

Ausarbeitung zur VU Kryptographie bei Prof. Uwe Egly

Grundlagen kryptographischer Hashfunktionen

Ausarbeitung von:

Philipp Bergh (0226843)
Sebastian Skritek (0226286)

SS 2005

Zusammenfassung. Hashfunktionen sind heutzutage in der Informatik weit verbreitet. Neben ihrem Einsatz bei Überprüfungen der Datenintegrität werden sie vor allem in kryptographischen Anwendungen, wie z.B. der digitalen Signatur, eingesetzt. An diese kryptographischen Hashfunktionen werden besondere Sicherheitsanforderungen gestellt. Dieser Text stellt einen Überblick über die Grundlagen der kryptographischen Hashfunktionen dar. Er geht dabei auf die Definitionen, Eigenschaften und Unterteilungsmöglichkeiten, sowie die Konstruktion und Sicherheit kryptographischer Hashfunktionen ein. Dabei wird vor allem die Konstruktion einer Hashfunktion aus einer Kompressionsfunktion behandelt. Als Beispiel einer kryptographischen Hashfunktion wird der SHA-1 Algorithmus gezeigt.

1 Einleitung

Hashfunktionen sind im Bereich der Informatik heute ein sehr wichtiges und oft eingesetztes Werkzeug.

Ihr Einsatzgebiet erstreckt sich von der Implementierung effizienter Datenstrukturen in Form von Hashtabellen, über die Fehlererkennung in jeglicher Art von elektronischen Dokumenten bis hin zum Einsatz in der digitalen Unterschrift.

Hashfunktionen ermöglichen nicht nur das einfache Auffinden von Übertragungsfehlern und helfen bei deren Korrektur, sondern dienen auch dazu, böswillige Änderungen an Dateien einfach feststellen zu können.

Zur Speicherung von Passwörtern werden, falls diese nicht „plaintext“ gespeichert werden, ebenfalls Hashfunktionen eingesetzt.

Außerdem sind sie geeignet „Wissen“, welches in Form elektronischer Dokumente vorliegt, zu vergleichen, ohne dass etwas davon preisgegeben werden müsste.

Hashfunktionen existieren in vielen Ausprägungen mit unterschiedlichen Eigenschaften. Ihre Komplexität reicht, je nach Anwendung, von einfachen Quersummen bis zu komplizierten mathematischen Funktionen und iterativ ausgeführten Permutations- und Kompressionsalgorithmen.

Einige der oben genannten Anwendungsbereiche sind dem Gebiet der Kryptographie zuzuordnen. Hashfunktionen, welche im kryptographischen Kontext eingesetzt werden, nennt man deshalb auch kryptographische Hashfunktionen. Sie gehören zu den komplexesten Hashfunktionen, da mitunter hohe Anforderungen an sie gestellt werden. Kryptographischen Hashfunktionen kommt aber auch, wenn auch nur indirekt, in der Öffentlichkeit große Aufmerksamkeit zu. Durch die Entwicklung, Einführung und immer größer werdende Akzeptanz der digitalen Unterschrift kommt den zu Grunde liegenden Algorithmen, und damit auch den kryptographischen Hashfunktionen, große Bedeutung zu. Doch gerade die Sicherheit der verwendeten Hashfunktionen wird immer wieder diskutiert, und nachdem MD4 schon länger als unsicher gilt, gibt es jetzt auch Meldungen dass sowohl MD5 als auch SHA-1, die beiden weitest verbreiteten kryptographischen Hashfunktionen, gebrochen wurden.

Wir versuchen eine allgemeine Einführung in dieses Thema zu geben.

Dabei beginnen wir mit Begriffsdefinitionen und -klärungen. Hierbei handelt es sich im Großen und Ganzen um Eigenschaften, an Hand derer wir im nächsten Abschnitt (kryptographische) Hashfunktionen klassifizieren wollen.

Anschließend folgt ein Überblick über die Konstruktionsmöglichkeiten parameterloser, kryptographischer Hashfunktionen. Wir gehen dabei genauer auf den zentralen Punkt der Überführung einer Kompressionsfunktion in eine Hashfunktion an Hand des Algorithmus von Damgård ein. Danach folgt ein kurzer Überblick über die Konstruktionsmöglichkeiten von Kompressionsfunktionen. Dabei gehen wir genauer auf den Algorithmus SHA-1, einen der zurzeit meistverwendeten Algorithmen, ein. Im Anschluss daran folgt eine kurze Übersicht über die Konstruktionsmöglichkeiten

parametrisierter Hashfunktionen. Zum Schluss betrachten wir allgemeine Gefahren für und Angriffe auf Hashfunktionen, ohne dabei speziell auf konkrete Angriffe auf spezielle Hashfunktionen einzugehen.

Wir benützen im Weiteren wie bisher den eingedeutschten Begriff der Hashfunktion anstelle der deutschen Begriffe der Streuwertfunktion.

2 Begriffe und Eigenschaften

Bei der Beschreibung und Diskussion von Hashfunktionen kann man zwischen bestimmten Eigenschaften der Funktionen unterscheiden. Vor einer genaueren Definition und Einteilung der Hashfunktionen geben wir einen kurzen Überblick über einige dieser Eigenschaften.

Hashfunktionen sind Funktionen, welche auf Strings operieren, d.h. sie bilden einen Eingabestring auf einem eindeutigen Ausgabestring ab.

Sei Σ ein Alphabet (d.h. eine Menge von Zeichen), dann bezeichnet Σ^n die Menge aller Zeichenfolgen der Länge n , bestehend aus Zeichen des Alphabets Σ . (Eine Zeichenfolge der Länge n wird auch String der Länge n genannt) Mit Σ^* werden alle Strings beliebiger Länge über dem Alphabet Σ bezeichnet (d.h. alle möglichen Zeichenfolgen bestehend aus Zeichen aus Σ).

Eine der grundlegenden Eigenschaften ist die Effizienz der Berechenbarkeit einer Funktion. Man unterscheidet zwischen leicht und schwer berechenbaren Funktionen. Eine exakte, formale Definition dieser Begriffe ist schwierig, weswegen dies in der Literatur oft unterlassen wird. [1] und [3] geben z.B. nur intuitive, informelle Definitionen, womit auch wir uns begnügen werden. Schwer (oder auch „praktisch unmöglich“) zu berechnen bedeutet, dass ein Programm zur Berechnung der Funktion (bei genügend großer) Eingabe entweder kein Ergebnis liefert oder zumindest nicht innerhalb eines vertretbaren Zeitrahmens. Eine Funktion ist z.B. dann schwer berechenbar, wenn sie exponentielle Laufzeit besitzt. Leicht berechenbar ist eine Funktion hingegen dann, wenn das Ergebnis hinreichend schnell erzeugt werden kann. (Wobei „hinreichend schnell“ nicht genau definiert werden kann und von der Anwendung abhängt, im Alltag wahrscheinlich max. innerhalb weniger Sekunden.)

Weitere Eigenschaften von Funktionen sind ihre Definitions- und Wertemengen (also die Ein- und Ausgabemengen). Hierbei ist für die weitere Betrachtung vor allem die Unterscheidung zwischen Funktionen, welche Strings beliebiger Länge auf Strings fixer Länge abbilden, und Funktionen, welche Strings von einer fixen Länge m auf Strings anderer fixer Länge n ($m > n$) abbilden, wichtig.

Formal beschrieben bedeutet der erste Fall eine Funktion h der Form $h: \Sigma^* \rightarrow \Sigma^n$ und der zweite Fall eine Funktion der Form $h: \Sigma^m \rightarrow \Sigma^n$ ($m > n$).

Existieren zwei Werte, x und x' ($x \neq x'$), deren Funktionswerte (auch „Hashwerte“) $h(x) = y$, $h(x') = y'$ gleich sind ($y = y'$), so spricht man von einer Kollision.

Es ist klar dass, sobald eine Funktion einen Definitionsbereich auf einen echt kleineren Wertebereich abbildet (wie bei $h: \Sigma^* \rightarrow \Sigma^n$ bzw. $h: \Sigma^m \rightarrow \Sigma^n$ ($m > n$) der Fall), solche Kollisionen existieren müssen.

Formal ist eine Kollision das Paar (x, x') .

Eine Funktion ist schwach kollisionsresistent („2nd – preimage resistance“), wenn es (im oben definierten Sinn) schwer ist, zu einem gegebenen x ein x' zu finden, für welches gilt, dass $h(x) = h(x')$ (und $x \neq x'$)

Stark kollisionsresistent („collision resistance“) wird eine Funktion genannt, wenn es schwer ist, irgendwelche zwei Eingaben x und x' ($x \neq x'$) zu finden, so dass gilt $h(x) = h(x')$.

