Aktuelle Angriffe auf SHA und MD5

Helmut Petritsch

Inhaltsverzeichnis:

Was bedeutet Angriff auf Hash Algorithmus?

Wann ist eine Hash Algorithmus gebrochen?

Ein Hash Algorithmus gilt im kryptographischen Sinne als gebrochen, wenn es eine Methode oder einen Angriff gibt, der schneller eine Kollision herbeiführt als eine Brute Force Attacke. [SCHNEIER]

Eine Kollision ist, wenn ein Hash Algorithmus für zwei unterschiedliche Dokumente den gleichen Hashwert ergibt, also diese beiden Dokumente nicht aufgrund eines eindeutigen Hash Wertes voneinander differenziert werden können. Je größer der Hash Wert, desto schwieriger ist es theoretisch eine Kollision herbeizurühren, da die Anzahl der möglichen Ergebnisse des Hash Algorithmus mit der Anzahl der bit quadratisch steigen. Praktisch ist dieser Zusammenhang nicht unmittelbar gegeben, da ein großer Hash Wert noch nicht zwingender Maßen eine sichere Hash Funktion ergibt.

So führen Schwächen in einem Hash-Algorithmus dazu, dass das Verfahren abgekürzt werden kann (Reduktion der Runden), wenn sich zum Beispiel bestimmte bits über mehrere Runden hinweg nicht auswirken. Auch eine größere Anzahl an Runden in einem Hash-Algorithmus muss nicht unbedingt eine Steigerung der Sicherheit mit sich führen, wie der Angriff von [BIHAM] auf SHA-0 zeigt.

Eine Brute Force Attacke beschreibt einen „naiven“ Angriff, der, ohne weitere Annahmen über das System oder von dem System etwas zu wissen, (alle) mögliche(n) Varianten ausprobiert, bis ein richtiges Dokument gefunden wurde. Dies ist sowohl für Angriffe auf Passwörter (Ausprobieren aller möglichen Kombinationen von Zeichen mit einer Länge bis zu – z.B. für Windowssystem – 16) als auch für Angriffe auf Hash - Algorithmen möglich: Erstellen von Dokumenten, Durchführen der Hash – Funktion, überprüfen des Ergebnisses. Die Einfachheit dieser Sorte von Angriff hat einen offensichtlichen Nachteil: Die Durchführung ist rechen- und somit zeitaufwändig.

Auswirkungen und Folgen

Wenn ein Algorithmus als gebrochen gilt bedeutet dies jedoch nicht zwingender weise, dass er für den täglichen Gebruach nicht mehr sicher genug ist. (Es kann dies bedeuten, muss aber nicht.)

So reduziert sich zum Beispiel der Rechenaufwand für einen Kollisions-Angriff auf SHA-1 aufgrund der Erkenntnisse der Chinesischen Forschungsgruppe im Februar 2005 von 2^{80} auf 2^{69} nötige Berechnungen, aber selbst für diese 2^{69} Möglichkeiten braucht auch ein Hochleistungsrechner Jahre um einen Angriff erfolgreich durchzuführen. So gilt SHA-1 seit Februar 2005 als gebrochen, der Einsatz kann

auch für Anwendungen mit hohen Sicherheitsanforderungen (Bankverbindungen, etc.) als sicher angesehen werden.

„Das rührt daher, dass auch 2^{69} noch eine recht große Zahl ist. Auf Spezialhardware kann man eine Hash-Operation in etwa 40 Takten ausführen. Selbst wenn man diese von 33 MHz auf 4 GHz beschleunigen könnte, würde sie immer noch 170.000 Jahre benötigen. Selbst ein Riesen-Cluster aus solchen Maschinen könnte innerhalb eines realistischen Zeitraums weniger Jahre keine Kollision finden.“[1]

Dieser Angriff zeigt auf der anderen Seite auch, dass der SHA-1 Algorithmus früher als erwartet gebrochen werden konnte, und dass die Suche nach Nachfolgern für SHA-1 beschleunigt werden muss.

So ist nicht zu erwarten, dass auf SHA-1 in den nächsten Jahren Kollisions- oder gar Preimageangriffe auch mit großer Rechenleistung in wenigen Jahren durchzuführen sind. Angriffe, die eine Reduktion der nötigen Operationen bei Preimage – Angriffen herbeiführen (SHA-1 und MD5), sind bis heute nicht bekannt.

Allerdings benötigt die Entwicklung, Testung, Implementierung und Einführung eines neuen Algorithmus ein erhebliches Maß an Zeit. (Von der Ausschreibung bis zur Wahl von AES vergingen fünf Jahre). Auch kann erst der Einsatz eines solchen Algorithmus zeigen, ob und wie sicher er ist.

Das Wechseln zu auf SHA-1 aufbauende Algorithmen (SHA-224, SHA-256, etc) kann auch nur eine kurzfristige Lösung sein, da nicht bekannt ist, inwieweit die in SHA-1 gefundenen Schwächen auch auf diese zutreffen. Aber auch das sofortige Umsteigen auf andere Algorithmen wie Whirlpool muss nicht unbedingt eine Steigerung der Sicherheit mit sich führen, da diese Algorithmen weniger gut erforscht sind und damit möglicherweise die selben oder mehr Probleme auftreten als beim (zwar im kryptographischen Sinn gebrochenen, aber gut erforschten und erprobten) SHA-1.

[1] http://www.heise.de/security/artikel/print/56555

Grundsätzliche Unterscheidung der Angriffe

Preimage Attack

Beim Preimage Angriff soll für eine bestehende (dem Angreifer aber unbekannte) Nachricht eine Nachricht gefunden werden, die denselben Hash - Wert ergibt, also zum bekannten Hashwert h eine Nachricht m, die diesen Hash-Wert ergibt:

$$h = hash(m)$$

Um diesen Angriff als Brute Force Attacke auszuführen, müssen für einen 160 bit langen Hash – Wert ca. 2^{160} (ca. $1{,}46*10^{48}$) Dokumente erstellt, deren Hash – Wert berechnet und mit dem gesuchten Hash – Wert verglichen werden. Bei ca. 2^{160} gehashten Dokumenten ist die statistische Wahrscheinlichkeit dann sehr groß, ein Dokument mit gleichem Hash – Wert zu finden.

Kollisionsangriff

Beim Kollisionsangriff sollen zwei Nachrichten m_1 und m_2 gefunden werden, die die gleiche Prüfsumme ergeben.

$$hash(m_1) = hash(m_2)$$

Bei diesem Angriff ist die Brute Force Attacke wesentlich schneller als beim Preimage Angriff: Um zwei Dokumente mit gleichem 160 bit langen Hash-Wert zu finden müssen „nur“ ca. 2^{80} ($1{,}28*10^{24}$) Dokumente erstellt, deren Hash-Werte erzeugt und mit den anderen verglichen werden.

