Bundesrealgymnasium
Gröhrmühlgasse 27
2700 Wiener Neustadt

Vorwissenschaftliche Arbeit

Titel der Vorwissenschaftlichen Arbeit:

Funktionsweise und Schwachstellen von kryptographischen Hashfunktionen

Verfasser:

Sebastian Hirnschall

Wiener Neustadt, im Februar 2017

Klasse:	8AL
Fachgebiet:	Informatik
Schuljahr:	2016/2017
Betreuer:	Mag. Christian Filipp

Abstract

Vordergründiges Ziel dieser Arbeit ist es, die Funktionsweise von Hashfunktionen zu beleuchten und zu untersuchen, auf welche Schwachstellen sich Angriffe stützen. Obwohl die Verwendung von Hashalgorithmen für den Nutzer meist nicht ersichtlich ist, finden sie dank ihrer nützlichen Eigenschaften in sehr vielen Bereichen der Informatik Anwendung. Zunächst werden die für das Verständnis von Hashfunktionen benötigten Grundlagen kurz angeschnitten und der Unterschied zu kryptographischen Hashfunktionen erläutert. Anschließend wird die Funktionsweise von drei der wichtigsten Hashalgorithmen (MD4, MD5, SHA) untersucht. Die letzten Kapitel befassen sich mit der Frage, inwiefern die hohe Geschwindigkeit moderner Hashalgorithmen eine Schwachstelle darstellt. Dabei werden drei repräsentative Angriffe auf Passwortlisten nachgestellt (Bruteforce, Markow-Kette, gezielter Angriff). Diese unterscheiden sich grundlegend, nützen jedoch alle die enorme Geschwindigkeit der verwendeten Hashfunktion aus.

Inhaltsverzeichnis

1 Einleitung

Hashfunktionen sind ein wichtiger Bestandteil der Kryptographie. Sie haben unzählige Anwendungsgebiete und doch bekommt der Nutzer im Normalfall nichts von ihrer Verwendung mit. In dieser Arbeit möchte ich deshalb die Funktionsweise und Schwachstellen von kryptographischen Hashfunktionen untersuchen und anschließend Aufschluss darüber geben, ob sie für die Sicherung von Passwörtern geeignet sind.
Der erste Teil dieser Arbeit befasst sich vor allem mit den Fragen, was eine kryptographische Hashfunktion ist und welche Arten von Angriffen es gibt. Dabei habe ich mich größtenteils auf Stinsons "Cryptography: theory and practice" (Stinson 1995) und auf Schneiers "Angewandte Kryptographie" (Schneier 1996) gestützt. In beiden Werken wird gut fassbar und auf interessante Art und Weise ein Überblick über Angriffe auf Hashfunktionen gegeben und die Funktionsweise einiger Algorithmen beschrieben.
Darauf aufbauend habe ich im zweiten Teil drei der wichtigsten Hashalgorithmen genau untersucht, untereinander verglichen und beispielhaft vorgerechnet.
Der dritten Teil der Arbeit befasst sich abschließend mit der Frage, ob sich kryptographische Hashfunktionen für die Sicherung von Passwörtern eignen. Um diese Frage zu beantworten, habe ich drei Angriffe nachgestellt und so gut wie möglich optimiert. Die daraus gezogenen Schlüsse finden sich im letzten Abschnitt wieder, in dem Probleme aufgezeigt, Verbesserungsvorschläge gemacht und ein Ausblick auf künftige Entwicklungen auf diesem Gebiet gegeben werden.

Die verwendeten Abbildungen, Grafiken und Statistiken wurden, wenn nicht anderweitig gekennzeichnet, von mir selbst erstellt.

2 Hashfunktionen

> *A hash function H projects a value from a set with many (or even an infinite number of) members to a value from a set with a fixed number of (fewer) members. Hash functions are not reversible.(Weisstein 2007)*

Die Funktion h bildet A auf B ab. Da die Menge B weniger Elemente besitzt als A, handelt es sich um eine nicht injektive[1] Abbildung. Das heißt, nicht jedem Element der Menge B wird genau ein Element der Menge A zugewiesen. Werden mehrere $a \in A$ auf das selbe $b \in B$ abgebildet, spricht man von einer Kollision.