Der Unterschied zur schwachen Kollisionsresistenz besteht also darin, dass beide Werte frei gewählt werden dürfen, während bei der schwachen Kollisionsresistenz ein Wert vorgegeben ist.

Daraus folgt, dass die Eigenschaft der starken Kollisionsresistenz wesentlich schwieriger zu erreichen ist als die der schwachen Kollisionsresistenz.

Eine noch strengere Bedingung stellt die Beinah – Kollisionsresistenz („near-collision resistance“) dar. Um eine Beinah – Kollision handelt es sich, wenn sie die Funktionswerte von zwei Eingaben x und x' in nur wenigen Stellen unterscheiden. (Bei Darstellung des Hashwertes als Bitfolge z.B. nur an einigen Bits)

Eine weitere Eigenschaft ist die Umkehrbarkeit (Invertierbarkeit) einer Funktion h , d.h. die Möglichkeit zu einem gegebenen (Ausgabe-) Wert y einen (Eingabe-) Wert x zu finden, so dass gilt $h(x) = y$. Im Weiteren wird uns eher die Nicht – Umkehrbarkeit („preimage resistance“) interessieren, d.h. dass es schwer oder unmöglich ist, zu einem y ein x mit $h(x) = y$ zu finden.

Ein weiteres Merkmal von Hashfunktionen ist, ob die Funktion h mit einem Schlüssel k ($k \in K$) parametrisiert ist („keyed hash function“), oder nicht („unkeyed hash function“). K bezeichnet dabei den Schlüsselraum von h . Eine parametrisierte Hashfunktion ist eine Familie $\{h_k: k \in K\}$.

Für parametrisierte Hashfunktionen existiert eine weitere Eigenschaft, die Fälschungsresistenz. D.h. dass es nicht möglich sein darf ohne Kenntnis des Schlüssels k den Hashwert $h_k(x)$ zu einer Eingabe x auszurechnen (also das Paar $(x, h_k(x))$ zu erzeugen), selbst wenn Paare $(x_i, h_k(x_i))$ bekannt sind.

Außerdem wird von Hashfunktionen meist verlangt, dass selbst kleine Unterschiede im Eingabewert zu großen Unterschieden in den daraus resultierenden Hashwerten führen.

3 Klassen von Hashfunktionen

Hashfunktionen lassen sich nun danach einteilen, welche der zuvor genannten Eigenschaften sie erfüllen, und welche nicht. (Die Bezeichnung der Hashfunktionen in der Literatur ist nicht einheitlich, hinzu kommen noch Unstimmigkeiten in manchen Übersetzungen. Wir versuchen mit so vielen Quellen wie möglich konform zu gehen, trotzdem kann es sein, dass die Bezeichnungen divergieren)

3.1. Definitionen

Eine Hashfunktion h muss folgende zwei Bedingungen erfüllen:

Für ein gegebenes x muss der Wert $h(x)$, also der Hashwert, einfach zu berechnen sein. h muss Strings beliebiger Länge auf Strings fester Länge abbilden. (also von der Form $h: \Sigma^* \rightarrow \Sigma^n$ sein)

Wann immer wir im Folgenden von einer Hashfunktion sprechen, werden diese Eigenschaften vorausgesetzt, und weitere genannte Eigenschaften sind jeweils zu diesen hinzuzufügen.

Eine Einwegfunktion muss die Eigenschaft der Nicht – Umkehrbarkeit besitzen. Zur Zeit ist es ungewiss, ob solche Funktionen überhaupt existieren. Es lässt sich zeigen, dass Einwegfunktionen dann existieren, wenn $P \neq NP$. Es gibt allerdings Funktionen welche dem heutigen Wissensstand nach Einwegfunktionen sind, d.h. Funktionen für deren Umkehrung kein effizienter Algorithmus bekannt ist. Es kann jedoch sein, dass Algorithmen existieren, um die Umkehrung effizient zu berechnen.

Eine (schwach/stark) kollisionsresistente Hashfunktion muss die Eigenschaft der (schwachen/starken) Kollisionsresistenz besitzen. Stark kollisionsresistente Hashfunktionen werden auch kollisionsfreie Hashfunktionen genannt.

Unter einer schwachen Einweg – Hashfunktion (auch „weak one – way hash function“) versteht man eine schwach kollisionsresistente Hashfunktion, welche zusätzlich eine Einwegfunktion ist.

Eine Einweg – Hashfunktion („strong one – way hash function“) ist eine stark kollisionsresistente Hashfunktion, welche zusätzlich eine Einwegfunktion ist. ([3] verlangt die Einweg – Eigenschaft nicht, sie wird jedoch u.a. in [4] und [2] gefordert, weiters wird in [1] behauptet dass man zeigen kann, dass kollisionsresistente Hashfunktionen Einwegfunktionen sein müssen).

Unter dem Begriff der „kryptographischen Hashfunktion“ versteht man nun meist die Einweg- Hashfunktion, wobei abhängig von der Verwendung auch noch zusätzliche Eigenschaften (wie z.B. die Beinahe – Kollisionsresistenz) gefordert sein können.

3.2. Beispielanwendungen

Welche Art von Hashfunktion benötigt wird, ist abhängig von ihrem Einsatzgebiet. Bei Hashtabellen werden Hashfunktionen beispielsweise benutzt, um eine String – Repräsentation eines Objekts (z.B. dessen Namen) auf eine Speicherstelle in der Hashtabelle abzubilden. Kollisionen sind in dem Zusammenhang zwar nicht wünschenswert, jedoch existieren Methoden um auftretende Kollisionen zu behandeln, so dass, falls die Anzahl der Kollisionen nicht übermäßig groß ist, die Funktionalität einer Hashtabelle nicht gravierend beeinträchtigt wird. Auch wäre es kein Problem, wenn man von dem Hashwert auf den ursprünglichen Wert schließen könnte.

Will man mittels einer Hashfunktion die Integrität eines Dokuments (wobei ein Dokument in dieser Hinsicht jede beliebige Bitfolge sein kann) überprüfen, kann man folgendermaßen vorgehen:

Man errechnet zum Zeitpunkt t_1 den Hashwert des Dokumentes und speichert diesen (sicher vor Veränderungen, z.B. auf einer Chipkarte). Will man zu einem späteren Zeitpunkt t_2 wissen, ob das Dokument verändert wurde, errechnet man den Hashwert von neuem und vergleicht ihn mit dem gespeicherten Wert. Stimmen die Werte überein, so geht man davon aus, dass das Dokument nicht verändert wurde. Stimmen die Werte nicht überein, wurde das Dokument verändert.

Um diese Annahme treffen zu können, muss es sich bei der verwendeten Hashfunktion um eine schwache Einweg – Hashfunktion handeln, d.h. es muss schwer sein, zum gegebenen Dokument ein zweites Dokument zu finden, welches denselben Hashwert besitzt.

Werden Passwörter nur in Form ihres Hashwertes gespeichert, so ist es vor allem wichtig, dass man von dem gespeicherten Hashwert nicht auf das Passwort schließen kann. ([3] nennt dies als einzige Bedingung, da jedoch meist nur der Hashwert des eingegebenen Passworts mit dem gespeicherten Wert verglichen wird, wäre zumindest eine schwach kollisionsresistente Einweg - Hashfunktion wünschenswert)

Im Falle digitaler Unterschriften werden schlussendlich stark kollisionsresistente Einweg – Hashfunktionen benötigt. Angenommen jemand (A) hat einen Vertrag aufgesetzt und möchte diesen von B unterschreiben lassen. Er verschlüsselt dazu den Vertrag und schickt ihn zu B. B entschlüsselt den Vertrag, berechnet den Hashwert des Vertrages und signiert anschließend diesen Hashwert (z.B. mit ihrem privaten Schlüssel). B schickt nun die Unterschrift (also den verschlüsselten Hashwert) wieder zurück an A. A soll jetzt nicht in der Lage sein den Vertrag durch einen anderen Vertrag, der den selben Hashwert besitzt, zu ersetzen, da B's Unterschrift ja sonst auch für den neuen Vertrag gelten würde. Da A die beiden Dokumente auswählen kann, ist starke Kollisionsresistenz nötig.

