Kapitel 7
Kryptographisch sichere Hashverfahren

§1: Nochmals elektronische Unterschriften

Elektronische Unterschriften, so wie wir sie bislang kennen, sind ungefähr so aufwendig wie die Verschlüsselung eines Dokuments mit einem asymmetrischen Verfahren. Da niemand ein längeres Dokument blockweise mit einem asymmetrischen Verfahren verschlüsselt, sollte klar sein, daß es auch niemand mit einem solchen Verfahren unterzeichnet. Genauso, wie man bei der Verschlüsselung das asymmetrische Verfahren nur zum Schlüsselaustausch verwendet, möchte man auch beim Unterschreiben das asymmetrische Verfahren nur ein einziges Mal anwenden müssen. Dies wird dadurch möglich, daß man nicht die Nachricht selbst unterschreibt, sondern nur einen daraus berechneten geeigneten Hashwert.

Allgemein ist ein Hashwert eine Art Prüfsumme, die von der gesamten Nachricht abhängt; wohlbekannt sind etwa die Prüfziffern am Ende eines Artikelcodes oder einer ISBN. Bei der Europäischen Artikelnummer EAN, die zumindest als Strichcode auf praktisch allen Waren zu finden ist, wird die letzte Ziffer so gewählt, daß eine durch zehn teilbare Zahl entsteht, wenn man die Summe bildet aus einmal der ersten, dreimal der zweiten, einmal der dritten, dreimal der vierten Ziffer *usw.* Bei der Internationalen Standardbuchnummer ISBN wird die erste der zehn Ziffern mit zehn multipliziert, die zweite mit neun *usw.* und die Summe muß durch elf teilbar sein. Falls man dazu eine zehnte „Ziffer“ zehn braucht, wird diese als „X“ geschrieben. Ähnlich aufgebaut sind auch die Hashfunktionen, die Informatiker in Suchalgorithmen verwenden.

Solche Prüfsummen sind für die Fehlererkennung bei Scannerkassen oder bei Buchbestellungen gut geeignet, da sie die typischen Flüchtigkeitsfehler mit hoher Wahrscheinlichkeit erkennen. Für kryptographische Anwendungen sind sind sie allerdings völlig unbrauchbar, denn hier haben wir es mit intelligenten Gegnern zu tun, die bereit sind, einen großen Aufwand zu treiben um zu einem unterschriebenen Hashwert eine zweite Nachricht mit dem gleichen Hashwert zu konstruieren um dann zu behaupten, die Unterschrift gehöre zu dieser zweiten Nachricht.

Von einer kryptographisch brauchbaren Hashfunktion müssen wir daher fordern, daß es rechnerisch nicht mit vertretbarem Aufwand möglich sein darf, zu einem gegebenen Hashwert einen Text zu konstruieren. Die Hashfunktion muß also, genau wie ein symmetrisches Kryptoverfahren, mit Konfusion und Diffusion arbeiten, so daß möglichst jedes Bit des Texts jedes Bit des Hashwerts in einer schwer durchschaubaren Weise beeinflußt.

Dies legt es nahe, den Hashwert über ein symmetrisches Kryptoverfahren zu berechnen: Man nimmt etwa den ersten Block als Schlüssel, verschlüsselt damit den zweiten, nimmt das Ergebnis als Schlüssel zur Verschlüsselung des dritten und so weiter; die Verschlüsselung des letzten Blocks ist dann der Hashwert.

Ein solches Verfahren ist allerdings gleichzeitig zu aufwendig und zu unsicher: Da jeder Block der Nachricht zum Endergebnis beiträgt, erhalten wir automatisch eine deutlich Reduktion der Redundanz; wir brauchen daher nicht so viele oder nicht so aufwendige Runden pro Block wie bei der Verschlüsselung eines einzelnen Blocks.

Gleichzeitig wäre bei einer solchen Vorgehensweise die Sicherheit drastisch reduziert, wenn wie der nächste Paragraph zeigen wird, brauchen wir für Hashverfahren bei gleicher Sicherheit eine doppelt so große Blocklänge wie bei Verschlüsselungsverfahren.

§2: Das Geburtstagsparadoxon

Der Grund dafür ist das sogenannte „Geburtstagsparadoxon“: Angenommen, in einem Raum befinden sich n Personen. Wie groß ist die

Wahrscheinlichkeit dafür, daß zwei davon am gleichen Tag Geburtstag haben?

Um diese Frage wirklich beantworten zu können, müßte man die (recht inhomogene) Verteilung der Geburtstage über das Jahr kennen; wir beschränken uns stattdessen auf ein grob vereinfachtes Modell ohne Schaltjahre mit 365 gleich wahrscheinlichen Geburtstagen. Dann ist die Wahrscheinlichkeit dafür, daß von n Personen keine zwei am gleichen Tag Geburtstage haben,

$$\prod_{k=0}^{n-1}\left(1-\frac{k}{365}\right)\,,$$

denn für eine Person ist das überhaupt keine Bedingung, und jede weitere Person muß die Geburtstage der schon betrachteten Personen vermeiden. (Da der Faktor mit $k = 365$ verschwindet, wird die Wahrscheinlichkeit für $n > 365$ zu null, wie es nach dem DIRICHLETschen Schubfachprinzip auch sein muß.)

Nachrechnen ergibt für $n = 23$ ungefähr den Wert 0,4927; bei 23 Personen liegt also die Wahrscheinlichkeit für zwei gleiche Geburtstage bei 50,7%. Tatsächlich dürfte sie noch deutlich höher liegen, denn bei Geburtstagen ist die Annahme einer Gleichverteilung sicherlich falsch.

Bei einer guten Hashfunktion allerdings sollten die Hashwerte in sehr guter Näherung gleichverteilt sein; falls es N mögliche Hashwerte gibt, liegt die Wahrscheinlichkeit dafür, daß unter n Nachrichten zwei zum selben führen daher bei

$$p_n = \prod_{k=0}^{n-1}\left(1-\frac{k}{N}\right)\,.$$

Da wir uns für große Werte von N interessieren, können wir davon ausgehen, daß

$$\left(1-\frac{1}{N}\right)^N \approx e \quad\text{und}\quad \left(1-\frac{1}{N}\right) \approx e^{-1/N}$$

ist; für nicht zu große Werte von k ist dann auch

$$\left(1-\frac{k}{N}\right) \approx e^{-k/N}$$

und für nicht zu große Werte von n gilt

$$p_n = \prod_{k=0}^{n-1}\left(1-\frac{k}{N}\right) \approx \prod_{k=0}^{n-1} e^{-k/N} = e^{-\frac{1}{N}\sum_{k=0}^{n-1} k} = e^{-\frac{n(n-1)}{2N}}\,.$$

Für $N = 365$ etwa ergibt dies den Näherungswert $p_{23} \approx 0{,}499998$ für den korrekten Wert 0,4927.

