F273A8F8DFCA5B2B17AE3E2})_{16}.$$

Beispielsweise wird

$$\begin{aligned}
h(\texttt{Ich stimme dem Vorschlag nicht zu.}) &= 72405624906446252930898673420897224461 = \\
&= (\texttt{3678D0ECE89EF7F867B40DFDD272070D})_{16},\\
h(\texttt{Ich stimme dem Vorschlag zu.}) &= 96377428031665635578899230277724489105 = \\
&= (\texttt{48819E11EB3E85500023989FD2B34591})_{16},\\
h(\texttt{Ich stimme dem Vorschlag doch zu.}) &= 86912114272284244704816029031245376341 = \\
&= (\texttt{4162AA123B6AD215ECC47A5EB5496B55})_{16},\\
h(\texttt{Ich stimme dem Vorschlag doch zu.}) &= 72405624906446252930898673420897224461 = \\
&= (\texttt{3678D0ECE89EF7F867B40DFDD272070D})_{16}.
\end{aligned}$$

Man sieht, dass wir zur Zeichenkette „`Ich stimme dem Vorschlag nicht zu.`“ die Zeichenkette „`Ich stimme  dem Vorschlag doch zu.`“ mit dem gleichen h-Wert gefunden haben. Die Funktion h ist also nicht (schwach) kollisionsresistent.

Geburtstagsattacke: Natürlich ist eine Hash-Funktion $h:\Sigma^*\to\Sigma^n$ nie injektiv. Wir haben zuvor gesehen: Wählt man eine Folge $a_1,a_2,\dots,a_k\in\Sigma^*$ und verhalten sich die Werte $h(a_1),h(a_2),h(a_3),\dots,h(a_k)$ wie eine Zufallsfolge, so ist die Wahrscheinlichkeit, dass es Indizes $i<j\le k$ gibt mit $h(a_i)=h(a_j)$, größer als 99 %, falls $k\ge 3.1\sqrt{|\Sigma^n|}=3.1|\Sigma|^{n/2}$ ist. Speichert man die Werte $(h(a_i),a_i)$ in einer Datei und sortiert diese nach $h(a_i)$, so sollte man bei entsprechender Wahl von k eine Kollision finden. Dies nennt man Geburtstagsattacke. Für die Praxis muss daher n (mit $\Sigma=\{0,1\}$) so groß sein, dass eine Geburtstagsattacke praktisch nicht durchzuführen ist.

Beispiel: Wir definieren eine Hash-Funktion h durch

$$h(a)=\text{die ersten 12 Bits von SHA-1}(a)=\text{die ersten 3 Hexadezimalstellen von SHA-1}(a).$$

Wir bestimmen die Hashwerte der ersten 1000 natürlichen Zahlen in Dezimaldarstellung. In den folgenden Fällen trat der gleiche Hashwert auf. (Hier ist $\Sigma = \{0, 1\}$, $n = 12$ und $3.1|\Sigma|^{n/2} \approx 198$.)

SHA-1a('n')	n
0159	161
0159	244
0558	293
0588	337
1368	150
1368	988
93ac	418
93ac	931
acf1	188
acf1	541
b202	406
b202	678
edd6	499
edd6	595

2. SHA-1

Das US-amerikanische National Institute of Standards and Technology (NIST) hat am 1. August 2002 im Dokument 'Federal Information Processing Standards Publication 180-2' (FIPS 180-2) kryptographische Hashfunktionen als Standard festgelegt - Secure Hash Standard. Um einen Eindruck vom Aufbau von Hash-Funktionen zu gewinnen, soll im Folgenden SHA-1 beschrieben werden.

Wir verwenden hier 32-Bit-Worte, also Bitfolgen $(a_1 a_2 \dots a_{31} a_{32})$ der Länge 32. Vermöge

$$(a_1 a_2 \dots a_{31} a_{32}) \quad \longleftrightarrow \quad a = \sum_{i=1}^{32} a_i 2^{32-i}$$

erhält man eine Bijektion mit den ganzen Zahlen a mit $0 \le a \le 2^{32} - 1$. Wir werden jetzt ein paar Operationen einführen: Seien a, b und c Zahlen mit $0 \le a, b, c \le 2^{32} - 1$ und zugehörig die Bitfolgen $(a_1 \dots a_{32})$, $(b_1 \dots b_{32})$ und $(c_1 \dots c_{32})$.