3.3. Funktionelle Gliederung – MDCs und MACs

Hashfunktionen, welche für digitale Signaturen oder zur Überprüfung der Integrität von Daten eingesetzt werden, lassen sich unter dem Begriff der MDC („modification detection codes“) zusammenfassen. Sie dienen, wie der Name schon sagt, hauptsächlich dazu, Änderungen an Daten zu erkennen. (Es handelt sich hierbei um eine funktionelle Gliederung nach [3]).

MDCs stellen dabei eine Untergruppe der parameterlosen Hashfunktionen dar. Parameterlose Hashfunktionen werden auch für andere Zwecke eingesetzt, so z.B. um zu bestätigen, dass man gewisse Daten besitzt, ohne dass man die Daten veröffentlichen müsste. Sie können ebenfalls eingesetzt werden um (neue) Nebenschlüssel von einem Hauptschlüssel abzuleiten (z.B. Rundenschlüssel), oder zur Erzeugung von Pseudo – Zufallszahlen.

Eine andere Gruppe stellen die bereits erwähnten parametrisierten Hashfunktionen dar. Eine Untergruppe der parametrisierten Hashfunktionen sind die „Message Authentication Codes“ (MAC). ([1] benützt MAC als synonym für parametrisierte Hashfunktionen, während [3] auf Grund der eher funktionellen Definition noch andere Untergruppen von parametrisierten Hashfunktionen zulässt, so z.B. Funktionen in Challenge – Response Identifikationsprotokollen.) MAC sind (parametrisierte) Hashfunktionen, welche zusätzlich die Bedingung der Fälschungsresistenz erfüllen.

Die Idee bei MAC ist wie folgt:

Will A ein Dokument an B schicken, so dass B sicher gehen kann, dass das Dokument auch wirklich von A stammt, so könnte A von dem zu übertragenden Dokument mittels eines MAC (und eines geheimen Schlüssels k , der außer A und B niemanden bekannt sein sollte) einen Hashwert anfertigen und diesen mit dem Dokument übertragen. B errechnet wiederum den Hashwert des Dokuments (mittels eines MAC und k), und vergleicht die beiden Hashwerte. Stimmen sie überein, kann B auf Grund der Fälschungsresistenz davon ausgehen, dass auch wirklich A der Sender war.

[3] weist extra darauf hin, dass sich dieses Vorgehen von dem mittels eines MDC und der Verschlüsselung des damit erstellten Hashwertes mit einem privaten Schlüssel unterscheidet.

4 Konstruktion von Hashfunktionen:

Wir gehen bei der Konstruktion von Hashfunktionen hauptsächlich auf die Konstruktion der populäreren MDC's sein, und geben anschließend nur einen kurzen Überblick über die Möglichkeiten bei der Konstruktion von MACs,

4.1. Allgemeiner Aufbau:

Ein Problem bei der Konstruktion von Hashfunktionen ist, dass sie auf beliebigen Eingabegrößen operieren sollen, und dabei ihre Eigenschaften (Kollisionsresistenz) nicht verlieren dürfen.

Merkle ([5],[6]) und Damgård ([7]) haben unabhängig voneinander eine Methode entwickelt, welche es ermöglicht, Strings beliebiger Länge durch mehrmalige Ausführung einer Kompressionsfunktion auf Teilen des Strings auf einen fixen Ausgabewert abzubilden. Diese iterative Vorgehensweise ist heute die Grundarchitektur beinahe aller Hashfunktionen.

Die Hashfunktion übernimmt bei dieser Vorgehensweise die Eigenschaften der Kompressionsfunktion bzgl. Kollisionsresistenz und Einweg – Verhalten.

Die beiden vorgeschlagenen Methoden sind sich sehr ähnlich. Damgård schreibt in der Einleitung von [7]: „*Our construction is very similar to Merkle's „meta-method“, discovered independently, in comparison, the present construction contains several extra elements that make a formal proof possible without any extra assumptions.*”

Wir geben im Folgenden einen Überblick über die von Damgård vorgestellte Vorgehensweise, und gehen im Anschluss kurz auf einige Unterschiede zu Merkle's Meta-Methode ein.

4.2. Erstellen einer Hashfunktion aus einer Kompressionsfunktion:

Sei f eine kollisionsresistente Kompressionsfunktion der Form $f: \{0,1\}^m \rightarrow \{0,1\}^t$, und h die daraus entstehende Hashfunktion $h: \{0,1\}^* \rightarrow \{0,1\}^t$.

Man muss zwischen den beiden Fällen $m-t > 1$ und $m-t = 1$ unterscheiden, wobei wir zuerst den allgemeineren Fall $m-t > 1$ betrachten. Sei x der Eingabestring für h der Form $\{0,1\}^*$.

Der Algorithmus dazu funktioniert wie folgt:

1. Teile x in Blöcke der Länge $u = m - t - 1$ auf.
2. Ist der letzte Block kürzer als u , so hänge so viele 0 an sein Ende bis er die nötige Länge hat. (d bezeichnet die Anzahl der benötigten 0, n die Länge von $x + d$)

Die Blöcke werden mit $x_1, x_2, \dots, x_{n/(m-t-1)}$ bezeichnet

3. An diese Blöcke wird ein Block $x_{n/(m-t-1)+1}$ angehängt, welcher d (in binärer Form) beinhaltet. Ist die binäre Darstellung von d kürzer als die Blockgröße, so wird der Block links von d mit 0 angefüllt.

4. Berechne aus den Blöcken $x_1, x_2, \dots, x_{n/(m-t)+1}$ Blöcke der Länge t ($h_1, h_2, \dots, h_{n/(m-t)+1}$).
Setze dazu: $h_1 = f(0^{t+1} || x_1)$, sowie $h_{i+1} = f(h_i || 1 || x_{i+1})$.

5. $h(x) = h_{n/(m-t)+1}$

Dabei sei 0^{t+1} ein String der Länge $t+1$ welcher nur aus 0 besteht, und $a || b$ die Aneinanderkettung von a und b ($a = 1010, b = 111 \Rightarrow a || b = 1010111$).

In den Schritten 1 – 3 wird der Eingabestring so vorbereitet, dass er später (in Schritt 4) als Eingabe für f genutzt werden kann. Die Anzahl d der an x angefügten 0 wird deswegen ebenfalls mitkodiert, damit eine Unterscheidung zwischen solchen Nachrichten herrscht, an die d 0 angehängt wurden, und solchen die einfach mit d 0 enden. (Ohne diesem Schritt hätten z.B. die Nachrichten 101 und 10100 denselben Hashwert, wenn man von einer Blocklänge von 5 bit ausgeht.)

Im Schritt 4 wird so lange f auf dem Ergebnis der letzten Runde und einem aus x entstandenen Block ausgeführt, bis alle in den Schritten 1 – 3 entstandenen Blöcke abgearbeitet werden. Das Ergebnis der letzten Runde wird vom neuen Eingabeblock jeweils durch eine 1 getrennt. Da in der ersten Runde noch kein Ergebnis aus einer vorhergehenden Runde vorliegt, werden vor den Block x_1 $t+1$ 0 gesetzt. Davon ersetzen/simulieren t der 0 das Ergebnis der vorhergehenden Runde, und statt dem trennenden 1 wird in der ersten Runde ebenfalls eine 0 benützt.

Die Hashfunktion h besteht also aus einer Initialisierung und dem iterativen Ausführen der Kompressionsfunktion. Das Ergebnis der Hashfunktion entspricht dem Ergebnis der letzten Ausführung der Kompressionsfunktion.

Für den Fall dass $m - t = 1$ ist klar, dass in jeder Iteration nur ein Zeichen des Eingabestrings verarbeitet werden kann, und dass die Trennung des Ergebnisses der letzten Runde von der neuen Eingabe mittels einer 1 nicht mehr durchführbar ist (da ja sonst kein Platz für die neue Eingabe bliebe).

Falls das Image von f (also alle möglichen von f erzeugten Ausgaben) gleichverteilt ist, so kann Schritt 4 mittels eines zufällig gewählter Bit-String der Länge t , y , wie folgt modifiziert werden: $h_1 = f(y || x_1)$, und $h_{i+1} = f(h_i || x_{i+1})$. (Diese Methode kann immer angewendet werden, wenn die Wahrscheinlichkeitsverteilung von f annähernd eine Gleichverteilung ist)

Erfüllt die Wahrscheinlichkeitsverteilung von f diese Eigenschaft nicht, so bleibt für den Fall $m - t = 1$ nur die ineffiziente Lösung, Schritt 4 wie folgt zu modifizieren: $h_1 = f(0^t || x_1)$, sowie $h_{i+1} = f(h_i || x_{i+1})$, wobei alle zu hashenden Nachrichten Präfix – frei kodiert werden müssen.