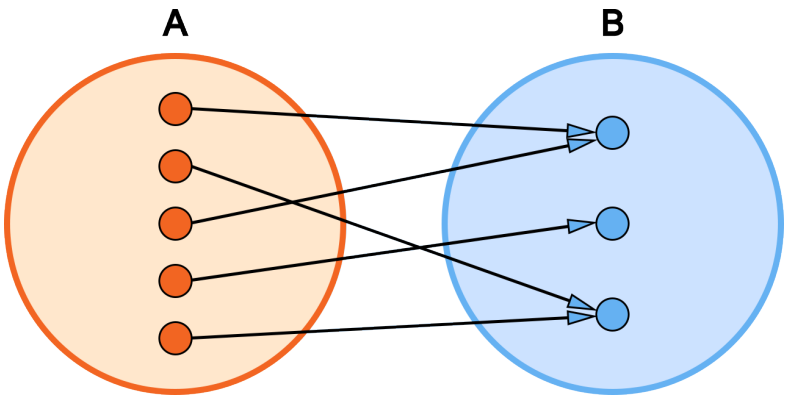

Abbildung 1: Nicht injektive Abbildung

Um die Funktionsweise einer Hashfunktion zu verstehen, betrachte man $h : X \rightarrow Z$.

$$h(x) = z = \lfloor x \mod 16 \rfloor \tag{2.1}$$

In diesem Beispiel werden alle $x \in \mathbb{R}$ auf $\{z \in \mathbb{N} \mid 0 \leq z < 16\}$ abgebildet. Die Länge des Hashwertes in Bit lässt sich mit $L = \log_2 |z|$ berechnen. Da z nur 2^4 verschiedene Werte annehmen kann, hat der Hashwert eine Länge von 4 Bit. Daraus folgt, dass die Berechnung von mehr als 2^4 Funktionswerten zwangsläufig zu einer Kollision führt.

[1] $f(x_1) = f(x_2) => x_1 = x_2$

2.1 Kryptographische Hashfunktionen

Damit die Verwendung von Hashfunktionen die Sicherheit nicht beeinträchtigt, muss h einige Bedingungen erfüllen.
Bei den meisten Angriffen wird versucht eine signierte Nachricht (x, y) zu fälschen, wobei $y = sig_K(h(x))$ ist. Als Erstes errechnet der Angreifer $z = h(x)$, um $x' \neq x$ zu finden, sodass $h(x') = h(x)$ ist. Gelingt es dem Angreifer, handelt es sich bei (x', y) um eine gültig signierte Nachricht, eine Fälschung. Um dieser Art von Angriffen vorzubeugen, ist es notwendig, dass h folgende kollisionsresistente Eigenschaften erfüllt: (vgl. Stinson 1995, S. 233)

Definition 2.1
Eine Hashfunktion h ist schwach kollisionsresistent, wenn es rechnerisch nicht möglich ist, zu einer gegebenen Nachricht x eine Nachricht $x' \neq x$ zu finden, sodass $h(x') = h(x)$ ist. (vgl. ebd., S. 233)

Definition 2.2
Eine Hashfunktion h ist stark kollisionsresistent, wenn es rechnerisch nicht möglich ist, Nachrichten x und x' zu finden, sodass $x' \neq x$ und $h(x) = h(x')$ ist. (vgl. ebd., S. 233)

Das folgende Protokoll, das zuerst von Gideon Yuval beschrieben wurde, zeigt, wie Alice Bob beschwindeln kann, wenn Definition 2.2 nicht erfüllt ist:
(Protokoll wörtlich übernommen. Schneier 1996, S. 491 f.)

1. Alice bereitet zwei Versionen eines Vertrages vor. Eine begünstigt Bob, die andere treibt ihn in den Ruin.

2. Alice bringt an jedem Dokument einige subtile Änderungen an und berechnet für jede den Hashwert. (Beispiele für solche Änderungen: ersetze ein Leerzeichen durch die Sequenz Leerzeichen, Backspace, Leerzeichen; füge vor einem Zeilenende ein oder zwei Leerzeichen ein usw.) Indem sie auf jeder von 32 Zeilen eine Änderung durchführt oder eben nicht, kann Alice leicht 2^{32} verschiedene Dokumente erzeugen.

3. Alice vergleicht die Hashwerte für alle Versionen der beiden Dokumente und sucht übereinstimmende Paare. (Liefert die Hashfunktion nur einen 64-Bit-Wert, findet sie gewöhnlich ein übereinstimmendes Paar innerhalb von 2^{32} Versionen). Sie rekonstruiert die beiden Dokumente, die denselben Hashwert liefern.

4. Alice läßt Bob die Vertragsversion unterzeichnen, die ihn begünstigt. Sie benutzt dazu ein Protokoll, bei dem er nur den Hashwert unterzeichnet.

5. Zu einem späteren Zeitpunkt ersetzt Alice den von Bob unterschriebenen Vertrag mit der Version, die er nicht unterzeichnet hat. Jetzt kann sie einen Richter davon überzeugen, daß Bob diesen Vertrag unterzeichnet hat.