Diese deutliche Vereinfachung bei einem 160 bit Hash um den Faktor 10^{24} ergibt sich aus dem Geburtstagsparadoxon.

Geburtstagsparadoxon

Das Geburtstagsparadoxon beschreibt die Frage, wie groß die Wahrscheinlichkeit ist, dass von einer bestimmten Anzahl zwei Personen am gleichen Tag Geburtstag haben.

Zunächst wird Q, die Wahrscheinlichkeit vom Gegenteil (Komplementärmenge) berechnet, also dass alle X Personen an einem anderen Tag Geburtstag haben. Danach errechnet sich die Wahrscheinlichkeit P, dass 2 oder mehr Personen an einem Tag Geburtstag haben mit

$$P = 1 - Q$$

Berechnung der Komplementärmenge:

Für die einzelnen Personen ist die Wahrscheinlichkeit, an einem noch nicht „belegten“ Tag Geburtstag zu haben,

$$Q = \frac{Anzahl\_der\_verfügbare\_Tage}{Anzahl\_der\_möglichen\_Tage}$$

Wobei die Anzahl der möglichen Tage der Einfachheit halber als 365 Tage (ohne 29. Februar) angenommen seinen.

Für die erste Person errechnet sich Q daher als

$$Q_1 = \frac{365}{365}$$

Die zweite Person hat nun einen Tag weniger zur Auswahl (den Tag, an dem Person 1 Geburtstag hat), Q ist daher

$$Q_2 = \frac{364}{365}$$

Für die X-te Person (bzw. allgemein) gilt damit

$$Q_X = \frac{365 - X + 1}{365}$$

Aus diesen Einzelwahrscheinlichkeiten $Q_1 - Q_X$ lässt sich Q errechnen (wie groß die Wahrscheinlichkeit ist, dass alle Personen an unterschiedlichen Tagen Geburtstag haben):

$$Q = Q_1 * Q_2 * ... * Q_X$$

Oder, anders angeschrieben

$$Q = \frac{365 * 364 * ...(365 - X + 1)}{365^X} = \frac{365!}{(365 - X)! * 365^X}$$

So ist für eine Gruppe mit 23 Personen, dass 2 Personen am gleichen Tag Geburtstag haben bereits > 50%.

Der gleiche Ansatz lässt sich auf für Hash – Algorithmen verfolgen. Die Personen werden zu Dokumenten, die Geburtstage die jeweils zugehörigen Hash Werte. Die Aufgabe ist also, zwei Dokumente (Personen) zu finden, die den gleichen Hash Wert (Geburtstag) haben. Um einen Kollisionsangriff erfolgreich durchführen zu können, stellt sich also die Frage, wie viel Dokumente erstellt und geprüft werden müssen, um eine Wahrscheinlichkeit größer 50% zu erreichen, dass zwei dieser Dokumente den gleichen Hashwert haben. Im Fall eines 160 bit Hashwertes stellt sich also die Frage, die groß X sein muss, sodass

$$1 - Q = 1 - \frac{\left(2^{160}\right)\_!}{(2^{160} - X)! * 2^{160^X}} > 0.5$$

Second Preimage Attack

Im Gegensatz zur Preimage Attack ist bei der Second Preimage Attack die Message bekannt und es soll eine zweite Message gefunden werden, die denselben Hash

Wert ergibt. So ist die bei der Preimage Attack der Hash-Wert h gegeben, gesucht ist eine Nachricht m, sodass

$$h = hash(m)$$

Bei der Second Preimage Attack ist die Nachricht m_1 und damit der zugehörige Hash-Wert h bekannt, gesucht ist eine Nachricht m_2, sodass

$$hash(m_1) = hash(m_2)$$

Zertifikate und digitale Unterschriften (Signaturen)

Zertifikate helfen bei der Authentifizierung zwischen zwei Kommunikationspartnern mit verschlüsselter Verbindung. Im Wesentlichen ist die Aufgabe eines Zertifikats sicherzustellen, dass der in einem Zertifikat übergeben öffentliche Schlüssel tatsächlich dem erwarteten Kommunikationspartner zugewiesen werden kann. Wenn dies nicht sichergestellt ist, würden die Daten zwar verschlüsselt und damit abhörsicher über das Netzwerk, respektive Internet, übertragen werden, es könnte aber nicht sichergestellt werden, dass die möglicherweise sensitiven Daten auch bei jenem Empfänger ankommen, für den sie gedacht waren. Ohne eine gesicherte Authentifizierung wäre es relativ einfach, Man in the Middle Attacken auch auf verschlüsselte Verbindungen durchzuführen.

Um diese Authentifizierung durchführen zu können, signieren so genannte Certificate Authorities (CA) Zertifikate mit einer digitalen Unterschrift. Für die Erstellung dieser digitalen Unterschrift werden eine Hash-Funktion sowie ein asymmetrisches Verschlüsselungsverfahren benötigt. In unten stehender Abbildung soll Nachricht A digital signiert werden. Dazu wird die Nachricht A zunächst mit dem Hash-Algorithmus gehasht und anschließen mit dem privaten Key von A verschlüsselt. Für die heute meist verwendeten X509 Zertifikate werden meist MD5 und SHA-1 als Hash-Funktion und RSA als asymmetrisches Verschlüsselungsverfahren verwendet (MD5withRSA, SHA1withRSA, etc.)

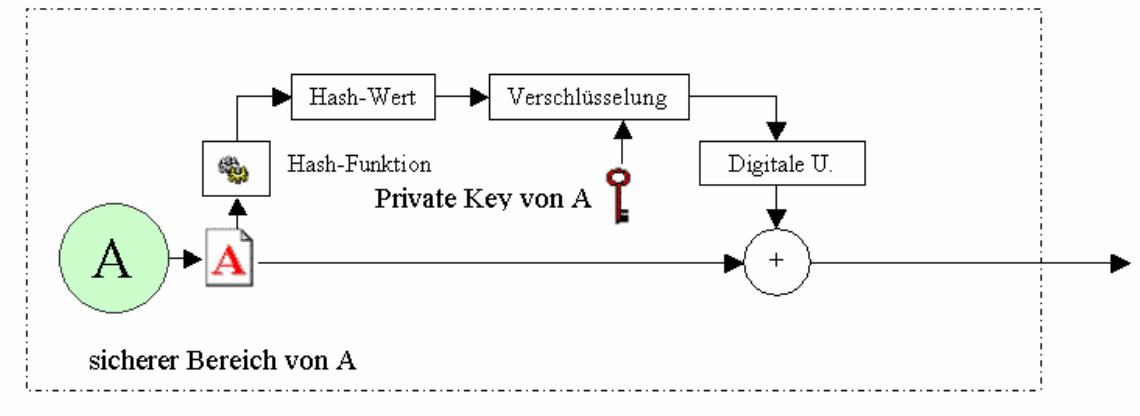

Abb. 1 Erstellen einer digitalen Unterschrift[2]