Wenn wir im Exponenten noch den Term $n(n-1)$ durch n^2 approximieren, können wir abschätzen, für welches n die Wahrscheinlichkeit p_n einen vorgegebenen Wert erreicht:

$$e^{-\frac{n^2}{2N}} = p \iff \frac{n^2}{2N} = -\ln p \iff n = \sqrt{-2N\ln p}\,.$$

Damit liegt p_n bei etwa 50%, falls $n \approx \sqrt{2N\ln 2} \approx 1{,}177\sqrt{N}$ ist; für $N = 365$ ergibt dies die immer noch recht gute Näherung 22,494.

Für $p = 1/1000$ ergibt sich $n \approx 3{,}717\sqrt{N}$, für $p = 999/1000$ entsprechend $n \approx 0{,}0447\sqrt{N}$. Die Wahrscheinlichkeit dafür, daß es unter n Nachrichten zwei mit demselben Hashwert gibt, wechselt also bei der Größenordnung $n \approx \sqrt{N}$ ziemlich schnell von sehr unwahrscheinlich zu sehr wahrscheinlich.

Damit ist klar, daß bei einem kryptographisch brauchbares Hashverfahren die Zahl N der möglichen Hashwerte so groß sein muß, daß die Erzeugung von $\sqrt{N}$ verschiedenen Nachrichten rechnerisch unmöglich ist. Ansonsten könnte nämlich ein Gegner zwei verschiedene Texte a und b erzeugen, von denen a so ist, daß ihn das Opfer unterschreibt, b aber für dieses äußerst nachteilig wäre. Dazu könnte er jeweils etwa $\sqrt{N}$ sinngleiche Modifikationen a_i und b_j erzeugen, indem er beispielsweise unabhängig voneinander an jedem Zeilenenden entweder ein Leerzeichen einfügt oder auch nicht, was bei z Zeilen bereits 2^z Varianten ergibt; genauso könnte er Kombinationen aus Leerzeichen und Rücktaste einfügen oder auch nicht, Tabulatoren durch Leerzeichen ersetzen oder auch nicht, und so weiter. Wie wir gerade gesehen haben, hätte er dann eine gute Chance, daß ein a_i denselben Hashwert hätte wie ein b_j; jede Unterschrift unter a_i wäre gleichzeitig eine unter b_j.

Die Schlüssellänge K eines symmetrischen Kryptoverfahrens wird so gewählt, daß die 2^K Schlüssel nicht mit realistischem Aufwand durchprobiert werden können. Bei einem kryptographisch sicheren Hashverfahren muß dementsprechend die Länge L des Hashwerts so gewählt werden, daß niemand mit realistischem Aufwand $\sqrt{2^L} = 2^{L/2}$ Nachrichten erzeugen kann, d.h. bei gleichen Sicherheitsanforderungen muß ein Hashwert etwa doppelt so lang sein wie ein Schlüssel eines symmetrischen Kryptosystems. Da AES mit Schlüssellängen 128, 192 und 256 arbeitet, sollte man also entsprechend mit Hashwerten der Länge 256, 384 und 512 arbeiten; Hashwerte mit nur 128 Bit sind genau wie Kryptoverfahren mit 64 Bit-Schlüsseln heute nicht mehr sicher.

§3: Die Familie der SHA-Algorithmen

Da es mit kryptographisch sicheren Hashfunktionen deutlich weniger Erfahrung gibt als mit Verschlüsselung, liest sich die bisherige Geschichte solcher Funktionen eher enttäuschend: Zu den meisten Verfahren wurden über kurz oder lang Angriffe gefunden.

In der alltäglichen Praxis wurden bis vor kurzer Zeit hauptsächlich zwei Hashverfahren verwendet: RIPEMD-160 und SHA-1, die beide mit 160 Bit Hashwerten arbeiten. In der Tat gibt es eine ganze Reihe chipkartenbasierter Systeme für elektronische Unterschriften, die nur mit Hashwerten dieser Länge arbeiten konnten. Die Sicherheit solcher Unterschriften entsprach nur der eines Kryptoverfahrens mit Schlüssellänge 80 war somit nach heutigen Anforderungen recht gering.

Von daher sollte es niemanden verwundern, daß in den letzten Jahren zunehmend Ansätze für Kollisionsattacken gegen die beiden Verfahren publiziert wurden und daß die Bundesnetzagentur nun längere Hashwerte vorschreibt. Genau wie vor einigen Jahren bei der „Schallgrenze“ 1024 Bit für RSA stieß sie auch hier wieder auf erbitterten Widerstand einiger Anwender; nach den Empfehlungen für 2010 sind nun aber für neue Unterschriften nur noch Verfahren erlaubt, die Hashwerte mit Mindestlänge 224 liefern; das etwa der Sicherheit von Triple-DES. Die einzigen dafür zugelassenen Verfahren sind derzeit die entsprechenden

Algorithmen aus der SHA-Familie, mit denen wir uns daher in diesem Paragraphen beschäftigen wollen.

Der *Secure Hash Algorithm* SHA wurde im Januar 1992 veröffentlicht und am 11. Mai 1993 als amerikanischer Standard FIPS 180 verkündet. Wegen einer (nie publizierten) „technischen Schwäche“ musste auch dieser Algorithmus alsbald nachgebessert werden; am 11. Juli 1994 wurde die Modifikation SHA-1 als Nachfolgestandard FIPS 180-1 eingesetzt. Dessen Nachfolger FIPS 180-2 vom 1. August 2002 ließ SHA-1 unangetastet, fügte aber drei neue, ähnlich aufgebaute Algorithmen SHA-256, SHA-384 und SHA-512 mit längeren Hashwerten dazu. In einem Zusatz vom 15. Februar 2004 wurde schließlich auch noch eine Variante SHA-224 normiert. Am 15. März 2006 wurden die amerikanischen Bundesbehörden angewiesen, künftig nur noch die neueren Versionen mit mindestens 224 Bit zu benutzen. Die derzeitig gültige Version FIPS 180-3 vom Oktober 2008 stellt im wesentlich nur bekanntes Material in anderer Weise zusammen.