Definition 2.3
Eine Hashfunktion ist eine Einwegfunktion, wenn es rechnerisch nicht möglich ist zu einem gegebenen Hashwert z eine Nachricht x zu finden, sodass $h(x) = z$ ist. (vgl. Stinson 1995, S. 234)

Da man zeigen kann, dass Definition 2.2 Definition 2.3 impliziert, ist es für eine Hashfunktion ausreichend, stark kollisionsresistent zu sein, um als kryptographische Hashfunktion zu gelten.
Man betrachte die Hashfunktion $h : X \to Z$. Bei X und Z handelt es sich um endliche Mengen. X lässt sich als Bitfolge der Länge $\log_2 |X|$ darstellen und Z als Bitfolge der Länge $\log_2 |Z|$. Da die Nachricht x um mindestens ein Bit länger sein soll als der Hashwert $z = h(x)$, gilt $|X| \geq 2|Z|$. Unter der Annahme, es existiere ein Algorithmus A, für den gilt $A(z) \in X$ und $h(A(z)) = z$, ist es möglich, folgendes Theorem zu beweisen. (vgl. ebd., S. 234)

Theorem 2.1
Angenommen $h : X \to Z$ ist eine Hashfunktion, wobei es sich bei X und Z um endliche Mengen handelt, für die $|X| \geq 2|Z|$ gilt und A eine Umkehrfunktion zu h ist, dann existiert ein Las Vegas Algorithmus, der mit Wahrscheinlichkeit von mindestens $1/2$ eine Kollision für h findet. (vgl. ebd., S. 234)

Beweis Man betrachte Algorithmus B. (vgl. ebd., S. 234 f.)

Algorithmus B:

1. wählt ein zufälliges $x \in X$
2. berechnet $z = h(x)$
3. berechnet $x_1 = A(z)$
4. if($x_1 \neq x$)
   x_1 und x kollidieren bei Verwendung von h (success)

Da es sich bei B eindeutig um einen Las Vegas Algorithmus handelt (entweder es wird eine Kollision gefunden oder kein Ergebnis ausgegeben), ist die Erfolgswahrscheinlichkeit zu berechnen. Man definiere

$$[x] \coloneqq \{x_1 \in X : h(x) = h(x_1)\}. \tag{2.2}$$

Da jede Äquivalenzklasse $[x]$ aus inversen Elementen von Z besteht, ist die Zahl der Äquivalenzklassen höchstens $|Z|$. Man bezeichne die Menge aller Äquivalenzklassen als C.
Für ein gegebenes x gibt es $|[x]|$ viele x_1, für die $h(x) = h(x_1)$ gilt. $|[x]|-1$ dieser x_1 unterscheiden sich von x. Die Wahrscheinlichkeit eine Kollision für ein gegebenes $x \in X$ zu finden, lässt sich demnach mit $(|[x]| - 1)/|[x]|$ berechnen.
Die Erfolgswahrscheinlichkeit $p(success)$ von B kann durch das Berechnen des Mittelwertes für alle möglichen x berechnet werden:

$$\begin{aligned} p(success) &= \frac{1}{|X|} \sum_{x \in X} \frac{|[x]| - 1}{|[x]|} \\ &= \frac{1}{|X|} \sum_{c \in C} \sum_{x \in c} \frac{|c| - 1}{|c|} \\ &= \frac{1}{|X|} \sum_{c \in C} |c| - 1 \\ &= \frac{1}{|X|} \left(\sum_{c \in C} |c| - \sum_{c \in C} 1 \right) \\ &\geq \frac{|X| - |Z|}{|X|} \\ &\geq \frac{|X| - |X|/2}{|X|} \\ &= \frac{1}{2}. \end{aligned} \tag{2.3}$$

Der Las Vegas Algorithmus B hat eine Erfolgswahrscheinlichkeit von mindestens $1/2$.

□

2.2 Bit-Operatoren

Um hohe Geschwindigkeiten zu erreichen, verwenden viele Hashalgorithmen Bit-Operatoren. Dabei werden die einzelnen Bit eines Bit-Strings verglichen, um einen neuen Bit-String der gleichen Länge zu bilden.

Bitweise AND-Verknüpfung

Der bitweise AND-Operator, dargestellt durch $\wedge$ oder &, liefert als Ausgabe nur dann 1, wenn beide verglichenen Bits 1 sind. In der Praxis wird der AND-Operator zum gezielten Löschen von Bits verwendet.