[2] http://www.ssw.uni-linz.ac.at/Teaching/Lectures/Sem/2000/Lehner/

Um die digitale Unterschrift zu verifizieren, hasht der Kommunikationspartner (in einem typischen Szenario einer https Verbindung somit der Client) die übermittelte Nachricht A, entschlüsselt die digitale Unterschrift mit dem Public Key von A und vergleicht beide Ergebnisse. Stimmen diese überein, kann die digitale Unterschrift als gültig verifiziert werden.

In einem X509 v3 Zertifikat sind neben Gültigkeitsdauer, Algorithmen ID, Seriennummer, Version, Aussteller etc. der Public Key des Zertifizierten, ein eindeutiger Name der Zertifizierungstelle und deren digitale Unterschrift enthalten. Diese digitale Unterschrift kann, sofern es sich nicht um die Root CA handelt, ebenfalls bei einer anderen CA überprüft werden. Für gewöhnlich sind die Zertifikate großer Zertifizierungsstellen wie Versign Inc. bereits in den entsprechenden Applikationen installiert, sodass diese nicht abgefragt werden müssen.

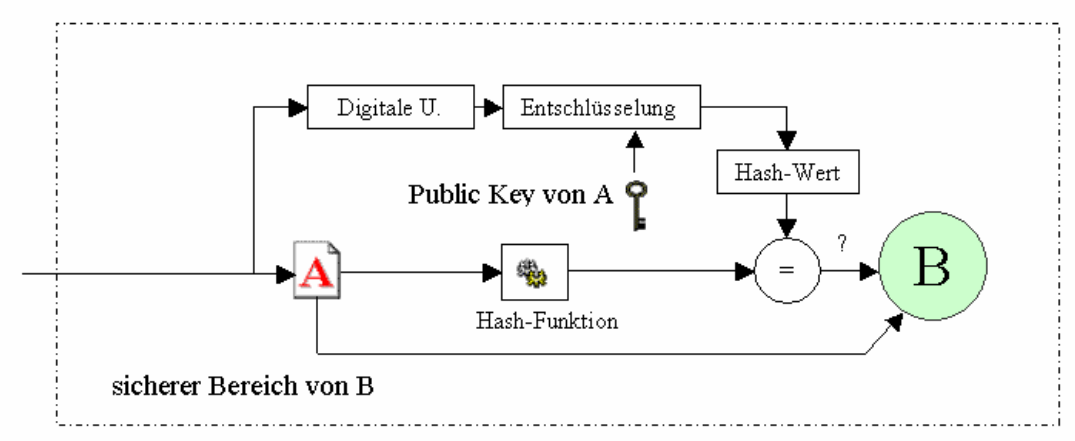

Abb. 2 Verifizieren einer digitalen Unterschrift

Angriffe auf Zertifikate

Bestehende Zertifikate können nur mit einer Second Preimage Attacke angegriffen werden. Wie sich in weiterer Folge herausstellen wird, gibt es für Second Preimage Attacken bereits schnellere als Brute Force [KELSEY], die Komplexität liegt jedoch (noch) in einem Bereich, sodass durch den noch immer sehr hohen Rechenaufwand Angriffe auf bestehende Zertifikate kein akut drohendes Szenario sind.

Ein Angriff ist jedoch auch mit einem Kollisionsangriff möglich, und zwar dann, wenn der Angreifer die Möglichkeit hat, sowohl das Original als auch die Fälschung des Zertifikats zu erstellen, der Angreifer also das Original von einer Certificate Authority für einen „vertrauenswürdigen Server“ signieren lässt und eine Fälschung mit gleichem Hash-value erzeugt hat. Da die Certification Authories (CA) für gewöhnlich nur den Hash-Wert eines Zertifikates signieren, kann somit ein zweites, gefälschtes Zertifikat erstellt werden. Der Certification Authory wird somit quasi ein gefälschtes Dokument zum Schreiben unterbreitet – es signiert den Hash-Wert des gefälschten Zertifikats ebenfalls als richtig.

Statt das beglaubigte Zertifikat auf dem Server zu installieren, kann der Angreifer jetzt ohne Probleme das gefälschte Zertifikat installieren.

Beim Aufruf einer Ressource, die mit diesem Zertifikat zertifiziert ist, wird das gefälschte Zertifikat als gültig ausgewiesen. Der Anwender erhält beim Aufruf des (gefälschten) Servers auch keine Fehlermeldung bzw. kann selbst der sicherheitsbewusste Anwender den Angriff nicht erkennen.

Auf diese Weise können beispielsweise Phishing Attacken oder auch Man-in-the-Middle-Attacken auf HTTPS Verbindungen durchgeführt werden. Eine ähnliche Vorgehensweise ist für alle mittels Zertifikaten gesicherten Verbindungen wie SSL/TLS Verbindungen oder für E-Mail via S/MIME möglich.

MD5

Der MD5 Algorithmus wurde als verbesserte Variante des MD4 von Prof. Ronald L. Rivest (Mitentwickler des RSA Algorithmus) am MIT entwickelt. Bei der Entwicklung von MD4 wurde besonderer Wert auf eine hohe Performance gelegt, sodass bald absehbar war, dass dieser Algorithmus gebrochen werden würde können. Im April 1992 stellte Rivest mit dem RFC 1321 MD5 vor [RIVEST].

Im Gegensatz zu MD4 wurden bei MD5 sechs Veränderungen vollzogen, wobei die Einführung einer zusätzlichen, vierten Runde als primär für ein geringfügig schlechteres Laufzeitverhalten verantwortlich gezeichnet werden kann.

MD5 – RFC 1321

MD5 ist darauf ausgelegt, auf 32 bit Rechnern möglichst effektiv zu laufen. Zum Starten des Algorithmus müssen vier 32 bit lange Wörter übergeben werden, der 128 bit lange Ergebnis Hash wird dann im Grunde aus diesen vier 32 bit Wörtern, über die der Algorithmus iteriert, zusammengesetzt.