Die fünf Algorithmen sind sehr ähnlich aufgebaut; SHA-1, SHA-224 und SHA-256 arbeiten mit 32-Bit-Wörtern und Blöcken der Länge 512, bei SHA-384 und SHA-512 verdoppeln sich diese Längen. Die Nachrichten, die SHA-1 bis SHA-256 verarbeiten können, müssen kürzer sein als 2^{64} Bit, also etwa zwei Millionen Terabyte; für SHA-384 und SHA-512 liegt die Grenze bei 2^{128} Bit. Da die gesamte in unserem Universum enthaltene Information nach Schätzungen der Physiker bei etwa 10^{120} Bit liegt, dürfte dies für alle praktischen Zwecke ausreichen.

Die Maximallänge spielt eine Rolle bei der Vorverarbeitung der Nachricht: Ist M die zu verarbeitende Nachricht, bestehend aus ℓ Bit, so wird am Ende ein Bit „1“ eingefügt, dann k Nullen und schließlich noch die Zahl ℓ als Block von 64 *bzw.* 128 Bit. Die Zahl k wird als kleinste nichtnegative ganze Zahl gewählt, für die $\ell + 1 + k + 64$ durch 512 teilbar ist für SHA-1, SHA-224 und SHA-256 *bzw.* für die $\ell + 1 + k + 128$ durch 1024 teilbar ist für SHA-384 und SHA-512, so daß in jedem Fall die Länge der Nachricht ein ganzzahliges Vielfaches der Blocklänge ist.

Damit läßt sich die Nachricht nun aufteilen in Blöcke $M^{(1)}, M^{(2)}, \ldots$;

jeder dieser Blöcke wiederum wird aufgeteilt in Wörter $M_0^{(i)}, \ldots, M_{15}^{(i)}$. (Man beachte, daß bei jedem der fünf Algorithmen die Blocklänge gleich der 16-fachen Wortlänge ist.)

Jeder der Algorithmen startet mit seinem eigenen Anfangshashwert. Bei SHA-1 besteht dieser aus den willkürlich *(?)* festgelegten fünf (hexadezimal angegebenen) Wörtern

$$H_0^{(0)} = 67452301, \quad H_1^{(0)} = \texttt{EFCDAB89}, \quad H_2^{(0)} = 98\texttt{BADCFE},$$
$$H_3^{(0)} = 10325476, \quad H_4^{(0)} = \texttt{C3D2E1F0}\,,$$

ähnlich auch bei SHA-224 mit

$$H_0^{(0)} = \texttt{C1059ED8}, \quad H_1^{(0)} = 367\texttt{CD507}, \quad H_2^{(0)} = 3070\texttt{DD17},$$
$$H_3^{(0)} = \texttt{F70E5939}, \quad H_4^{(0)} = \texttt{FFC00B31}, \quad H_5^{(0)} = 68581511,$$
$$H_6^{(0)} = 64\texttt{F98FA7}, \quad H_7^{(0)} = \texttt{BEFA4FA4}\,.$$

Bei den drei Standards mit längeren Hashwerten zeigt sich der Trend zu nachvollziehbaren mathematischen Definitionen: Hier gibt es jeweils acht Wörter $H_0^{(0)}, \ldots, H_7^{(0)}$, die über die hexadezimal geschriebenen gebrochenen Anteile der Quadratwurzeln von Primzahlen p definiert sind; es geht also um die hexadezimal geschriebenen ganzen Zahlen $\left[2^r\{\sqrt{p}\}\right]$, wobei $r = 32$ *bzw.* 64 die Wortlänge des jeweiligen Algorithmus ist. Für SHA-256 und SHA-512 nimmt man die erste bis achte Primzahl, für SHA-384 die neunte bis sechzehnte. In der nachstehenden Tabelle sind diese Werte zu finden, wobei für SHA-256 natürlich nur die jeweils linke Hälfte relevant ist.

Daneben gibt es noch Konstanten, die auch bei SHA-1 wieder willkürlich *(?)* festgelegt sind als

$$K_t = \begin{cases} 5A827999 & \text{für } \ 0 \le t \le 19 \\ 6\texttt{ED9EBAL} & \text{für } 20 \le t \le 39 \\ 8\texttt{FLBBCDC} & \text{für } 40 \le t \le 59 \\ \texttt{CA62C1D6} & \text{für } 60 \le t \le 99 \end{cases},$$

und für die drei anderen Algorithmen gegeben sind durch die ersten Bits der gebrochenen Anteile der Kubikwurzeln der ersten Primzahlen. Für SHA-224 und SHA-256 nimmt man die ersten 32 Bit und die

i	p_i	$H_i^{(0)} = \left[2^r\{\sqrt{p_i}\}\right]$
1	2	6A09E667 F3BCC908
2	3	BB67AE85 84CAA73B
3	5	3C6EF372 FE94F82B
4	7	A54FF53A 5F1D36F1
5	11	510E527F ADE682D1
6	13	9B05688C 2B3E6C1F
7	17	1F83D9AB FB41BD6B
8	19	5BE0CD19 137E2179
9	23	CBBB9D5D C1059ED8
10	29	629A292A 367CD507
11	31	9159015A 3070DD17
12	37	152FECD8 F70E5939
13	41	67332667 FFC00B31
14	43	8EB44A87 68581511
15	47	DB0C2E0D 64F98FA7
16	53	47B5481D BEFA4FA4

ersten 64 Primzahlen, für SHA-384 und SHA-512 entsprechend die ersten 64 Bit und die ersten achtzig Primzahlen. In jedem Fall hat man also eine Folge von Wörtern $K_0, K_1, \ldots$ bis K_{63} *bzw.* K_{79}. Die Hexadezimalentwicklungen der gebrochenen Anteile der Kubikwurzeln der ersten achtzig Primzahlen sind unten in der Tabelle zu finden, wobei wieder darauf zu achten ist, daß K_t für SHA-224 und SHA-256 nur aus den ersten 32 Bit (oder acht Hexadezimalziffern) der angegebenen Werte besteht.