Die Laufzeit des angegebenen Algorithmus beträgt $O(n/(m - t) + 1)$, wenn man die Ausführung von f als elementare Operation betrachtet. Die einzige Stelle im Algorithmus in welcher die Eingabegröße eine Rolle spielt, ist Schritt 4, da die Länge der Eingabegröße die Anzahl der zu berechnenden h_i bestimmt, deren Anzahl auf $n/(m - t) + 1$ beschränkt ist.

Damgård zeigt in [7] ebenfalls, dass es möglich ist, h auf einer Eingabe mit der Länge n in $O(\log_2(n/t) * t / (m-t))$ Schritten durchzuführen, wenn man die Auswertung von f wiederum als einen Schritt betrachtet, und $n/(2t)$ Prozessoren zur Verfügung hat (m und

t sind wiederum die Eingabe – bzw. Ausgabelängen der zugrunde liegenden Kompressionsfunktion f.).

Generell können durch den Einsatz mehrerer Prozessoren (Anzahl = c) die Ergebnisse für längere Eingaben um ein c – faches schneller berechnet werden als mit dem Einsatz von nur einem Prozessor (Dazu muss die Eingabe in c, etwa gleich lange, Blöcke aufgeteilt werden, welche einzeln gehasht und zum Schluss zusammengefügt werden.).

Mittels eines indirekten Beweises lässt sich zeigen, dass die auf diese Art und Weise konstruierte Hashfunktion h kollisionsresistent ist, unter der Annahme, f sei kollisionsresistent. Dabei wird versucht, eine Kollision von h auf eine Kollision in f zurückzuführen. Da f aber eine kollisionsresistente Kompressionsfunktion ist, ist eine Kollision für f schwer zu finden, und damit auch für h. (Wir folgen hierbei dem von Damgård in [7] erbrachten Beweis.)

Angenommen, es gäbe einen Algorithmus g, mit welchem sich ein Paar (x, x') , $x \neq x'$, finden lässt, für das gilt: $h(x) = h(x')$ (also eine Kollision), und seien h_i, x_i (bzw. h'_i, x'_i) die jeweiligen Zwischenergebnisse in Schritt 4.

Gilt $\text{länge}(x) \neq \text{länge}(x') \bmod (m-t)$, d.h. dass der letzte Block von x und x' mit einer verschiedenen Anzahl von 0 gefüllt wurde, so unterscheiden sich auch die jeweils letzten Blöcke $(x_{n/(m-t)+1}$ bzw. $x'_{n'/(m-t)+1}$), welche nur diese Zahl darstellen. Da $h(x) = h(x')$ zu $f(h_{n/(m-t)} || 1 || x_{n/(m-t)+1}) = f(h'_{n'/(m-t)} || 1 || x'_{n'/(m-t)+1})$ (Schritt 5) führt, ergibt dies eine Kollision für f (die Eingaben von f sind offensichtlich verschieden).

Gilt $\text{länge}(x) \neq \text{länge}(x') \bmod (m-t)$ nicht (d.h. beide Eingaben werden mit der selben Anzahl an 0 „aufgefüllt“), so folgt aus einer Kollision für h:

$f(h_{n/(m-t)} || x_{n/(m-t)+1}) = f(h'_{n'/(m-t)} || x'_{n'/(m-t)+1})$. Ist $h_{n/(m-t)} \neq h'_{n'/(m-t)}$, so folgt daraus eine Kollision für f. Sind die Werte gleich so betrachtet man, wie sich diese Werte ergeben (Schritt 4), und kommt auf die Gleichung

$$f(h_{n/(m-t)-1} || x_{n/(m-t)}) = f(h'_{n'/(m-t)-1} || x'_{n'/(m-t)})$$

Diese Vorgehensweise lässt sich rekursiv auf h und h' fortsetzen. Da $x \neq x'$ muss zu irgendeinem Zeitpunkt gelten, dass $x_i \neq x'_i$, was zu einer Kollision für f führt. Sind x und x' unterschiedlich lang (und sei o.b.d.A. $\text{länge}(x') \geq \text{länge}(x)$), und gilt für kein i $x_i \neq x'_i$, so erhält man die Gleichung $f(0^{t+1} || x_1) = f(h'_1 || 1 || x'_{i+1})$, was zu einer Kollision für f führt, da h'_1 nur die Länge t besitzt und somit auf der rechten Seite an Stelle t+1 eine 1 steht, während links eine 0 steht.

Somit lässt sich eine Kollision von h immer auf eine Kollision von f zurückführen.

Einer der Unterschiede zwischen dieser Methode und Merkle's Meta – Methode ist, dass bei Merkle nicht die Anzahl der dem letzten Block hinzugefügten 0 ans Ende der Nachricht gehängt wird, sondern die Länge der Originalnachricht (also die Länge von x). Außerdem schlägt Merkle in [6] vor, diese Länge „rechtsbündig“ im letzten Nachrichten - Block hinzuzufügen, und nur falls dort der Platz nicht ausreicht, einen neuen Block mit der Länge anzuhängen. Die Grundidee- und Vorgehensweise ist jedoch in beiden Fällen dieselbe.

4.3. Erzeugen von Kompressionsfunktionen

Die Erzeugung von (sicheren) Hashfunktionen kann mittels dieser Methode also auf das Erzeugen von (sicheren) Kompressionsfunktionen reduziert werden.

Es ist allerdings nicht bekannt, ob kollisionsresistente Kompressionsfunktionen überhaupt existieren. Es wurde bis jetzt weder gezeigt, dass es kollisionsresistente Kompressionsfunktionen gibt, noch konnte ihre Nicht – Existenz bislang bewiesen werden. Man benützt deshalb Kompressionsfunktionen, welche als kollisionsresistent gelten, d.h., für die die Existenz von Kollisionen noch nicht nachgewiesen wurde. (Ähnlich wie die Sicherheit von fast allen Verschlüsselungsfunktionen zurzeit nicht nachgewiesen werden kann, etwa RSA.)

Heutzutage gibt es drei Hauptgruppen von Kompressionsfunktionen, die in Hashfunktionen eingesetzt werden:

- Kompressionsfunktionen, welche auf modularer Arithmetik basieren,
- Kompressionsfunktionen, die auf Verschlüsselungsfunktionen aufbauen, oder
- Kompressionsfunktionen, welche speziell für den Einsatz in Hashfunktionen konstruiert wurden.

4.4. Kompressionsfunktionen basierend auf modularer Arithmetik:

Gründe, Funktionen modulo M als Grundlage für Kompressionsfunktionen zu benützen, sind zum einen die Wiederverwendbarkeit bestimmter Softwarekomponenten (z.B. aus Public–Key Systemen), zum Anderen die Tatsache, dass die Sicherheit über die Wahl der Größe der benützten Zahlen recht gut gesteuert werden kann. Außerdem kann (z.B. bei Sicherheitsbetrachtungen) auf Ergebnisse der Zahlentheorie zurückgegriffen werden. So geht die Sicherheit dieser Kompressionsfunktionen mit Erkenntnissen in der Zahlentheorie einher. Es ist zum Beispiel bekannt, dass bestimmte Kompressionsfunktionen zumindest dann kollisionsresistent sind, wenn das Berechnen diskreter Logarithmen in $(\mathbb{Z}/p\mathbb{Z}^*)$ schwer ist. [1]

Gegenüber speziell für Hashfunktionen erstellten Funktionen sind auf modularer Arithmetik basierende Funktionen allerdings langsamer in der Berechnung ([2] besagt außerdem, dass in der Vergangenheit viele nicht sichere Vorschläge unterbreitet wurden).

Darmgård führt in [7] eine Hashfunktion vor, welche auf quadrieren modulo (bzw. dessen schwerer Rückführbarkeit) einer aus zwei Primzahlen zusammengesetzten Zahl beruht. Die Grundidee ist wie folgt:

Sei $n = p \cdot q$ (p und q seien große Primzahlen), und s die Länge von n in bits. Weiters benötigt man eine geeignete Menge I , eine Untermenge von $\{1, \dots, s\}$ (darauf, wann eine Menge geeignet ist, gehen wir hier nicht ein), sowie eine Funktion $f_i(y)$, wobei $f_i(y)$ jedem String y (bestehend aus den bits y_1, \dots, y_s) den String zuordnet welcher aus den y_i besteht, dessen $i \in I$.