Im ersten Schritt wird die b lange Nachricht mit bits aufgefüllt (padding). Von Position m_0 bis Position m_{b-1} steht die eigentliche Nachricht, an Stelle m_b folgt eine 1. Die folgenden Stellen werden mit Nullen aufgefüllt, bis die Länge der gesamten Nachricht mod 512 = 448 ist.

Im zweiten Schritt werden die folgenden 64 bit mit der Angabe der Länge der ursprünglichen Nachricht (b) befüllt. Die Länge der resultierenden Nachricht mod 512 ergibt 0. Die resultierende Nachricht M kann in N 32 bit lange Wörter aufgeteilt werden, wobei N mod 16 gleich 0 ist. Dies ist notwendig, da später 512bit Blöcke in 32 bit Wörter aufgeteilt werden und diese in 16 Runden abgearbeitet werden.

Im dritten Schritt werden vier 32 bit Wörter (A, B, C, D) mit den beim Starten übergebenen Werten initialisiert (IV, Init Values).

Für Schritt vier müssen zunächst die benötigten Operationen und vier Hilfsfunktionen definiert werden:

- + als Addition von Wörtern
- X <<< s einen linken, rotierenden Shift von X um s Positionen

- not(X) eine bitweise Negierung von X
- X v Y ein bitweises OR von X und Y
- X xor Y ein bitweises XOR von X und Y
- XY ein bitweises AND von X und Y

Die darauf aufbauenden Funktionen:

- F(X, Y, Z) = XY v not(X) Z
- G(X, Y, Z) = XZ v Y not (Z)
- H(X, Y, Z) = X xor Y xor Z
- I(X, Y, Z) = Y xor (X v not(Z))

Des Weiteren wird eine 64 Element lange Liste verwendet, die mit einer Sinusfunktion erzeugt wird, wobei T[i] der Integerwert von *4294967296 * |sin(i)|* ist und i in Radiant gewertet wird.

Damit kann die eigentliche Berechnung des Algorithmus beginnen (Pseudocode siehe Appendix A). Vereinfacht gesagt, wird die in den Schritten eins bis drei erzeugte Nachricht in 512 bit große Teile M_0 bis $M_{N/16-1}$ herunter gebrochen, wobei N*32 die Länge der Nachricht repräsentiert. Die Menge aller M werden in der ersten Schleife mit dem Zähler i durchlaufen bzw. abgearbeitet. Innerhalb der ersten Schleife werden die Worte A, B, C und D aus den vorigen Runden bzw. mit den Init Values in die Variablen AA, BB, CC und DD kopiert, die folgenden Runden arbeiten also immer mit den Werten aus den vorherigen Runden.

Danach wird das 512 bit lange M_i in 16 Stück, 32 bit lange Wörter X_0 bis X_{15} kopiert. In 4 Runden werden jeweils 16 Operationen auf A, B, C und D ausgeführt. Am Ende jedes Schleifendruchlaufs über i (nachdem die 4 * 16 Operationen ausgeführt wurden) werden A, B, C und D mit ihren Kopien (AA, BB; etc.) addiert (A = A + AA) und der nächste Schleifendurchlauf gestartet.

Nachdem die ganze Nachricht abgearbeitet wurde, werden die vier Wörter als Output ausgegeben, wobei dem letzten bit von A (least significant bit) begonnen und mit dem ersten bit von D (most significant bit) geendet wird.

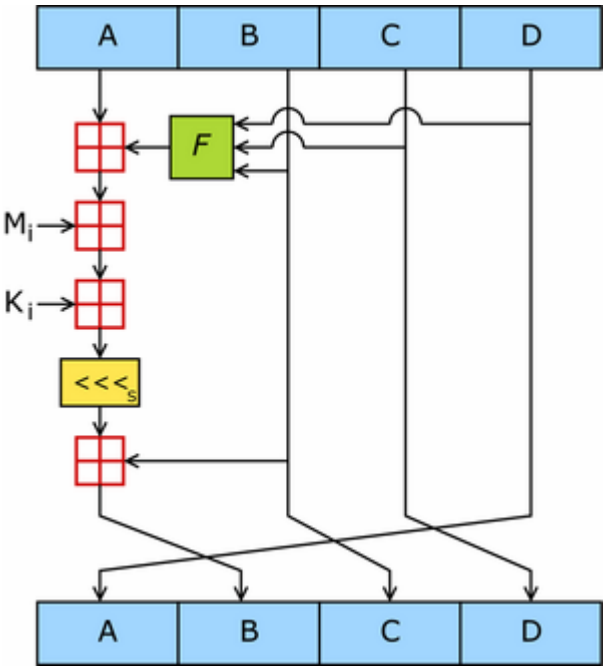

Abb. 3: Funktionsweise von MD5[3],

Angriffe auf MD5

1993, [BOER]

Bereits 1993 fanden Boer und Bosselaers [BOER] eine Möglichkeit Pseudokollision in MD5 zu erzeugen, wobei Kollision und Pseudokollision nach [DOB] wie folgend definiert werden:

„Using the term „collision of a compress function" we assume that the initial value is he same for both inputs, i.e. an initial value IV an two different inputs X *and* $\tilde{X}$ *are given such that*

$$compress(IV; X) = compress(IV, \tilde{X})$$

On the other hand we use the term „pseudo-collision" if two different initial values IV *and* $I\tilde{V}$ *and (possibly identical) input* X *,* $\tilde{X}$ *are given such that*

$$compress(IV; X) = compress(I\tilde{V}, \tilde{X})$$ "

(Bei einer „Kollision" wird davon ausgegangen, dass sich die Initialisierungs-Werte für die Hash Funktion nicht unterscheiden, sodass

$$compress(IV; X) = compress(IV, \tilde{X})$$

Auf der anderen Seite unterscheiden sich bei der „Pseudo-Kollision" die Initialisierungs-Werte, sodass

[3] http://en.wikipedia.org/wiki/MD5

$$compress(IV; X) = compress(\tilde{IV}, \tilde{X})$$

[BOER] geht vor allem mit jenen zwei Änderungen von MD4 auf MD5, die für eine Härtung des Algorithmus hätten sorgen sollen, hart ins Gericht.

Die additive Konstante für Schritt k enthält die ersten 32 bits des Absolutwerts von sin(k). Dies in Kombination mit dem folgenden Verhältnis von vier aufeinander folgenden Sinuswerten

$$(\sin(k) + \sin(k+2))\sin(k+2) = (\sin(k+1) + \sin(k+3))\sin(k+1)$$

ergibt eine annähernd gleiche Relation zwischen 4 frei gewählten aufeinander folgenden additiven Konstanten. Dies hätte leicht vermieden werden können, wenn die nächsten 32 bit der binären Expansion der Sinuswerte verwendet worden wären. Die zweite Änderung, das Ergebnis der Vorrunde mit dem der aktuellen Runde zu addieren, erlaubt es, Kollisionen für die Kompressionsfunktion von MD5 herbeizuführen.