Außer diesen Konstanten sind für die fünf Algorithmen noch Funktionen definiert, die später im Konfusionsschritt eingesetzt werden; sie verknüpfen jeweils drei Wörter zu einem vierten, indem die angegebenen logischen Operationen bitweise angewendet werden.

In allen fünf Algorithmen wird die Funktion

$$Ch(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$$

eingesetzt, in der $\oplus$ für die Addition in $\mathbb{F}_2$ oder (äquivalent) das exklusive Oder steht, hat für wahres x den Wahrheitswert $y \oplus$ *falsch*, also

i	p_i	$K_t = \lfloor 2^r \{\sqrt[3]{p_i}\} \rfloor$	i	p_i	$K_t = \lfloor 2^r \{\sqrt[3]{p_i}\} \rfloor$
1	2	428A2F98 D728AE22	41	179	A2BFE8A1 4CF10364
2	3	71374491 23EF65CD	42	181	A81A664B BC423001
3	5	B5C0FBCF EC4D3B2F	43	191	C24B8B70 D0F89791
4	7	E9B5DBA5 8189DBBC	44	193	C76C51A3 0654BE30
5	11	3956C25B F348B538	45	197	D192E819 D6EF5218
6	13	59F111F1 B605D019	46	199	D6990624 5565A910
7	17	923F82A4 AF194F9B	47	211	F40E3585 5771202A
8	19	AB1C5ED5 DA6D8118	48	223	106AA070 32BBD1B8
9	23	D807AA98 A3030242	49	227	19A4C116 B8D2D0C8
10	29	12835B01 45706FBE	50	229	1E376C08 5141AB53
11	31	243185BE 4EE4B28C	51	233	2748774C DF8EEB99
12	37	550C7DC3 D5FFB4E2	52	239	34B0BCB5 E19B48A8
13	41	72BE5D74 F27B896F	53	241	391C0CB3 C5C95A63
14	43	80DEB1FE 3B1696B1	54	251	4ED8AA4A E3418ACB
15	47	9BDC06A7 25C71235	55	257	5B9CCA4F 7763E373
16	53	C19BF174 CF692694	56	263	682E6FF3 D6B2B8A3
17	59	E49B69C1 9EF14AD2	57	269	748F82EE 5DEFB2FC
18	61	EFBE4786 384F25E3	58	271	78A5636F 43172F60
19	67	FC19DC68 2B8CD5B5	59	277	84C87814 A1F0AB72
20	71	240CA1CC 77AC9C65	60	281	8CC70208 1A6439EC
21	73	2DE92C6F 592B0275	61	283	90BEFFFA 23631E28
22	79	4A7484AA 6EA6E483	62	293	A4506CEB DE82BDE9
23	83	5CB0A9DC BD41FBD4	63	307	BEF9A3F7 B2C67915
24	89	76F988DA 831153B5	64	311	C67178F2 E372532B
25	97	983E5152 EE66DFAB	65	313	CA273ECE EA26619C
26	101	A831C66D 2DB43210	66	317	D186B8C7 21C0C207
27	103	B00327C8 98FB213F	67	331	EADA7DD6 CDE0EB1E
28	107	BF597FC7 BEEF0EE4	68	337	F57D4F7F EE6ED178
29	109	C6E00BF3 3DA88FC2	69	347	06F067AA 72176FBA
30	113	D5A79147 930AA725	70	349	0A637DC5 A2C898A6
31	127	06CA6351 E003826F	71	353	113F9804B EF90DAE
32	131	14292967 0A0E6E70	72	359	1B710B35 131C471B
33	137	27B70A85 46D22FFC	73	367	28DB77F5 23047D84
34	139	2E1B2138 5C26C926	74	373	32CAAB7B 40C72493
35	149	4D2C6DFC 5AC42AED	75	379	3C9EBE0A 15C9BEBC
36	151	53380D13 9D95B3DF	76	383	431D67C4 9C100D4C
37	157	650A7354 8BAF63DE	77	389	4CC5D4BE CB3E42B6
38	163	766A0ABB 3C77B2A8	78	397	597F299C FC657E2A
39	167	81C2C92E 47EDAEE6	79	401	5FCB6FAB 3AD6FAEC
40	173	92722C85 1482353B	80	409	6C44198C 4A475817

denselben Wert wie y. Ist x dagegen falsch, erhalten wir *falsch* $\oplus$ z, also den Wahrheitswert von z. In SHA-1, SHA-224 und SHA-256 wird sie bitweise auf Wörter der Länge 32 angewandt, bei SHA-384 und SHA-512 auf solche der Länge 64.

Ebenfalls allen Algorithmen gemeinsam ist die Mehrheitsfunktion $Maj(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$, die in derselben Weise bitweise angewandt wird. Falls genau zwei ihrer Eingabebits gesetzt sind, ist genau eine der drei Klammern eins, also auch das Ergebnis. Sind alle drei Eingabebits gesetzt, erhalten wir eine Summe von drei Einsen, also ebenfalls eins. In allen anderen Fällen sind alle drei Summanden null, also auch das Ergebnis.

Nur SHA-1 arbeitet mit der Funktion $Maj(x, y, z) = x \oplus y \oplus z$, deren Ergebnisbit genau dann gleich eins ist, wenn eine ungerade Anzahl der drei Eingabebits gleich eins ist.

Zur Diffusion verwenden alle fünf Algorithmen Verschiebungen. Die zyklischen Verschiebungen nach rechts und links bezeichnen wir mit ROTR und ROTL, die entsprechenden nichtzyklischen Verschiebungen mit SHL und SHR. Diese Operatoren verschieben um jeweils ein Bit; für Verschiebungen um größere Distanzen sorgen ihre Potenzen.