- ROTL sei der zirkuläre Linksshift, d.h.
$$\mathrm{ROTL}((a_1 a_2 \dots a_{32})) = (a_2 a_3 \dots a_{32} a_1).$$
- $c = a \oplus b$ wird durch die komponentenweise Addition modulo 2 definiert, d.h. $c_i \equiv a_i + b_i \bmod 2$.
- $c = a \wedge b$ wird durch das logische UND definiert, d.h. $c_i \equiv a_i b_i \bmod 2$.
- $c = \neg a$ wird durch das logische NICHT definiert, d.h. $c_i \equiv 1 - a_i \bmod 2$.
- $c = a + b$ ist die Addition modulo 2^{32}: $c \equiv a + b \bmod 2^{32}$.

Damit kann man die SHA-1-Funktionen $f_0, f_1, \dots, f_{79}$ definieren. Sie operieren auf 32-Bit-Worten bzw. auf den Zahlen zwischen 0 und $2^{32} - 1$.

$$f_t(x, y, z) = \begin{cases} (x \wedge y) \oplus ((\neg x) \wedge z) & 0 \le t \le 19, \\ x \oplus y \oplus z & 20 \le t \le 39, \\ (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z) & 40 \le t \le 59, \\ x \oplus y \oplus z & 60 \le t \le 79. \end{cases}$$

Weitere werden folgende Konstanten benutzt

$$K_t = \begin{cases} 0x5a827999 = 1518500249 & 0 \le t \le 19, \\ 0x6ed9eba1 = 1859775393 & 20 \le t \le 39, \\ 0x8f1bbcdc = 2400959708 & 40 \le t \le 59, \\ 0xca62c1d6 = 3395469782 & 60 \le t \le 79. \end{cases}$$

$((K_{20}/K_0)^2 \approx 3/2$, $(K_{40}/K_0)^2 \approx 5/2$, $(K_{60}/K_0)^2 \approx 10/2$.)

Wir nehmen an, wir haben eine Nachricht M in Form einer Bytefolge gegeben (1 Byte = 8 Bits), wobei wir uns Bytes durch Zahlen zwischen 0 und $2^8 - 1 = 255$ repräsentiert denken.

Padding - Auffüllen: An die Nachricht werden Bytes angehängt, sodass die Bytelänge der neuen Nachricht durch 64 teilbar ist.

- Sei L die Bytelänge von M.
- Bestimme eine 8-Byte-Darstellung der Zahl $8L$, d.h.

$$8L = \ell_1 \cdot 256^7 + \ell_2 \cdot 256^6 + \cdots + \ell_7 \cdot 256 + \ell_8 \quad \text{mit} \quad 0 \le \ell_i \le 255.$$

- Hänge an M ein Byte 128 an. Dies entspricht der Bitfolge (10000000).
- Bestimme $0 \le k \le 64$ mit $k \equiv -\ell - 9 \bmod 64$. Hänge an M nun k Bytes 0 an.
- Hänge an M die Bytes $\ell_1, \ell_2, \ldots, \ell_8$ an.
- Die Bytezahl des modifizierten M ist jetzt durch 64 teilbar.

Nun wird M in 64-Byte-Blöcke $M_1, M_2, \ldots, M_N$ unterteilt.
Die folgenden Zahlen liefern die Startwerte für die Hashwerte.

$$\begin{aligned} H_0 &= 0x67452301 = 1732584193, \\ H_1 &= 0xefcdab89 = 4023233417, \\ H_2 &= 0x98badcfe = 2562383102, \\ H_3 &= 0x10325476 = 271733878, \\ H_4 &= 0xc3d2e1f0 = 3285377520. \end{aligned}$$

Für jeden 64-Byte-Block M_i, $i = 1, \ldots, N$ werden jetzt nacheinander folgende Schritte durchgeführt:

(1) M_i hat 64 Bytes, dies wird in 16 Blöcke mit je 4 Bytes aufgeteilt, die mit $W_0, \ldots, W_{15}$ bezeichnet werden. W_i hat also 4 Bytes, ist also ein 32-Bit-Wort, das einer Zahl zwischen 0 und $2^{32} - 1$ entspricht.