1996, [DOB]

Im Mai 1996, veröffentlichte Dobbertin eine Kollision für MD5, die auch für einen gemeinsamen Init Value funktioniert [DOB]. Die Kollision aus diesem Paper erwartet für Init Value IV und X folgende Werte:

IV = 0x12AC2375 0x3B341042 0x5F62B97C 0x4BA763ED

X_0 = 0xAA1DDA5E	X_4 = 0x1006363E	X_8 = 0x96A1FB19	X_{12} = 0x1326ED65
X_1 = 0xD98ABFF5	X_5 = 0x7216209D	X_9 = 0x1FAE44B0	X_{13} = 0xD93E0972
X_2 = 0x55F0E1C1	X_6 = 0xE01C135D	X_{10} = 0x236BB992	X_{14} = 0xD458C868
X_3 = 0x32774244	X_7 = 0x9DA64D0E	X_{11} = 0x6B7A669B	X_{15} = 0x6B72735A

Der zweite X Wert $\tilde{X}_i = X_i (i < 16, i \neq 14)$ und $\tilde{X}_{14} = X_{14} + 2^9$ ergeben eine Kollision, also

$$MD5 - compress(IV; X) = MD5 - compress(IV, \tilde{X})$$

sodass das Ergebnis beider (IV, X)

0xBF90E670 0x752AF92B 0x9CE4E3E1 0xB12CF8DE

ergibt.

2004, [WANG]

Im August 2004 fanden [WANG] et. al. einen Angriff, der auch mit dem originalen MD5 Init Value eine Kollision zweier 1024 bit Nachrichten erzeugen kann. Der Angriff ist außerdem unabhängig vom Init Value.

Die erste Nachricht wird aus zwei Teilen M und N zusammengesetzt, die zweite Nachricht aus M' und N', wobei M, N, M' und N' folgenden Regeln genügen müssen:

$$M' = M + \Delta C_1, \Delta C_1 = (0,0,0,0,2^{31},0,...,2^{15},...,2^{31},0)$$

$$N'_i = N_i + \Delta C_2, \Delta C_2 = (0,0,0,0,2^{31},0,...,-2^{15},...,2^{31},0)$$

(die nicht-0er auf Position 4, 11 und 14)

Mit dem MD5 Init Value

```
IV = 0x67452301 0xefcdab89 0x98badcfe 0x10325476
```

ist

$$compress(IV,(M,N_i)) = compress(IV,(M',N'_i))$$

Auf einem IBM P690 (32 PowerP4+ CPUs, ca. 5,5Mio$[4]) benötigt es ca. eine Stunde, um so ein M und M' zu finden, danach nur ca. 15 Sekunden bis zu 5 Minuten, um das passende N_i und N'_i zu finden. In Appendix B findet sich eine Tabelle mit Beispieldaten aus dem Paper von [WANG].

2005, [KLIMA]

Der Algorithmus des Angriffs nach [WANG] wurde von den Autoren geheim gehalten, es gibt jedoch mehrere ([KLIMA], [HAWKES]) Versuche, diesen nachzubilden. Besonders sei hier [KLIMA] erwähnt, der das Verfahren von [WANG] zwar nicht nachbilden konnte, dafür allerdings (aufbauend auf den in [WANG] definierten Bedingungen für M, N, M' und N') ein anderes Verfahren nachbilden konnte, das noch schneller arbeitet (ca. 1-8 Stunden auf einem durchschnittlichen PC). Details über den Algorithmus wurden allerdings auch von [KLIMA] nicht veröffentlicht.

[4] http://www-1.ibm.com/servers/eserver/pseries/hardware/highend/index_p4.html, 23.05.2005

SHA

SHA (Secure Hash Algorithmus) ist eine „Familie“ von Hash Algorithmen die als Nachfolger für MD5 konzipiert sind. SHA, oft auch als SHA-0 bezeichnet, wurde 1993 als Secure Hash Standard, FIPS PUB 180 vom NIST (National Institute of Standards and Technology) publiziert und lehnt sich im Wesentlichen an den MD4 an. Die auf MD4 bzw. MD5 geführten Angriffe konnten aufgrund der Änderungen/Erweiterungen nicht auf SHA übertragen werden [CHAB]. Kurz nach der Publikation wurde der Standard aufgrund von kryptographischen Schwächen von der NSA zurückgezogen und 1995 durch den FIPS PUB 180-1 ersetzt, bekannter als SHA-1. SHA-1 mit einem erzeugten Hash-Wert mit 160 bit Länge wird in zahlreichen, sicherheitsrelevanten Anwendungen eingesetzt, unter anderem SSL, TLS, PGP, SSH, S/MIME, IPSec, etc.

Inzwischen wurden SHA-224, SHA-256, SHA-384 und SHA-512 (FIPS PUB 180-2, SHA-2) veröffentlicht, die sich abgesehen von der Outputlänge auch in leicht unterschiedlichen Algorithmen unterscheiden.

SHA-1 – RFC 3174

Diese Beschreibung des SHA-1 bezieht sich auf den RFC 3174 [NWG].

Im ersten Schritt muss die bestehende Nachricht auf eine Länge, die ein Vielfaches von 512 ist, gebracht werden (padding). So wie bei MD5 werden nach einem 1er bit lauter 0er bis auf eine Länge von mod512 = 446 aufgefüllt, in die letzten 64 bit wird die Länge der Original Nachricht geschrieben.

Es werden vier Funktionen f in Abhängigkeit von t definiert:

- f(t;B, C, D) = (B AND C) OR ((NOT B) AND D) für 0 <= t <= 19
- f(t;B, C, D) = B XOR C XOR D für 20 <= t <= 39
- f(t;B, C, D) = (B AND C) OR (B AND D) OR (C AND D) für 40 <= t <= 59
- f(t; B, C, D) = B XOR C XOR D für 60 <= t <= 79

Des Weiteren vier konstante Wörter K (vier * 8bit = 32 bit) in Abhängigkeit von t:

- K(t) = 5A827999 für 0 <= t <= 19
- K(t) = 6ED9EBA1 für 20 <= t <= 39
- K(t) = 8F1BBCDC für 40 <= t <= 59
- K(t) = Ca62C1D6 für 60 <= t <= 79

In [NWG] werden zwei Methoden zur Berechnung eines SHA-1 Hashes, die das gleiche Ergebnis liefern, beschrieben, wobei der Zweite mit einem 5 Wörter großen Register weniger auskommt, dafür die Berechung in Schritt c aufwändiger ist. Der Einfachheit halber sei hier nur die erste Variante beschrieben.