Die Kompressionsfunktion $f: \{0,1\}^m \rightarrow \{0,1\}^t$ ($m = s-8$, $t = |I|$) sieht nun wie folgt aus:
 $f(x) = f_i((3F|x)^2 \bmod n)$,

wobei $3F|x$ bedeutet dass das Byte $3F$ (hex) vor das x gehängt wird (deswegen $m = s-8$). Dies führt dazu, dass das Ergebnis des Quadrats größer als n ist, und dadurch reduziert wird, und verhindert noch weitere „einfache“ Kollisionen.

Eine weitere auf modularer Arithmetik beruhende Hashfunktion stellt z.B. MASH (MASH-1 und MASH-2) dar.

4.5. Kompressionsfunktionen basierend auf Verschlüsselungsfunktionen:

Wenn man von Kompressionsfunktionen spricht, die auf Verschlüsselungsfunktionen aufbauen, dann sind mit „Verschlüsselungsfunktionen“ Blockchiffren (meist im CBC – Modus) gemeint.

Der Grundgedanke ist folgender:

Eine Blockchiffre nimmt zwei Eingabewerte (einen Schlüssel k und den zu verschlüsselnden Block n), und bildet diese Eingaben auf eine Ausgabe mit derselben Größe wie n ab. Kompressionsfunktionen, welche auf Blockchiffren basieren, haben deswegen meist neben der Blockchiffre noch mindestens eine weitere Komponente, welche den Eingabestring (oder Teile davon) auf den Schlüssel der Blockchiffre aufteilt. Blockchiffren werden in diesem Fall meist im CBC – Modus verwendet, so dass alle vorhergehenden Blöcke in den aktuell zu bearbeitenden Block miteinfließen, und der letzte Block als Hashwert verwendet werden kann.

Solche Kompressionsfunktionen haben den Vorteil, dass es für die Blockchiffre bereits (meist effiziente) Implementierungen gibt, welche man zum Hashen sehr einfach verwenden könnte.

Ein weiterer erhoffter Vorteil ist, dass man sich die Zertifizierung einer Hashfunktion, also das Überprüfen auf gewisse Sicherheitseigenschaften, ersparen kann, wenn die Blockchiffre gewisse Eigenschaften besitzt. Merkle nennt dieses Eigenschaft in [5] „pre-certified“.

So stellt z.B. die Tatsache, dass Blockchiffren keine echten Zufallsergebnisse besitzen (da sie ja umkehrbar sein sollen), ein Problem dar.

Es sind zwar einige nötige Eigenschaften bekannt, welche eine Blockchiffre erfüllen muss, damit sie in einer sicheren Hashfunktion eingesetzt werden kann, man weiß allerdings noch nicht, welche Eigenschaften einer Blockchiffre hinreichend sind, um eine sichere Hashfunktion zu erzeugen.

Man unterteilt aus Blockchiffren erzeugte Kompressionsfunktionen, abhängig von der Größe ihres Hashwertes in Vergleich zur der Größe der Blockgröße der verwendeten Blockchiffre, in solche, deren Hashwert genauso groß sind wie die Blockgröße der Chiffren („single-length“), und jene, deren Blockgröße doppelt so groß ist („double-length“).

Der Grund, warum man den Hashwert doppelt so groß macht wie die Blockgröße, ist folgender:

Viele Blockchiffren operieren auf 64 bit Blöcken. Hashfunktionen welche nur 64 bit Hashwerte produzieren, sind allerdings bereits anfällig sind gegen die sog.

Geburtstagsattacke, und somit nicht mehr stark kollisionsresistent. ([1] fordert einen Hashwert von mind. 128 bit um die Geburtstagsattacke abzuwehren, genaueres dazu findet sich im Abschnitt Angriffe)

Single-Length – Hashfunktionen welche z.B. auf DES (oder anderen Blockchiffren mit 64 bit) aufbauen, sind dementsprechend nicht geeignet.

Eine weitere „Kenngröße“ für Blockchiffren basierte Kompressionsfunktionen ist, wie oft die zugrunde liegende Blockchiffre in einem Durchlauf der Kompressionsfunktion aufgerufen wird.

4.6. Speziell für Hashfunktionen entworfene Funktionen:

Neben Kompressionsfunktionen, welche auf modularer Arithmetik oder existierenden Verschlüsselungsfunktionen basieren, gibt es auch solche, welche ausschließlich für den Einsatz in Hashfunktionen konstruiert wurden. Da dabei nicht versucht wurde, bestehende Komponenten wiederzuverwenden, konnte ein besonderes Augenmerk auf die effiziente Verarbeitung beim Hashen gelegt werden.

Beispiele für solche Kompressionsfunktionen sind z.B. MD5, SHA-1, sowie RIRPMED (-128, -160)

5 SHA-1

SHA-1 wurde genau wie sein Vorgänger SHA, oder auch SHA-0 genannt, vom NSA und dem NIST entwickelt. Beim NSA handelt es sich um eine Institution, die die nationale Sicherheit der vereinten Nationen von Amerika gewährleisten soll und das NIST ist ein Institut zur Standardisierung.

SHA steht für Secure Hash Algorithm und ist, wie der Name schon sagt, eine kryptographische Hashfunktion. Ihr eigentlicher Verwendungszweck sollte die digitale Signatur sein, doch in der Realität wird SHA-1 auch zu anderen Zwecken verwendet. Der Algorithmus entspricht sowohl den Secure Hash Standards als auch dem Standard für Digitale Signaturen (DSS)

SHA-1 wird dazu verwendet, Integritätschecks von Software zu machen, Passwörter zu speichern oder digitale Signaturen zu erzeugen. SHA-1 hat weiters einen Einsatzort in S/MIME als auch in OpenPGP. IPsec als auch SSH und TPA setzen auf diese kryptographische Hashfunktion. Neben RSA, AES und MD5 ist SHA-1 eine der meistverwendeten Sicherheitsalgorithmen und sehr viel hängt von seiner Sicherheit ab. MD5 wurde schon als unsicher erklärt. Wie lange dauert das noch bei SHA-1? Durch die immer schneller werdenden Rechner ist es notwendig, die Hashwerte immer länger werden zu lassen. SHA-1 wird deshalb im Laufe der Jahre von einem Nachfolger ersetzt werden, der längere Hashwerte benützt und dessen Entschlüsselung sehr schwer ist.

Unmöglich wird es nie sein, da die Brute Force Attacke immer möglich sein wird und dadurch auch bei sehr langen Hashwerten wieder Kollisionen gefunden werden können.

SHA-1 erfüllt die Voraussetzung, dass zwei nahe aneinander liegende Werte sehr verschiedene Werte haben. Zum Beispiel die beiden Zeichenketten „Test1“ und „Test2“

Test1: 99ea7bf70f6e69ad71659995677b43f8a8312025
Test2: 2b84f621cofd4ba8bd514c5c43ab9a897c8c014e

5.1. Algorithmus:

Wie der Pseudocode für SHA-1 zum Beispiel aussehen kann, sieht man im folgenden Code, der aus <http://en.wikipedia.org/wiki/SHA-1> kopiert ist.

Note: All variables are unsigned 32 bits and wrap modulo 2^{32} when calculating

Initialize variables:

h0 := 0x67452301
h1 := 0xEFCDAB89
h2 := 0x98BADCFE
h3 := 0x10325476
h4 := 0xC3D2E1F0

Pre-processing:

append a single "1" bit to message
append "0" bits until message length $448 - 64 \pmod{512}$
append length of message, in *bits* as 64-bit big-endian integer to message

Process the message in successive 512-bit chunks:

break message into 512-bit chunks
for each chunk
 break chunk into sixteen 32-bit big-endian words $w(i)$, $0 \leq i \leq 15$

Extend the sixteen 32-bit words into eighty 32-bit words:

for i **from** 16 to 79
 $w(i) := (w(i-3) \mathbf{xor} w(i-8) \mathbf{xor} w(i-14) \mathbf{xor} w(i-16)) \mathbf{leftrotate} 1$

Initialize hash value for this chunk:

a := h0
b := h1
c := h2
d := h3
e := h4

Main loop:

for i **from** 0 to 79

```

if 0 ≤ i ≤ 19 then
  f := (b and c) or ((not b) and d)
  k := 0x5A827999
else if 20 ≤ i ≤ 39
  f := b xor c xor d
  k := 0x6ED9EBA1
else if 40 ≤ i ≤ 59
  f := (b and c) or (b and d) or (c and d)
  k := 0x8F1BBCDC
else if 60 ≤ i ≤ 79
  f := b xor c xor d
  k := 0xCA62C1D6

temp := (a leftrotate 5) + f + e + k + w(i)
e := d
d := c
c := b leftrotate 30
b := a
a := temp