SHA-1 verwendet diese Operatoren direkt, bei den neueren Algorithmen gibt es stattdessen vier Funktionen auf Wörtern der jeweiligen Länge, die mehrere dieser Verschiebungen additiv kombinieren. Bei SHA-224 und SHA-256 sind dies

$$\begin{aligned}
\Sigma_0^{(256)}(x) &= \mathrm{ROTR}^2(x) \oplus \mathrm{ROTR}^{13}(x) \oplus \mathrm{ROTR}^{22}(x),\\
\Sigma_1^{(256)}(x) &= \mathrm{ROTR}^6(x) \oplus \mathrm{ROTR}^{11}(x) \oplus \mathrm{ROTR}^{25}(x),\\
\sigma_0^{(256)}(x) &= \mathrm{ROTR}^7(x) \oplus \mathrm{ROTR}^{18}(x) \oplus \mathrm{SHR}^3(x),\\
\sigma_1^{(256)}(x) &= \mathrm{ROTR}^{17}(x) \oplus \mathrm{ROTR}^{19}(x) \oplus \mathrm{SHR}^{10}(x),
\end{aligned}$$

bei SHA-384 und SHA-512

$$\begin{aligned}
\Sigma_0^{(512)}(x) &= \mathrm{ROTR}^{28}(x) \oplus \mathrm{ROTR}^{34}(x) \oplus \mathrm{ROTR}^{39}(x),\\
\Sigma_1^{(512)}(x) &= \mathrm{ROTR}^{14}(x) \oplus \mathrm{ROTR}^{18}(x) \oplus \mathrm{ROTR}^{41}(x),\\
\sigma_0^{(512)}(x) &= \mathrm{ROTR}^1(x) \oplus \mathrm{ROTR}^8(x) \oplus \mathrm{SHR}^7(x),\\
\sigma_1^{(512)}(x) &= \mathrm{ROTR}^{19}(x) \oplus \mathrm{ROTR}^{61}(x) \oplus \mathrm{SHR}^6(x).
\end{aligned}$$

Die weitere Vorgehensweise ist bei den neueren Algorithmen etwas anders als bei SHA-1; wir betrachten diese daher getrennt. In allen Fällen gehen wir davon aus, daß die Nachricht als Folge von Blöcken

$M^{(1)}, M^{(2)}, \ldots$ vorliegt gemäß der eingangs erläuterten Vorverarbeitung.

Der Algorithmus SHA-1 unterwirft diese Blöcke den folgenden Verarbeitungsschritten:

1. Setze

$$W_t = \begin{cases} M_t^{(i)} & \text{für } 0 \le t \le 15 \\ \text{ROTL}(W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) & \text{für } 16 \le t \le 79 \end{cases}.$$

Dies ist offensichtlich ein Diffusionsschritt.

2. Initialisiere die fünf internen Variablen mit den fünf Hashwerten der vorigen Runde (für die erste Runde wurden die Hashwerte $H_j^{(0)}$ oben definiert):

$$a \leftarrow H_0^{(i-1)}, \quad b \leftarrow H_1^{(i-1)}, \quad c \leftarrow H_2^{(i-1)},$$
$$d \leftarrow H_3^{(i-1)}, \quad e \leftarrow H_4^{(i-1)}$$

3. Der Konfusionsschritt: Führe für $t = 0$ bis $t = 79$ die folgenden Anweisungen aus:

$$T \leftarrow \text{ROTL}^5(a) + f_t(b,c,d) + K_t + W_t, \quad e \leftarrow d, \quad d \leftarrow c,$$
$$c \leftarrow \text{ROTL}^{30}(b), \quad b \leftarrow a, \quad a \leftarrow T,$$

wobei $f_t = \begin{cases} Ch & \text{für} \quad \le t \le 19 \\ Maj & \text{für } 40 \le t \le 59 \text{ ist.} \\ Maj & \text{sonst} \end{cases}$

(Die Konstanten K_t wurden oben definiert.)

4. Berechne den Hashwert der Runde:

$$H_0^{(i)} \leftarrow a + H_0^{(i-1)}, \quad H_1^{(i)} \leftarrow b + H_1^{(i-1)}, \quad H_2^{(i)} \leftarrow c + H_2^{(i-1)},$$
$$H_3^{(i)} \leftarrow d + H_3^{(i-1)}, \quad H_4^{(i)} \leftarrow e + H_4^{(i-1)}$$

Ergebnis des Algorithmus sind die aneinandergesetzten Hashwerte der letzten Runde.

Die Algorithmen SHA-224, SHA-256, SHA-382 und SHA-512 unterscheiden sich abgesehen von Längenparametern und den bereits definierten Anfangskonstanten kaum voneinander. Für jeden Block $M^{(i)}$,

aufgeteilt in die Wörter $M_0^{(i)}$ bis $M_{15}^{(i)}$, werden die folgenden Operationen durchgeführt:

1. Setze

$$W_t = \begin{cases} M_t^{(i)} & \text{für } 0 \le t \le 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16} & \text{für } 16 \le t \le r \end{cases},$$

wobei $r = 63$ für SHA-224 und SHA-256 und $r = 80$ sonst.

Das Pluszeichen steht hier folgendermaßen zu interpretieren: Die einzelnen Summanden werden als ganze Zahlen zwischen Null und 2^{32} *bzw.* 2^{64} interpretiert und dann als ganze Zahlen addiert. Das Ergebnis wird modulo 2^{32} *bzw.* 2^{64} betrachtet, d.h. ein etwaiger Überlauf wird ignoriert.

Bei σ_0 und σ_1 ist darauf zu achten, daß es sich hier bei SHA-224 und SHA-256 um $\sigma_0^{(256)}$ und $\sigma_1^{(256)}$ handelt, sonst aber um $\sigma_0^{(512)}$ und $\sigma_1^{(512)}$.