Diese Methode verwendet zwei fünf Wörter lange Register, wobei die Wörter des ersten Registers mit A, B, C, D und F, benannt werden, die Wörter des zweiten Registers mit H_0, H_1, H_2, H_3 und H_4. Eine 80 Wörter lange Sequenz wird mit W_0, W_1, … W_{79} bezeichnet, die jede Runde auf Basis der Blöcke M_i (W_0 bis W_{15}) befüllt bzw. expandiert werden (siehe Schritt b). Zusätzlich gibt es noch einen ein Wort langer Buffer mit TEMP.

Als M_1, M_2 … M_n werden die 16 Wörter langen Nachrichten Blocks bezeichnet, für jeden Block M_i müssen 80 Schritte abgearbeitet werden. Vor dem ersten Schritt werden die Hs wie folgt initialisiert:

H_0 = 67452301, H_1 = EFCDAB89, H_2 = 98BADCFE, H_3 = 10325476, H_4 = C3D2E1F0

Im Folgenden werden in n Runden die Nachrichten Blocks abgearbeitet, wobei die Schritte a und b zur Vorbereitung bzw. Expansion dienen (befüllen der Worte W_0 bis W_{79}), in Runde c die Worte H_0 bis H_5 in A bis E zwischengespeichert bzw. gesichert werden. In Runde d wird für die 80 Wörter W_0 bis W_{79} mit Hilfe des einen Worts langen Zwischenspeichers TMEP die eigentliche Kompressionsfunktion durchgeführt. In Runde e werden die Ergebnisse dieses Blocks mit denen der vorigen Runden verbunden. Abb.1 zeigt die Architektur graphisch.

a) Block M_i wird in 16 Wörter W_0, W_1, … W_{15}

b) Für t = 16 bis 79 sei

   $W_t = W_{t-3} <<< 1$ XOR W_{t-8} XOR W_{t-14} XOR W_{t-16}

c) Sei $A = H_0$, $B = H_1$, $C = H_2$, $D = H_3$, $E = H_4$

d) Für t = 0 bis 79 sei

   $TEMP = A <<< 5 + f(t; B, C, D) + E + W(t) + K(t)$

   $E = D$; $D = C$; $C = B <<< 30$; $B = A$; $A = TEMP$

e) Sei $H_0 = H_0 + A$, $H_1 = H_1 + B$, $H_2 = H_2 + C$, $H_3 = H_3 + D$, $H_4 = H_4 + E$

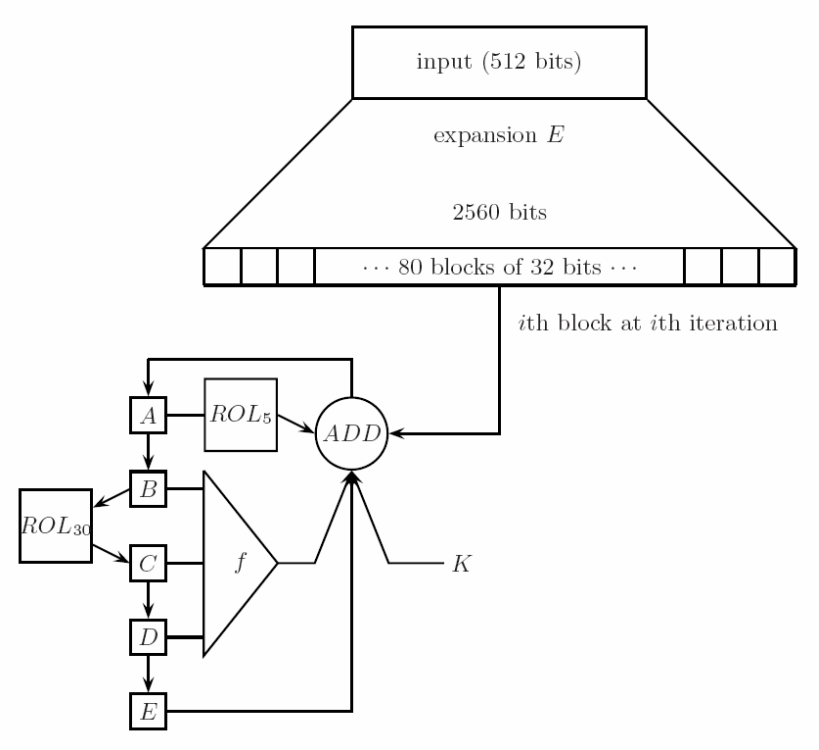

Abb. 4: [CHAB]: SHA Architektur

Nachdem M(n) bearbeitet wurde, steht der Hash Wert in den Wörtern H_0 bis H_4 (in der Graphik als A bis E bezeichnet).

SHA – FIPS PUB 180

Eine Änderung von SHA auf SHA-1 wurde nur in Schritt b – Expansion der 512 bit langen Blöcke M_i (W_0 bis W_{15}) auf 80 Wörter (bis W_{79}) - vollzogen. Die Änderung bringt einen Linksshit des ersten Wortes:

in SHA ist

$$W_t = W_{t-3} \text{ XOR } W_{t-8} \text{ XOR } W_{t-14} \text{ XOR } W_{t-16}$$

in SHA-1 ist

$$W_t = W_{t-3} <<< 1 \text{ XOR } W_{t-8} \text{ XOR } W_{t-14} \text{ XOR } W_{t-16}$$

Angriffe aus SHA

Im Februar 2005 publizierte [WANG2] eine kurze Notiz über eine Reduktion der Komplexität für Kollisionsattacken auf SHA-1 von 2^{80} auf 2^{69}. Technische Details wurden hier zwar ausgespart (außer einer Kollision für SHA-1, siehe Appendix D) bzw. für später angekündigt. Nach Ihren Angaben hatten sie es geschafft, SHA-1 von 80 auf 58 Runden zu reduzieren und waren auf eine Kollision nach weniger als 2^{33} Hash-Operationen gestoßen. Allerdings wurde darauf hingewiesen, dass der Angriff im Wesentlichen auf den Angriffen auf SHA-0 und MD5 basiert. In diesem Sinne seinen hier die Angriffe auf SHA-0 beschrieben.

1998, [CHAB]

Florent Chabaud und Antoine Joux präsentierten auf der CRYPTO 98 einen Angriff auf SHA-1, der die Komplexität eines Kollisionsangriffes auf 2^{61} (von 2^{80}) reduziert. Eine Reduzierung der Komplexität eines Angriffs auf den bereits 1998 veröffentlichten SHA-1 ist mit dieser Methode nicht möglich.