```

Add this chunk's hash to result so far:

```

h0 := h0 + a
h1 := h1 + b
h2 := h2 + c
h3 := h3 + d
h4 := h4 + e

```

digest = hash = h0 **append** h1 **append** h2 **append** h3 **append** h4 (*expressed as big-endian*)

Zuerst müssen einige Variablen definiert werden. Danach beginnt die eigentliche Verarbeitung der Daten beginnend mit einer Vorverarbeitung. Es wird ein Einser an die mittels SHA-1 zu verschlüsselnde Nachricht angehängt und dann werden so viele Nullen angehängt, bis die zu verschlüsselnde Nachricht eine Länge hat, die ein Vielfaches von 512 ist. Als letzter Schritt der Vorverarbeitung wird noch die Länge der ursprünglichen Nachricht im Big-Endian Format angehängt. Big-Endian Format heißt, dass höherwertige Bits vor niedrigerwertigen Bits gespeichert werden. Die Vorverarbeitung wird genau einmal pro Verschlüsselung durchgeführt und spielt somit für die Dauer der Verschlüsselung keine wesentliche Rolle.

Nach der Vorverarbeitung wird die zu verschlüsselnde Nachricht in mehrere kleinere Teile gebrochen. Diese Teile haben jeweils eine Länge von 512 Bit. Diese Teile werden nun weiter in 32 Bit Werte gebrochen, die abermals im Big-Endian Format abgespeichert werden. Danach kann die eigentliche Verarbeitung beginnen. SHA-1 ist ein 80 Runden langes Verfahren, bei dem in jeder Runde verschiedene Variablen verodet, verodert oder mittels xor verknüpft werden. Die 80 Runden werden hierfür in vier Gruppen zu je 20

Runden zusammengefasst, die jeweils eine Kombination aus den genannten logischen Operationen darstellt. In der ersten Gruppe ist diese Funktion $(b \text{ and } c) \text{ or } ((\text{not } b) \text{ and } d)$, in der zweiten Gruppe $b \text{ xor } c \text{ xor } d$, in der dritten Gruppe $(b \text{ and } c) \text{ or } (b \text{ and } d) \text{ or } (c \text{ and } d)$ und schlussendlich sieht die Funktion in der vierten Gruppe abermals so aus, wie in der zweiten Gruppe.

Nach den 80 Runden hat man die Ergebnisse in fünf Variablen aufgeteilt. Die Ergebnisse werden addiert und geschiftet und schlussendlich verkettet. Das endgültige Ergebnis ist im Big-Endian Format. Es ist nicht zwingend notwendig, dass das Big-Endian Format verwendet wird, doch sinnvoll, es gegen das Little-Endian Format auszutauschen ist es nur, falls das berechnende Gerät auf Little-Endian optimiert wurde.

In der Initialisierungsphase werden diese fünf Variablen gesetzt.

```
h0 := 0x67452301
h1 := 0xEFCDAB89
h2 := 0x98BADCFE
h3 := 0x10325476
h4 := 0xC3D2E1F0
```

Diese 5 Variablen sind spezifisch für SHA-1. SHA-1 Nachfolgerversionen wie beispielsweise, SHA-256 oder auch SHA-384, haben erstens nicht fünf sondern mehr Variablen die zum Verschlüsseln verwendet werden, und zweitens sind bei Variablen von SHA-256 und SHA-384 andere Initialisierungswerte gegeben.

5.2. Eigenschaften

SHA-1 ist langsamer als die Verfahren der MD-Serie wie zum Beispiel MD4 oder MD5. Grund dafür sind die vielen Runden, die durchlaufen werden und der längere Schlüssel. SHA-1 ist spezifiziert als ein 80 Runden langes Verfahren, dass, obwohl es ein wenig langsamer ist als einige andere Algorithmen, ein sehr breites Einsatzspektrum hat. Es ist derzeit eine der meistverwendeten kryptographischen Hashfunktionen. Es gibt viele verschiedene Untervarianten und Ausprägungen von SHA-1, bei denen unter anderem die Rundenanzahl variieren kann. Diese entsprechen allerdings nicht mehr allen Standards, denen SHA-1 unterliegt. Bei Veränderungen ist im Einzelfall die Sicherheit des Algorithmus zu prüfen.

Ein maßgeblicher Vorteil von viel verwendeten Algorithmen wie beispielsweise SHA-1 ist, dass viele Leute sich mit verschiedenen Aspekten des Algorithmus auseinandersetzen und somit auch schneller mögliche Fehler und Lücken in der Sicherheit finden. Security by Obscurity hat sich in der Praxis nicht sonderlich gut bewährt, da ein guter Algorithmus auch noch sicher ist, wenn man die Methodik kennt. Insofern ist es gut, über die Sicherheitsrisiken im speziellen jetzt für SHA-1 bescheid zu wissen und dementsprechend falls möglich, schützend einzugreifen.

Die Kryptoanalyse hat nun folgendes Ziel: Sie soll Mängel im Algorithmus finden. Die Schwachstelle der kryptographischen Hashfunktionen sind Kollisionen. Kollisionen sind zwei Werte, die sich nur durch den realen Wert unterscheiden, deren Hashwert jedoch gleich ist. Diese gilt es zu finden und ein gewisses Muster darin zu entdecken.

Ein Kollisionsbeispiel in einem 58-stufigen SHA1 [13]

$$h_1 = \text{compress}(h_0, M_0) = \text{compress}(h_0, M'_0)$$

h_0 : 67452301 efc dab89 98badcfe 10325476 c3d2e1f0

M_0 : 132b5ab6 a115775f 5bfddd6b 4dc470eb
 0637938a 6cceb733 0c86a386 68080139
 534047a4 a42fc29a 06085121 a3131f73
 ad5da5cf 13375402 40bdc7c2 d5a839e2

M'_0 : 332b5ab6 c115776d 3bfddd28 6dc470ab
 e63793c8 0cceb731 8c86a387 68080119
 534047a7 e42fc2c8 46085161 43131f21
 0d5da5cf 93375442 60bdc7c3 f5a83982

h_1 : 9768e739 b662af82 a0137d3e 918747cf c8ceb7d4

Die gezeigte Kollision reduziert den Algorithmus von ursprünglich 80 auf 58 Runden. Dies hat zur Folge, dass man schneller durchprobieren kann. Die kollidierenden Nachrichten M_0 und M'_0 sind hierbei sicher nicht die einzigen, die sich im Hashabbild gleichen. Das Finden dieses Paares hat zur Folge, dass man auch andere Kollisionen erzeugen kann. Es ist am schwierigsten, die erste Kollision zu finden. Von ihr kann man weitere Kollisionen ableiten. Ein Theorem, nachdem vorgegangen wird, ist die Geburtstagsattacke.

5.3. Aktuelle Angriffe auf SHA-1 und Sicherheit

Wie bereits beschrieben nutzen sehr viele sicherheitsrelevante Anwendung SHA-1. In letzter Zeit kursierten viele Meldungen, SHA-1 sei gebrochen. Die endgültige Studie ist noch nicht publiziert, jedoch wackelt das ganze Gedankenwerk um SHA-1. Viele Leute sind sehr verunsichert, da ihre Programme auf SHA-1 aufbauen. Fakt ist, dass es bereits möglich ist, Kollisionen zu erzeugen, was das Verfahren allgemein schwächer macht. Für private Anwendungen ist die Sicherheit sicherlich auch weiterhin ausreichend, doch für hochgradig sicherheitskritische Anwendungen wird man sich um Verfahren mit einem längeren Hashwert oder eventuell einem anderen Algorithmus bemühen müssen. SHA-1 ist MD4 sehr ähnlich, hat jedoch einen längeren Hashwert. Insofern ist anzunehmen, dass die Ergebnisse der bereits gebrochenen Funktion MD4 dazu führen, dass SHA-1 auch gebrochen wird, da eventuell dieselbe oder zumindest eine vom Konzept her ähnliche Vorgangsweise angewendet werden kann. Krypto-Experten wie Schneier und Kaliski gehen davon aus, dass man SHA-1 durchaus noch recht unbesorgt einsetzen

kann, da man bislang immer noch 2^{69} Varianten probieren muss, um sicher auf eine Übereinstimmung zu kommen. Dies dauert zwar recht lange aber ist mit sehr hoher Rechnerleistung sicherlich möglich. Der zweite Grund, wieso sie von der Sicherheit von SHA-1 ausgehen, ist deshalb fast noch wichtiger: Um beispielsweise einen digital signierten Vertrag nachträglich zu fälschen, müsste der Angreifer einen Preimage-Angriff durchführen. Er müsste also zum vorgegebenen Hash-Wert des echten Vertrags einen zweiten, gefälschten Vertrag finden, der denselben Hash-Wert ergibt. Das erfordert schon prinzipiell sehr viel mehr Operationen (2^{160}) [14]

Das NIST (National Institute of Standards and Technology) hat bereits offizielle Pläne geäußert, SHA-1 durch SHA-256 und SHA-512 zu ersetzen. Diese Änderung wird viele Institutionen, insbesondere in den USA, treffen. [17]

Diese Erhöhung des Aufwandes gilt allerdings nur, wenn der Hashwert nicht frei gewählt werden kann.