2. Initialisiere die acht internen Variablen mit den acht Hashwerten der vorigen Runde (für die erste Runde wurden die Hashwerte $H_j^{(0)}$ oben definiert):

$$a \leftarrow H_0^{(i-1)}, \quad b \leftarrow H_1^{(i-1)}, \quad c \leftarrow H_2^{(i-1)}, \quad d \leftarrow H_3^{(i-1)},$$
$$e \leftarrow H_4^{(i-1)}, \quad f \leftarrow H_5^{(i-1)}, \quad g \leftarrow H_6^{(i-1)}, \quad h \leftarrow H_7^{(i-1)}$$

3. Der Konfusionsschritt: Führe für $t = 0$ bis $t = r$ (also je nach Algorithmus 63 oder 79) die folgenden Anweisungen aus:

$$T_1 \leftarrow h + \Sigma_1(e) + Ch(e,f,g) + K_t + W_t$$
$$T_2 \leftarrow \Sigma_0(a) + Maj(a,b,c), \quad h \leftarrow g, \quad g \leftarrow f, \quad f \leftarrow e,$$
$$e \leftarrow d + T_1, \quad d \leftarrow c,$$
$$c \leftarrow b, \quad b \leftarrow a, \quad a \leftarrow T_1 + T_2.$$

4. Berechne den Hashwert der Runde:

$$H_0^{(i)} \leftarrow a + H_0^{(i-1)}, \quad H_1^{(i)} \leftarrow b + H_1^{(i-1)}, \quad H_2^{(i)} \leftarrow c + H_2^{(i-1)},$$
$$H_3^{(i)} \leftarrow d + H_3^{(i-1)}, \quad H_4^{(i)} \leftarrow e + H_4^{(i-1)}, \quad H_5^{(i)} \leftarrow f + H_5^{(i-1)},$$
$$H_6^{(i)} \leftarrow g + H_6^{(i-1)}, \quad H_7^{(i)} \leftarrow h + H_7^{(i-1)},$$

Ergebnis des Algorithmus sind die aneinandergesetzten Hashwerte der letzten Runde.

§4: Weitere Anwendungen sicherer Hashfunktionen

Die Nützlichkeit kryptographisch sicherer Hashfunktionen beschränkt sich nicht auf elektronische Unterschriften; sie sind auch Teil einer Reihe von weiteren Protokollen zum Schutz der Integrität von Daten, für sogenannte „Zeitstempel“ und einiges mehr. Auch die amerikanische Norm für den DSA sieht für die Schlüsselerzeugung den Einsatz von SHA vor. Hier soll vor allem dieser letztere Fall behandelt werden sowie als erstes der

a) Schutz der Integrität von Daten

Wie bereits zu Beginn der Vorlesung erwähnt, hat ein Gegner viele Angriffsmöglichkeiten; angesichts der geringen Sicherheitsstandards der am häufigsten verwendeten Betriebssysteme wird oft der direkte Angriff auf einen Computer die einfachste Möglichkeit sein. Gegen das Ausnutzen von Sicherheitslücken ist die Kryptographie machtlos; sie kann aber doch helfen, zumindest gewisse Manipulationen zu entdecken.

Schädliche Programme lassen sich am leichtesten dann in einen Computer einschleusen, wenn sie der Besitzer selbst installiert. Das wird er natürlich kaum freiwillig tun, aber wenn wenn man ihm eine manipulierte Variante eines Programms unterschiebt, das er ohnehin installieren möchte, wird es vielleicht versehentlich tun.

Teilweise können schon kleinste Manipulationen ein Einfallstor für künftige Angriffe sein: Wie wir bei der Diskussion von SSL/TLS gesehen haben, sind beispielsweise die elektronischen Unterschriften der wichtigsten Zertifizierungsagenturen im Programmcode der gängigen Browser enthalten. Fügt man dort nur einen einzigen Datensatz mit der elektronischen Unterschrift einer Bogus-Agentur hinzu, wird der Browser künftig ohne Nachfrage beliebige Seiten als sicher bezeichnen, wenn sie von dieser Agentur zertifiziert sind. Falls der manipulierte Browsercode auch noch einige Anweisungen mit „Ersatzadressen“ für populäre Ziele enthält, sind praktisch alle Arten von Angriffen möglich.

Ein Anwender muß daher sicher sein, daß seine Programme von einer sicheren Quelle kommen und nicht manipuliert sind. Falls er seine Programme von dubiosen Adressen herunterlädt (die selbstverständlich allesamt seriöse Namen haben) oder wenn er das in der Vergangenheit getan hat, kann ihm auch die Kryptographie nicht mehr helfen.

Falls er aber seinen Computer samt Software-Erstausstattung von seriösen und sorgfältig arbeitenden Produzenten und Händlern erworben hat (so es das im Konsumentenbereich geben sollte), dann kann er sichere Internetverbindungen zu seriösen Anbietern aufbauen und auch sicher sein, daß er mit dem gewünschten Partner verbunden ist. Jetzt muß er nur noch wissen, daß die Software, die er von dort (oder einer näher gelegenen und/oder billigeren Quelle) herunterlädt identisch ist mir der des Anbieters.

Zu diesem Zweck veröffentlichen viele Anbieter Hashwerte zu ihren Programmen. Falls diese mit einem kryptographisch sicheren Hashverfahren berechnet werden, kann man damit nicht nur überprüfen, ob die Software fehlerfrei übertragen wurde, sondern man kann auch überprüfen, daß sie, selbst wenn sie von einem unbekannten *mirror* kommt, identisch ist mit dem Original.

Auf dieselbe Weise lassen sich auch kritische Dateien auf dem eigenen Rechner schützen: Falls man zu jeder einen Hashwert berechnet und diesen idealerweise noch auch einem externen Speichermedium festhält, kann man jederzeit überprüfen, ob eine Datei verändert wurde.

b) Wie zufällig müssen unsere Schlüssel sein?

Idealerweise sollten unsere Schlüssel stets zufällig gewählt sein; jeder in Frage kommende Schlüssel sollte also genau dieselbe Wahrscheinlichkeit haben. Alles andere gibt einem Gegner zumindest prinzipiell Ansätze zu einer Kryptanalyse, die schneller ist als das Durchprobieren aller Schlüssel.

Nun sind aber in der asymmetrischen Kryptographie ohnehin alle vernünftigen kryptanalytischen Ansätze deutlich schneller als bloßes Probieren – deshalb brauchen wir ja auch so lange Schlüssel. Unter Sicherheitsgesichtspunkten ist völlig ausreichend, daß ein Gegner das

Verfahren nicht mit weniger als etwa 2^{128} Rechenschritten knacken kann; das Bundesamt für Sicherheit in der Informationstechnik ist sogar bereits mit einer Unterschranke von 2^{100} zufrieden.