Das Paper beschreibt auf detaillierte Art und Weise, ausgehend von einer auf lineare Funktionen (die f(t;B, C, D) Funktion und das ADD am Ende jeder Schleife über M werden durch XOR ersetzt) reduzierten SHA Funktion (SHI1), welche Schwächen in SHA-0 gefunden wurden und wie diese genutzt werden können. Ausgehend von der reduzierten Funktion SHI1 wird die Funktion bitweise analysiert und die anderen Pseudo-Funktionen SHI2 (originale f Funktionen, aber ADD durch XOR ersetzt) und SHI3 (mit ADD, aber die f Funktionen durch XOR ersetzt) analysiert und schließlich wieder zur ganzen SHA-0 Funktion zusammengesetzt.

Die Methode, die in diesem Paper verwendet wird, ist vergleichbar mit differentieller Kryptoanalyse für Blockchiffren. Im Wesentlichen wird untersucht, wie sich Änderungen im Input auf den Output auswirken, welche bits unter welchen Bedingungen erhalten bleiben etc.

2004 [BIHAM], [JOUX] & [WANG]

Eli [BIHAM] and Rafi Chen fanden Beinahe Kollisionen (near-collisons) für SHA-0: zwei Nachrichten, die einen sehr ähnlichen Hash-Wert haben (142 der 160 bit sind gleich). Außerdem fanden sie (volle) Kollisionen, die sich mit nur 65 der 80 Runden (SHA, Schritt d) berechnen ließen.

Sie nutzen den nach ihrem Paper „*sehr überraschenden Umstand*“, dass die Nachrichten viele neutrale bits haben, die sich über 15 bis 20 Runden nicht auswirken. Außerdem zeigen sie, dass eine Variante des SHA-0 mit mehr Runden deutlich schwächer ist als der originale, was zeigt, dass die Stärke von SHA-0 nicht in direktem Zusammenhang mit der Anzahl der Runden steht.

[JOUX] zeigt in seinem Paper, dass in iterierende Hash-Funktionen (z.B. MD5, SHA) Multikollisionen nicht sonderlich schwerer zu finden sind als einfache Kollisionen. Außerdem wird gezeigt, dass der Aufwand für das Finden von Kollisionen für größere Nachrichten approximativ mit dem Logarithmus der Anzahl der Blöcke (z.B. für MD5 und SHA 512bit) relativ langsam wächst. Die in diesem Paper präsentierten Angriffe ermöglicht Attacken auf die wichtigsten Eigenschaften von Hash-Funktionen: kollisionsfreiheit, Resistenz gegen Preimage und Second Preimage Attacken.

Aufbauend auf dem Paper von Biham und Joux veröffentlichten am 12. August 2004 Joux, Carribault, Lemuet und Jalby eine Kollision in SHA-0. Für den Angriff der Komplexität 2^{51} arbeitete ein mit 256 Intel-Itanium2 Prozessoren ausgerüsteter Supercomputer ca. 80.000 CPU Stunden.

Auf der CRYPTO 2004 erwähnte [WANG] neben den Angriffen auf MD4, MD5, HAVAK-128 und RIPEMD, dass er in der Lage sei, für SHA-0 Kollisionen mit einem Aufwand von 2^{40} herbeizuführen.

Im Februar 2005 präsentierte [WANG2] in dem Aufsehen erregenden Paper eine Kollision für SHA0, die mit einem Aufwand von weniger als 2^{39} herbeigeführt wurde. Nähere Erläuterungen des Angriffs oder technische Details wurden allerdings auch hier ausgespart.

2004 [KELSEY] Second Preimages Attack

John [KELSEY] und Bruce Schneier fanden 2004 einen Angriff auf n-bit iterative Hashfunktionen mit n-bit langen Zwischenwerten (z.B. die Worte A-D für MD5 bzw. A-E für SHA-1). Für einen 2^k bit langen Nachrichtenblock braucht ihre Methode einen Aufwan von $k*2^{n/2+1}+2^{n-k+1}$. Für Angriffe auf die Familie der SHA – Funktionen benötigt ihre Methode sogar nur $3*2^{n/2+1}+2^{n-k+1}$, also für eine 2^{60} lange Nachricht mit SHA-1 gehasht ergibt sich somit ein Aufwand von 2^{106}.

Literaturverzeichnis

[BIHAM] Eli Biham, Rafi Chen: Near-Collisions for SHA-0, 2004
[BOER] Bert den Boer, Antoon Bosselaers: Collisions for the compression function
[CHAB] Florent Chabaud, Antoine Joux: Differential Collision in SHA-0, 1998
[DOB] H. Dobbertin: Cryptoanalysis of MD5 Compress, May 1996
of MD5, Juli 1993
[HAWKES] Philip Hawkes, Michael Paddon, Gregory G. Rose: Musings on the Wang et al. MD5 Collision, Oktober 2004
[JOUX] Antoine Joux: Multicollisions in Iterated Hash Functions. Application to Cascaded Construcations, 2004
[KELSEY] John Kelsey (NIST), Bruce Schneier: Second Preimages on n-bit Hash Functions for much less than 2^n work, 2004
[KLIMA] Vlastimil Klíma: Finding MD5 Collisions – a Toy For a Notebook, 5. März 2005
[NWG] Network Working Group: US Secure Hash Algorithm 1 (SHA1), RFC 3174, September 2001
[RIVEST] R. Rivest: The MD5 Message-Digest Algorithm, RFC 1321, April 1992
[SCHNEIER] Bruce Schneier: A self-study course in block-cipher crytoanalysis
[WANG] Xiaoyun Wang et.al.: Collisions for Hash Functions, August 2004
[WANG2] Xiaoyun Wang, Yiqun Lisa Yin, Hongbo Yu: Collision Search Attacks on SHA1, Februar 2005