Ein weiteres großes Einsatzgebiet von SHA-1 neben dem der digitalen Signatur, ist die Verschlüsselung von Passwörtern. Dies wird sowohl im Web auf diversen Seiten mit passwortgeschützten Bereichen praktiziert, als auch in Betriebssystemen. Hier gilt es noch als aussichtslos, Passwörter aufgrund eines Hashwertes zu knacken. Hier bilden andere Attacken eine weitaus größere Gefahr. [14]

Es gibt weltweit einige Forschergruppen, deren Ziel es ist, kryptographische Hashfunktionen zu brechen. Einige von diesen sind in China beheimatet, unter anderem auch diejenige, die jetzt behauptet, sie hätten einen in der Praxis praktikablen Angriff auf SHA-1. Genauere Ergebnisse sollen auf einem der nächsten Kongresse über dieses Thema veröffentlicht werden. Abhängig von den Forschungsergebnissen dieser Gruppe, wird sich wohl auch zwangsläufig entscheiden, welcher Algorithmus den Platz von SHA-1 einnehmen wird. Es ist unter Garantie eine kryptographische Hashfunktion mit einem Hashwert, der länger ist als 160 Bit, eventuell ein Nachfolger aus der SHA-Reihe.

6 Konstruktion von parametrisierten Hashfunktionen:

Ebenso wie für die parameterlosen Hashfunktionen gibt es auch für die Erstellung parametrisierten Hashfunktionen mehrere Möglichkeiten.

Genau wie parameterlose Hashfunktionen können auch MAC (wir werden den Ausdruck MAC in diesem Abschnitt als Synonym für parametrisierte Hashfunktionen benutzen – der Kürze wegen) durch hintereinander Ausführen einer Kompressionsfunktion erzeugt werden, nur muss der Parameter k (auch Schlüssel k) zusätzlich in die Ausgabe mit einfließen.

Auch MACs können auf verschiedenen kryptographischen Elementen basieren. So z.B. auf Blockchiffren, Stromchiffren und MDCs, also nicht parametrisierten Hashfunktionen. Wie bei den nicht parametrisierten Hashfunktionen gibt es auch für MACs speziell entwickelte Funktionen.

Auf Blockchiffren basierende MACs benützen die Blockchiffren meist im CBC – Modus. Dabei kann wie bei der normalen Verschlüsselung vorgegangen werden (der Eingabestring wird auf Blöcke der Länge n aufgeteilt, und die Blöcke der Reihe nach verarbeitet, wobei jeder Block n mit dem Ergebnis der Verarbeitung des vorangegangenen Blocks XOR verknüpft wird, bevor er die eigentliche Blockchiffre durchläuft und dort mit dem Schlüssel verknüpft wird), nur dass das Endergebnis sich nicht aus den Ergebnissen der Verarbeitung der einzelnen Blöcke zusammensetzt, sondern nur aus dem Ergebnis des letzten Blocks besteht (in das alle vorherige Blöcke eingegangen sind). Beispiele für auf Blockchiffren basierende MACs DES-CBC MAC, sowie OMAC oder PMAC.

[3] rät zur besonderer Vorsicht, falls MACs auf nicht parametrisierten Hashfunktionen (meist MDCs) basieren. Hierbei wird der Schlüssel des MACs als Teil des Eingabestrings behandelt. Die Gefahr dabei ist, dass an nicht parametrisierte Hashfunktionen andere Anforderungen als an MACs gestellt werden, und erstere angenommene Eigenschaften oft nicht erfüllen. So reicht z.B. die Einweg- Eigenschaft nicht aus, um auch zu garantieren, dass mit Hilfe eines MAC - Wertes nicht MAC – Werte von anderen Nachrichten erzeugt werden können, auch ohne dass der Schlüssel aus der ursprünglichen Nachricht extrahiert werden kann.

Ein Beispiel für einen auf einer nicht parametrisierten Hashfunktion basierenden MAC ist HMAC. (Eine kurze Beschreibung findet sich u.a. in [9]).

Beispiele für speziell entwickelte MACs sind MAA und MD5-MAC (Kurze Beschreibungen finde sich in [3]).

7 Angriffe

Es gibt zwei verschiedene Ansätze Hashfunktionen anzugreifen. Einerseits den Preimage-Angriff und andererseits den Kollisionsangriff. Die beiden Varianten haben verschiedene Fragestellungen:

Preimage-Angriff: Wie schwer ist es, zu einem vorgegebenen Hash-Wert eine Nachricht zu erzeugen, die denselben Hash-Wert ergibt?

Kollisionsangriff: Wie schwer ist es, zwei verschiedenen Nachrichten mit gleicher Prüfsumme zu finden? [14]

Beide Angriffsarten können verschieden realisiert werden. Der Brute Force Angriff, einem Angriff mit Rechnergewalt, kann beispielsweise beide Arten verwirklichen. SHA-1 bildet einen Hash-Wert mit 160 Bit. Es gibt somit nur eine endliche Anzahl von Möglichkeiten Hash-Werte zu bilden (2^{160}). Um alle durchzuprobieren, würde es viele, viele Jahre dauern. Falls es eine Möglichkeit gibt, das Verfahren abzukürzen, gilt die Funktion als gebrochen.

7.1. Der Brute Force Angriff

Für diesen Angriff ist es nicht notwendig, einen sonderlich komplizierten Algorithmus zu bauen. Der Angriff ist viel eher durch die Rechenleistung beeinflussbar. Es werden Hashwerte gebildet und daraufhin mit bereits errechneten Resultaten verglichen. Bei einer Übereinstimmung hat man eine Kollision gefunden. Dieser Angriff ist mit zunehmender Hashlänge nicht sonderlich effizient. Deshalb wird dieses Verfahren nur in Kombination mit anderen Verfahren verwendet. Beispielsweise der Geburtstagsattacke.

Theoretisch ist ein Erfolg bei Brute Force Angriffen immer möglich, jedoch sinkt die Wahrscheinlichkeit mit steigender Schlüssellänge. Gerade deshalb sind in der Kryptographie die verwendeten Schlüssel sehr lang. Schwachstelle bleibt jedoch auf jeden Fall der Benutzer, da man von einem Benutzer eines sicherheitskritischen Systems nicht verlangen kann, dass er sich ein beispielsweise 30 Zeichen langes Passwort merken kann.

7.2. Geburtstagsattacke

Die Geburtstagsattacke ist eine der schwerwiegendsten Attacken gegen Hashfunktionen. Dabei wird nach Kollisionen gesucht. Kollisionen sind zwei Werte die sich nur durch den realen Wert unterscheiden, deren Hashwert jedoch gleich ist. Kollisionen stellen, auch bzw. gerade wegen ihrer Unvermeidbarkeit, das zentrale Sicherheitsrisiko von Hashfunktionen dar.

Der Geburtstagsattacke liegt das Geburtstagsparadoxon zu Grunde, welches lautet: *„In einem Raum müssten 183 Personen sein, damit mit einer Wahrscheinlichkeit größer 50 % eine Person anwesend ist, die den gleichen Geburtstag hat wie der Gastgeber. Es müssen aber nur 23 sein, um mit einer Wahrscheinlichkeit größer 50 % zwei Personen mit demselben (aber beliebigen) Geburtstag zu finden.“* [12]. Dies wird nun auf die Länge des Hashwertes umgelegt, und somit ein Vergleich für die Zeitdauer des Entschlüsselns ermittelt.