Deshalb müssen wir bei RSA und DSA, selbst wenn wir Primzahlen der Länge 1024 oder gar 2048 Bit suchen, nicht unbedingt jeder solchen Primzahl dieselbe Chance geben: Es reicht, daß wir einen (echten) Zufallsanteil von mindestens 128 Bit haben; der Rest kann dann durchaus deterministisch daraus abgeleitet sein. Dabei muß freilich sichergestellt sein, daß das deterministische Verfahren keine erkennbare (und damit vielleicht auch ausnutzbare) Struktur in die Schlüssel bringt, und dazu ist eine kryptographisch sichere Hashfunktion ein geeignetes Werkzeug.

c) Erzeugung großer Zufallszahlen aus kleinen

Angenommen, wir haben eine Hashfunktion h, die Werte zwischen Null und $2^m - 1$ liefert, und wir haben als Startwert eine Zufallszahl $0 < x_0 < 2^m - 1$. (In der englischsprachigen Literatur wird ein solcher Startwert als *seed* = Keim bezeichnet.) Was wir wirklich wollen, ist aber eine Zahl y zwischen 2^{L-1} und 2^L für ein L, das deutlich größer ist als m.

Dazu gehen wir ähnlich vor wie beim Counter-Mode eines symmetrischen Kryptoverfahrens (*s.* Kap. 3, §5*e*): Wir schreiben $L - 1 = nm + b$ mit $0 \le b < m$ und berechnen $m + 1$ „Ziffern“ zur Basis 2^m als $v_i = h(x_0 + i)$. Dann setzen wir

$$z = \sum_{i=0}^{n-1} v_i \cdot 2^{mi} + (v_n \bmod 2^b) \cdot 2^{(n+1)m} .$$

Das ist noch kein geeigneter Kandidat, denn da $v_n \bmod 2^b$ kleiner ist als 2^b, ist auch $z < 2^{L-1}$. Genau deshalb aber ist $y = 2^{L-1} + z$ eine Zahl zwischen 2^{L-1} und 2^L, die auf einer Zufallszahl mit n zufälligen Bit beruht.

Auf der Suche nach RSA-Moduln können wir dann beispielsweise von dort aus das Sieb des ERATOSTHENES laufen lassen und nach der nächsten Primzahl suchen.

Falls wir extrem vorsichtig sind und dem Gegner keinen Ansatz geben möchten, die sehr inhomogene Verteilung der Differenzen aufeinanderfolgender Primzahlen auszunutzen, werden wir allerdings nur y auf Primalität testen und bei negativem Ausgang des Tests von vorne anfangen.

Ein ERATOSTHENES-Schritt ist jedoch selbst unter diesen Bedingungen völlig problemlos: Da es in dem Größenbereich, in dem wir suchen, keine geraden Primzahlen gibt, sollten wir im Fall eines geraden y den Wert von $y + 1$ testen. Auf diese Weise halbieren wir die Anzahl der zu erwartenden Durchläufe und haben selbst theoretisch keinen Sicherheitsverlust.

d) Primzahlen für DSA

In manchen Fällen brauchen wir nicht einfach *irgendwelche* Primzahlen, sondern zwei Primzahlen, die in einer bestimmten Relation zueinander stehen. Als Beispiel dazu betrachten wir den aus Kapitel 5 bekannten digitalen Unterschriftsalgorithmus DSA, in den USA standardisiert als *Digital Signature Standard* DSS

In seiner derzeit gültigen Form wurde er am 27. Januar 2000 als FIPS 186-2 veröffentlicht. Er benötigt bekanntlich eine kleine Primzahl q und eine große Primzahl $p \equiv 2q \bmod 1$, wobei FIPS 186-2 die Größenordnungen $2^{159} < q < 2^{160}$ und $2^{L-1} < p < 2^L$ mit $512 \le L \le 1024$ vorsieht; ein Nachtrag vom 5. Oktober 2001 läßt nur noch den Wert $L = 1024$ zu, und hier in Deutschland sehen die Empfehlungen der Bundesnetzagentur ohnehin höhere Werte auch für q vor. Im Jahr 2000 gab es allerdings nur den Algorithmus SHA-1 und der war auf die Unterschriftenlänge 160 des DSS abgestimmt. Heute wird man wohl eher mit den neueren SHA-Algorithmen arbeiten und auch längere Unterschriften bevorzugen. Da der Leser keine Schwierigkeiten haben sollte, im folgenden Algorithmus die Zahl 160 überall durch einen anderen Wert zu ersetzen, ist hier aber trotzdem der Originalalgorithmus aus FIPS 186-2 angegeben. Wenn man mit einer SHA-Variante arbeitet, die längere Werte als die gewünschte Unterschrift liefert, muß man sich gegebenenfalls bei der Anwendung von SHA auf die ersten Bits beschränken.

In *Appendix 2* des Standards wird folgendes Verfahren zur Produktion dieser Primzahlen vorgeschlagen: Sei

$$L - 1 = 160n + b \quad \text{mit} \quad 0 \leq b < 160\,.$$

1. Wähle eine zufällige 160-Bit-Zahl x_0.

2. Berechne u = SHA-1(x_0) $\oplus$ SHA-1($x_0 + 1 \bmod 2^{160}$).

u ist eine Folge von 160 Bit; falls allerdings das führende Bit gleich Null ist, beschreibt sie eine Zahl, die kleiner ist als 2^{159}, und wenn das letzte Bit gleich Null ist, haben wir eine gerade Zahl die ebenfalls nicht als Wert für q in Frage kommt. Deshalb:

3. Ersetze in u das führende wie auch das letzte Bit durch Einsen.

4. Teste, ob q eine Primzahl ist.

5. Falls nicht: Zurück zum ersten Schritt.

6. Setze *Zähler* = 0 und *offset* = 2.

7. Berechne v_k = SHA-1(x_0 + *offset* + $k \bmod 2^{160}$) für $k = 0, \ldots, n$.

8. Setze $w = v_0 + v_1 \cdot 2^{160} + \ldots + v_{n-1} \cdot 2^{160(n-1)} + (v_n \bmod 2^b) \cdot 2^{160n}$. Diese Zahl ist kleiner als 2^{L-1}; daher liegt $x0w + 2^{L-1}$ zwischen 2^{L-1} und 2^L.

x soll natürlich ein Kandidat für p sein; allerdings wird x im allgemeinen nicht kongruent eins modulo $2q$ sein. Dies korrigieren wir im nächsten Schritt:

9. Sei $c = x \bmod 2q$ und $p = x - (c - 1)$

10. Falls $p < 2^{L-1}$: Weiter bei Schritt 13.

11. Teste, ob p prim ist.

12. Falls ja, weiter bei Schritt 15

13. Setze *Zähler* = *Zähler* + 1 und *offset* = *offset* + n + 1.

14. Falls *Zähler* $\geq 2^{12} = 4096$, fängt alles wieder bei Schritt eins von vorne an, andernfalls geht es mit den neuen Werten zurück zu Schritt sieben.