Appendix A: Pseudocode MD5

```
/* Process each 16-word block. */
For i = 0 to N/16-1 do

     /* Copy block i into X. */
     For j = 0 to 15 do
       Set X[j] to M[i*16+j].
     end /* of loop on j */

     /* Save A as AA, B as BB, C as CC, and D as DD. */
     AA = A
     BB = B
     CC = C
     DD = D

     /* Round 1. */
     /* Let [abcd k s i] denote the operation
          a = b + ((a + F(b,c,d) + X[k] + T[i]) <<< s). */
     /* Do the following 16 operations. */
     [ABCD  0  7  1]  [DABC  1 12  2]  [CDAB  2 17  3]  [BCDA  3 22  4]
     [ABCD  4  7  5]  [DABC  5 12  6]  [CDAB  6 17  7]  [BCDA  7 22  8]
     [ABCD  8  7  9]  [DABC  9 12 10]  [CDAB 10 17 11]  [BCDA 11 22 12]
     [ABCD 12  7 13]  [DABC 13 12 14]  [CDAB 14 17 15]  [BCDA 15 22 16]

     /* Round 2. */
     /* Let [abcd k s i] denote the operation
          a = b + ((a + G(b,c,d) + X[k] + T[i]) <<< s). */
     /* Do the following 16 operations. */
     [ABCD  1  5 17]  [DABC  6  9 18]  [CDAB 11 14 19]  [BCDA  0 20 20]
     [ABCD  5  5 21]  [DABC 10  9 22]  [CDAB 15 14 23]  [BCDA  4 20 24]
     [ABCD  9  5 25]  [DABC 14  9 26]  [CDAB  3 14 27]  [BCDA  8 20 28]
     [ABCD 13  5 29]  [DABC  2  9 30]  [CDAB  7 14 31]  [BCDA 12 20 32]

     /* Round 3. */
     /* Let [abcd k s t] denote the operation
          a = b + ((a + H(b,c,d) + X[k] + T[i]) <<< s). */
     /* Do the following 16 operations. */
     [ABCD  5  4 33]  [DABC  8 11 34]  [CDAB 11 16 35]  [BCDA 14 23 36]
     [ABCD  1  4 37]  [DABC  4 11 38]  [CDAB  7 16 39]  [BCDA 10 23 40]
     [ABCD 13  4 41]  [DABC  0 11 42]  [CDAB  3 16 43]  [BCDA  6 23 44]
     [ABCD  9  4 45]  [DABC 12 11 46]  [CDAB 15 16 47]  [BCDA  2 23 48]

     /* Round 4. */
     /* Let [abcd k s t] denote the operation
          a = b + ((a + I(b,c,d) + X[k] + T[i]) <<< s). */
     /* Do the following 16 operations. */
     [ABCD  0  6 49]  [DABC  7 10 50]  [CDAB 14 15 51]  [BCDA  5 21 52]
     [ABCD 12  6 53]  [DABC  3 10 54]  [CDAB 10 15 55]  [BCDA  1 21 56]
     [ABCD  8  6 57]  [DABC 15 10 58]  [CDAB  6 15 59]  [BCDA 13 21 60]
     [ABCD  4  6 61]  [DABC 11 10 62]  [CDAB  2 15 63]  [BCDA  9 21 64]
```

```
    /* Then perform the following additions. (That is increment each
       of the four registers by the value it had before this block
       was started.) */
    A = A + AA
    B = B + BB
    C = C + CC
    D = D + DD

end /* of loop on i */
```

Appendix B: Beispieldaten für eine MD5 Kollision nach [WANG]

X_1	M	2dd31d1 c4eee6c5 69a3d69 5cf9af98 87b5ca2f ab7e4612 3e580440 897ffbb8 634ad55 2b3f409 8388e483 5a417125 e8255108 9fc9cdf7 f2bd1dd9 5b3c3780
	N_1	2dd31d1 c4eee6c5 69a3d69 5cf9af98 87b5ca2f ab7e4612 3e580440 897ffbb8 634ad55 2b3f409 8388e483 5a417125 e8255108 9fc9cdf7 f2bd1dd9 5b3c3780
X_1	M´	2dd31d1 c4eee6c5 69a3d69 5cf9af98 87b5ca2f ab7e4612 3e580440 897ffbb8 634ad55 2b3f409 8388e483 5a417125 e8255108 9fc9cdf7 f2bd1dd9 5b3c3780
	N_1´	d11d0b96 9c7b41dc f497d8e4 d555655a c79a7335 cfdebf0 66f12930 8fb109d1 797f2775 eb5cd530 baade822 5c15cc79 ddcb74ed 6dd3c55f d80a9bb1 e3a7cc35
H		9603161f f41fc7ef 9f65ffbc a30f9dbf
X_2	M	2dd31d1 c4eee6c5 69a3d69 5cf9af98 7b5ca2f ab7e4612 3e580440 897ffbb8 634ad55 2b3f409 8388e483 5a41f125 e8255108 9fc9cdf7 72bd1dd9 5b3c3780
	N_2	313e82d8 5b8f3456 d4ac6dae c619c936 b4e253dd fd03da87 6633902 a0cd48d2 42339fe9 e87e570f 70b654ce 1e0da880 bc2198c6 9383a8b6 2b65f996 702af76f
X_2	M´	2dd31d1 c4eee6c5 69a3d69 5cf9af98 87b5ca2f ab7e4612 3e580440 897ffbb8 634ad55 2b3f409 8388e483 5a417125 e8255108 9fc9cdf7 f2bd1dd9 5b3c3780
	N_2´	313e82d8 5b8f3456 d4ac6dae c619c936 34e253dd fd03da87 6633902 a0cd48d2 42339fe9 e87e570f 70b654ce 1e0d2880 bc2198c6 9383a8b6 ab65f996 702af76f
H		8d5e7019 6324c015 715d6b58 61804e08

Appendix C: Beispieldaten für eine SHA-0 Kollision nach [WANG2]

h_1 = compress(h_0, M_0)

h_2= compress(h_1, M_1')= compress(h_1, M_1')

h_0	67452301 efcdab89 98badcfe 10325476 c3d2e1f0
M_0	65c24f5c 0c0f89f6 d478de77 ef255245 83ae3a1f 2a96e508 2c52666a 0d6fad5a 9d9f90d9 eb82281e 218239eb 34e1fbc7 5c84d024 f7ad1c2f d41d1a14 3b75dc18
h_1	39f3bd80 c38bf492 fed57468 ed70c750 c521033b
M_1	474204bb 3b30a3ff f17e9b08 3ffa0874 6b26377a 18abdc01 d320eb93 b341ebe9 13480f5c ca5d3aa6 b9f3bd88 21921a2d 4085fca1 eb65e659 51ac570c 54e8aae5
M_1'	c74204f9 3b30a3ff 717e9b4a 3ffa0834 6b26373a 18abdc43 5320eb91 3341ebeb 13480f1c 4a5d3aa6 39f3bdc8 a1921a2f 4085fca3 6b65e619 d1ac570c d4e8aaa5
H_2	2af8aee6 ed1e8411 62c2f3f7 3761d197 0437669d

Appendix D: Bespieldaten für eine SHA-1 Kollision nach [WANG2]

$h_1 = compress(h_0, M_0) = compress(h_0, M_0')$

h_0	67452301 efcdab89 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