Angenommen, man arbeitet mit 64-Bit-Hashwerten, so würde es (bei einer Rechenleistung von 1 Mio. Hashwerten pro Sekunde) rund 600.000 Jahre dauern, bis ein zweites Dokument mit dem gleichen Hashwert wie ein vorgegebenes Dokument gefunden wäre. Um ein Paar von Dokumenten mit übereinstimmendem Hashwert zu finden, benötigte derselbe Rechner aber nur rund eine Stunde. [12]

Um kryptographische Hashfunktionen sicherer zu machen, als sie derzeit sind, müsste man die Hashwerte die erzeugt werden, bei weitem länger machen. Die Rede ist von 160 Bits oder mehr.

Das Problem der Kollisionen wäre nicht, dass man ab und zu welche findet, sondern dass gefundene Kollisionen ein Muster ergeben können. Sobald solch ein Muster gefunden ist, ist eine kryptographische Hashfunktion unbrauchbar, insbesondere für die digitale Signatur.

Eine generelle Problematik der digitalen Signatur ist die sog. „Man in the middle Attack“. Dabei leitet ein potentieller Angreifer die Kommunikation zweier anderer (A und B) über sich um. Er kann somit nicht nur die Kommunikation abhören, sondern auch Daten während der Übertragung verändern.

Da die Daten signiert sein können, und der Empfänger merken könnte, dass etwas verändert wurde, falls die Signatur nicht stimmt, muss sich der Angreifer eine Taktik überlegen. Einerseits kann er sich den Signierungsschlüssel des Senders beschaffen, um mit Hilfe dieses Schlüssels ein Dokument anzufertigen. Eine Möglichkeit, für die man keine Kryptoanalyse benötigt, zumindest nicht unbedingt. Eine andere Möglichkeit ist es, ein zweites Dokument anzufertigen, welches denselben Hashwert wie das Originaldokument hat. Hierbei handelt es sich um das Ausnutzen einer Kollision. Und genau zu diesem Zweck bedient man sich der Kollisionen.

Diese Gefahr besteht bei SHA-1 derzeit noch nicht. Es gibt noch kein Verfahren, welches so schnell eine Kollision finden könnte, dass A und B oder zumindest der Empfänger, nicht mitbekommt, dass mit der Nachricht etwas nicht stimmt. Ausnahme wäre natürlich, dass der Angreifer Daten bezüglich der Verschlüsselung des Senders kennt.

Als praktisches Beispiel die Geburtstagsattacke nach Yuval [3]:

INPUT: Vorgegebene Nachricht x_1 ; Gesuchte Nachricht x_2 ; m -bit Einweg-Hashfunktion h .

OUTPUT: x'_1 , x'_2 Ergebnisse der kleinen Änderungen von x_1 , x_2 mit $h(x'_1) = h(x'_2)$ (Das bedeutet, dass x'_1 und x'_2 den selben Hashwert haben).

1. Generiere $t = 2^{(m/2)}$ kleine Modifikationen x'_1 von x_1 .
2. Erzeuge den Hashwert jeder solcher modifizierten Nachricht und speichere diese Werte (Gruppiert nach entsprechenden Nachrichten) so, dass Teile davon gesucht und effizient gefunden werden können. Da es sich sowieso um eine Hashtabelle handelt, ist der Aufwand $O(t)$ realistisch.
3. Erzeuge Modifizierungen für x_0
2 von x_2 , berechne $h(x'_2)$ für jeden wert und überprüfe auf Übereinstimmungen bisheriger Ergebnisse. Führe es sooft durch, bis ein Ergebnis gefunden wurde.

Im Grunde sind alle Geburtstagsattacken sehr ähnlich. Es gibt natürlich kleinere Unterschiede, doch die Geburtstagsattacke nach Yuval zeigt sehr schön die Charakteristik und die Vorgehensweise bei dieser Art von Angriffen.

8 Zusammenfassung und Diskussion

Wir haben einen Überblick über die Eigenschaften und den daraus entstehenden Arten von Hashfunktion gegeben. Es ist klar, dass stärkere Bedingungen für Hashfunktionen nicht nur schwerer zu erreichen, sondern auch mit höheren Kosten verbunden sind. Hashfunktionen werden heutzutage meist aus Kompressionsfunktionen erstellt, mit Hilfe der Algorithmen von Damgård und Merkle. Zur Erstellung der Kompressionsfunktionen existieren mehrere Möglichkeiten, wobei die Verwendung von Blockchiffren oder modularer Arithmetik meist darauf abzielt, bereits vorhandene Komponenten wieder verwenden zu können. Ein zweiter Wunsch beim Einsatz dieser Methoden ist eine „automatisch“ sichere Hashfunktion, unter der Voraussetzung, dass die zugrunde liegende Methode gewisse Eigenschaften besitzt. Welche Eigenschaft dies sein müssen ist leider noch nicht vollständig bekannt.

Speziell für Hashfunktionen entworfene Algorithmen haben meist einen Effizienzvorteil, was ihre Verarbeitung in einem Computersystem anbelangt. Ihre Sicherheit muss jedoch extra bewiesen werden, bzw. ist darüber von vorneherein nichts bekannt. SHA-1 ist ein Beispiel einer der der am weitesten verbreiteten Hash- Algorithmen heutzutage.

Bei den Überlegungen zur Sicherheit wird deutlich, dass heutige kryptographische Hashfunktionen eine Balance zwischen Effizienz (beim Berechnen und Speichern) und der Sicherheit sind. Dies schlägt sich zum einen in der Länge des Hashwertes nieder. Das Grundproblem bei kryptographischen Hashfunktionen sind die, auf Grund der Kompression unvermeidlichen, Kollisionen.

Es hat sich in der Vergangenheit (MD4, MD5, SHA-1,...) gezeigt, dass kryptographische Hashfunktionen meist nur für einen gewissen Zeitraum sicher sind, und dass man überlegen sollte, von Zeit zu Zeit die Länge der Hashwerte zu erhöhen, zum Einen um der steigenden Rechenleistung Rechnung zu tragen, und zum Anderen, da nach einiger Zeit in den meisten der verwendeten Hashfunktionen Schwachstellen gefunden wurden, wodurch bei einem weiteren Einsatz der Funktion die Sicherheit nicht mehr gegeben ist. Eine Auseinandersetzung mit der Problematik ist somit unumgänglich, zumal noch nicht bekannt ist, ob es wirklich sichere Kompressions -, und damit auch Hashfunktionen, überhaupt gibt.

Quellen:

- [1] J. Buchmann. Einführung in die Kryptographie. Springer-Verlag Berlin Heidelberg 2004, 3.Auflage. Berlin, Deutschland. ISBN: 3-540-40508-9
- [2] A. Beutelspacher, J. Schwenk, K-D Wolfenstett. Moderne Verfahren der Kryptographie. Vieweg 1999. 3. Auflage. Göttingen, Deutschland. ISBN: 3-528-26890-6
- [3] A. Menezes, P. van Oorschot, s. Vanstone. Handbook of Applied Cryptographie. CRC-Press 1997. ISBN: 0-8493-8523-7 (kapiteldownload unter <http://www.cacr.math.uwaterloo.ca/hac/>)
- [4] <http://www.rsasecurity.com/rsalabs/node.asp?id=2176>
- [5] R. C. Merkle. A certified digital signature. Lecture Notes in Computer Science. Vol. 435: Proceedings: Advances in Cryptology – CRYPTO '89. Springer Berlin- Heidelberg 1990. p. 218-238
- [6] R. C. Merkle. One Way Hash Funktionen und DES. Lecture Notes in Computer Science. Vol. 435: Proceedings: Advances in Cryptology – CRYPTO '89. Springer Berlin- Heidelberg 1990. p. 428-445
- [7] I.B. Damgård. A Design Principle for Hash Functions. Lecture Notes in Computer Science. Vol. 435: Proceedings: Advances in Cryptology – CRYPTO '89. Springer Berlin- Heidelberg 1990. p. 416-427
- [8] <http://www.rsasecurity.com/rsalabs/node.asp?id=2177>
- [9] <http://en.wikipedia.org/wiki/HMAC>
- [10] <http://de.wikipedia.org/wiki/Hashfunktion>
- [11] http://en.wikipedia.org/wiki/Hash_function
- [12] <http://www.remote.org/frederik/projects/cash/cash-3.html>
- [13] <http://cryptome.org/sha1-attacks.htm>
- [14] <http://www.heise.de/security/artikel/56555>
- [15] <http://en.wikipedia.org/wiki/SHA-1>
- [16] <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf>
- [17] <http://www.fcw.com/fcw/articles/2005/0207/web-hash-02-07-05.asp>