(2) $W_{16}, \ldots, W_{79}$ werden wie folgt definiert:

$$W_t = \mathrm{ROTL}(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \quad \text{für} \quad 16 \le t \le 79.$$

(3) Man setzt

$$a := H_0, \quad b := H_1, \quad c := H_2, \quad d := H_3, \quad e := H_4.$$

(4) Für $0 \le t \le 79$ führt man folgende Berechnungen durch:

$$\begin{aligned} T &:= \mathrm{ROTL}^5(a) + f_t(b,c,d) + e + K_t + W_t, \\ e &:= d, \\ d &:= c, \\ c &:= \mathrm{ROTL}^{30}(b), \\ b &:= a, \\ a &:= T. \end{aligned}$$

(5) Nun werden die Hash-Werte aktualisiert:

$$H_0 := a + H_0, \quad H_1 := b + H_1, \quad H_2 := c + H_2, \quad H_3 := d + H_3, \quad H_4 := e + H_4.$$

Am Ende hat man H_0, H_1, H_2, H_3, H_4. Jedes H_i ist ein 32-Bit-Wort, hängt man die H_i's aneinander, erhält man einen 160-Bit-Hashwert:

$$\text{SHA-1}(M) = H_0 H_1 H_2 H_3 H_4.$$

Bemerkung: Wir haben eine Maple-Funktion 'SHA_1' nach obigen Anweisungen geschrieben, die allerdings nicht sehr schnell arbeitet.

Beispiel: Die folgenden Beispiele sind offiziel dokumentiert, dienen als auch zum Testen, ob richtig programmiert wurde:

SHA-1('abc') =
= A9993E36 4706816A BA3E2571 7850C26C 9CD0D89D,
SHA-1('abcdbcdecdefdefgefghfghighijhijkijkljklmklmnlmnomnopnopq') =
= 84983E44 1C3BD26E BAAE4AA1 F95129E5 E54670F1.

3. Parametrisierte Hash-Funktionen - MACs

Mit kryptographischen Hash-Funktionen kann man überprüfen, ob eine Nachricht verändert wurde. Man kann aber nicht feststellen, von wem die Nachricht stammt. Will man die Authentizität einer Nachricht überprüfbar machen, kann man parametrisierte Hash-Funktionen h_K (K Element eines Schlüsselraums) benutzen. Sie werden auch **message authentication codes** – MACs genannt.

Wir geben zwei Anwendungsbeispiele:

Beispiel: Ein Benutzer A berechnet den MAC seiner Dateien (mit seinem Schlüssel K) und speichert die Werte in einer Tabelle. Verändert ein Eindringling B die Dateien von A, so kann B ohne Kenntnis des zugehörigen Schlüssels von A die neuen Hashwerte für die Tabelle nicht berechnen. A bemerkt also, dass seine Dateien verändert wurden durch Berechnung der Hashwerte mit seinem Schlüssel K.

Beispiel: Ein Professor will eine Liste a mit Prüfungsergebnissen an das Prüfungsamt schicken. Die Liste muss nicht geheimgehalten werden, sollte aber auch nicht verändert werden können. Der Professor und das Prüfungsamt haben einen MAC h_K und einen Schlüssel K vereinbart. Der Professor schickt zusammen mit der Liste a auch den Hashwert $h_K(a)$ an das Prüfungsamt. Dort wird der Wert $h_K(a')$ der erhaltenen Liste a' berechnet. Nur wenn $h_K(a) = h_K(a')$ gilt, akzeptiert das Prüfungsamt die Liste.

Wir geben zwei einfache Beispiele, wie man MACs konstruieren kann:

Beispiel: Man nimmt ein symmetrisches Verschlüsselungsverfahren mit Verschlüsselungsfunktionen E_K, eine Hash-Funktion h und setzt $h_K(a) = E_K(h(a))$.

Beispiel: Man verwendet als Verschlüsselungsfunktionen E_K ein symmetrisches Verschlüsselungsverfahren im CBC-Modus (cipherblock chaining mode). $h_K(a)$ wird der letzte Block von $E_K(a)$.