15. Wenn der Algorithmus hier ankommt, haben wir geeignete Werte für q und p gefunden. Um beweisen zu können, daß alles mit rechten Dingen zuging, verlangt der Algorithmus von uns, daß wir uns den aktuellen Wert von x_0 merken.

d) Wie bekommt man echte Zufallszahlen?

Durch Hashverfahren können wir zwar die Anzahl benötigter Zufallsbits zumindest bei asymmetrischen Kryptoverfahren deutlich reduzieren, wir können aber definitiv nicht ohne sie auskommen. Alles beruht darauf, daß wir mit mindestens 128 „wirklich“ zufälligen Bits starten. Wie bekommen wir diese? Und was bedeutet das Wort „zufällig“?

Im Alltagsgebrauch bezeichnen wir ein Ereignis als zufällig, wenn es keine Erklärung gibt, warum ausgerechnet dieses Ereignis an Stelle von mehreren ebenfalls möglichen Alternativen eingetreten ist: Fällt beim Würfel die Zahl „drei“, so ist das Zufall.

Untersuchen wir allerdings die Bewegung des Würfels genauer, so können wir ohne größere Probleme aus den Anfangsbedingungen (Ort, Geschwindigkeit, Drehimpuls des Würfels beim Abwurf) zumindest im Prinzip dessen Bahn berechnen und das Ergebnis vorhersagen; es ist also nicht mehr zufällig. In der Tat gibt es Spieler, die in der Lage sind, mit recht guter Trefferwahrscheinlichkeit jede gewünschte Zahl zu würfeln. Bei genauerer Betrachtung des Vorgangs wird die Zufälligkeit des Ergebnisses hier also doch eher problematisch.

In der Tat ist es algorithmisch unmöglich, zu entscheiden, ob eine gegebene Zahlenfolge zufällig ist; wir können nie ausschließen, daß wir einfach das korrekte Bildungsgesetz noch nicht gefunden haben.

Auch bei physikalischen Phänomenen, die uns zufällig erscheinen, müssen wir immer die Möglichkeit im Auge behalten, daß wir vielleicht einfach noch nicht die korrekte Theorie zu ihrer Erklärung gefunden haben. Zumindest in der Quantentheorie gibt es allerdings durchaus Sätze, wonach das Verhalten gewisser Systeme *nicht* mit bislang noch unbekannten „verborgenen Parametern“ deterministisch erklärt werden kann, und Phänomene wie der radioaktive Zerfall oder thermisches Rauschen liefern Werte, die als zufällig interpretiert werden können. Sie sind zwar

im allgemeinen nicht gleichverteilt, aber dieses Problem läßt sich durch geeignete statistische Aufbereitung beheben.

Dem Normalanwender stehen physikalische Quellen üblicherweise nicht zur Verfügung. Trotzdem gibt es auch auf seinem Computer eine ganze Reihe von nicht vorhersehbaren Ereignissen, die keineswegs alle auf Softwarefehlern beruhen: Mißt man etwa den Abstand zwischen zwei Tastaturinterrupts mit einer Genauigkeit von einer Tausendstel Sekunde, so verhält sich die letzte Ziffer sicherlich zufällig: Niemand kann seine Bewegungen mit einer Genauigkeit in diesem Bereich steuern. Auch aus Mausbewegungen, Festplattenzugriffen und aus Netzwerkaktivitäten lassen sich entsprechende Zufallszahlen gewinnen.

Linux (und verschiedenen andere UNIX-Systeme) sammeln die so gewonnene Entropie und stellen sie in /dev/random als Folge gleichverteilter Zufallszahlen zur Verfügung; mit

```
dd if=/dev/random of=Dateiname bs=1 count=n
```

lassen sich n Zufallsbytes gewinnen – sofern hinreichend viel Entropie zur Verfügung steht. Andernfalls blockiert /dev/random und liefert erst dann wieder neue Bytes, wenn neue zufällige Ereignisse eingetreten sind.

Eine zweite Spezialdatei, /dev/urandom, greift ebenfalls auf den Entropiepool des Betriebssystems zu, liefert aber bei nicht ausreichender Entropie algorithmisch berechnete Pseudozufallsbytes anstatt zu blockieren.

Unter cygwin stehen /dev/random und /dev/urandom im Prinzip auch für Windows-Anwender zur Verfügung, allerdings liefern dann beide nur Pseudozufallszahlen in üblicher Windowsqualität, die auf keinen Fall für kryptographische Zwecke verwendet werden dürfen.

Wer keinen Zugang zu physikalischen Zufallsquellen oder Computern mit echten Entropiequellen hat, muß leider seine Zufallsbit auf konventionelle Weise erzeugen, d.h. er muß Münzen oder Würfel werfen und versuchen, dies möglichst „zufällig“ zu tun. Mit einiger Übung ist es nicht sehr schwer, Münzen oder Würfel so zu werfen, daß das Ergebnis

ziemlich vorhersehbar ist; man sollte für „echten“ Zufall auf jeden Fall möglichst hoch werfen und auch da die Kraft variieren.

§5: Literatur

SHA-1 ist bei BUCHMANN und in weiteren neueren Lehrbüchern der Kryptologie behandelt; die übrigen SHA-Algorithmen sind dort allerdings noch nicht zu finden. Alle sind aber ausführlich beschrieben in den Originaldokumenten, deren neueste Version man via

```
http://csrc.nist.gov/CryptoToolkit/tkhash.html
```

und den Link zu *Secure hashing* finden sollte. Entsprechend führt der Link *Digital Signatures* zum DSS.

Grundsätzliches über kryptographische Hashfunktionen und Angriffe dagegen findet man im fünften Kapitel des Buchs

NIELS FERGUSON, BRUCE SCHNEIER, TADAYOSHI KOHNO: *Cryptography Engineering – Design Principles and Practical Applications*, Wiley,2010