Bit a	Bit b	a$\wedge$b
0	0	0
0	1	0
1	0	0
1	1	1

Tabelle 1: Wahrheitstafel - AND

Im folgenden Beispiel wird nun jedes Bit der beiden 8-Bit-Strings verglichen und ein neuer 8-Bit-String gebildet.

$$\begin{aligned} & 1110\ 1101 \\ \& \ & 1001\ 1000 \\ = \ & 1000\ 1000 \end{aligned}$$

Bitweise OR-Verknüpfung

Der bitweise OR-Operator, dargestellt durch $\vee$ oder |, liefert als Ausgabe 1, wenn mindestens eines der beiden verglichenen Bits 1 ist. In der Praxis wird das bitweise OR verwendet, um ein Bit in einem Register auf 1 zu schalten.

Bit a	Bit b	a$\vee$b
0	0	0
0	1	1
1	0	1
1	1	1

Tabelle 2: Wahrheitstafel - OR

Im folgenden Beispiel wird nun jedes Bit der beiden 8-Bit-Strings verglichen und ein neuer 8-Bit-String gebildet.

$$\begin{aligned} & 1110\ 1101 \\ \mid\ & 1001\ 1000 \\ = & \ 1111\ 1101 \end{aligned}$$

Bitweise XOR-Verknüpfung

Der bitweise XOR-Operator, dargestellt durch $\oplus$ oder $\hat{}$, liefert als Ausgabe 1, wenn genau eines der beiden verglichenen Bits 1 ist. In der Praxis wird der XOR-Operator verwendet, um ein Bit in einem Register umzuschalten (engl. to toggle).

Bit a	Bit b	a$\oplus$b
0	0	0
0	1	1
1	0	1
1	1	0

Tabelle 3: Wahrheitstafel - XOR

Im folgenden Beispiel wird nun jedes Bit der beiden 8-Bit-Strings verglichen und ein neuer 8-Bit-String gebildet.

$$\begin{aligned} & 1110\ 1101 \\ \hat{}\ & 1001\ 1000 \\ = & \ 0111\ 0101 \end{aligned}$$

Bitweises Komplement

Das bitweise Komplement $\neg$ oder $\sim$ invertiert jedes einzelne Bit.

Bit a	$\neg$a
0	1
1	0

Tabelle 4: Wahrheitstafel - NOT

Im folgenden Beispiel wird nun jedes Bit des 8-Bit-Strings invertiert.

$$\begin{aligned} \sim & \ 1110\ 1101 \\ = & \ 0001\ 0010 \end{aligned}$$

Linksverschiebung

Mit einer Linksverschiebung, dargestellt durch $\ll n$, werden alle Stellen eines Bit-Strings um n Stellen nach links verschoben. Dabei entstehende Leerstellen (rechts) werden mit Nullen aufgefüllt. Dieser Vorgang kommt einer Multiplikation mit 2^n gleich.
In der Praxis wird eine Linksverschiebung häufig verwendet, um ein Byte mit einem Einser an der gewünschten Stelle zu erhalten.

$$1 = 0b00000001$$
$$(1 \ll 2) = 0b00000100$$

Soll ein Bit-String fester Länge um n Bit linksverschoben werden, werden die höchstwertigen n Bit "abgeschnitten".

$$(10011101 \ll 2) = 01110100$$

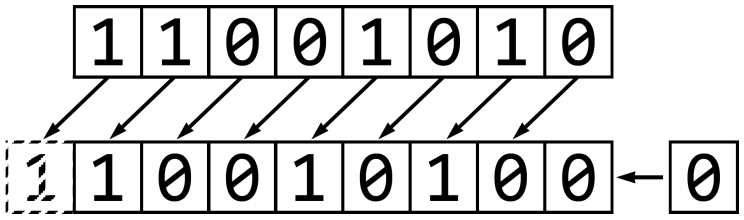

Abbildung 2: Linksverschiebung

Soll das MSB (höchstwertiges Bit) nicht verloren gehen, wird eine zirkuläre Linksverschiebung, dargestellt durch $\lll n$, angewandt. Dabei werden abgeschnittene Stellen hinten angehängt.

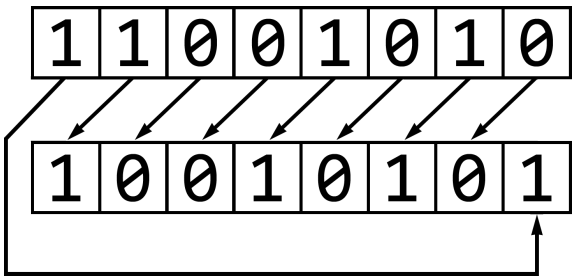

Abbildung 3: Zirkuläre Linksverschiebung

2.3 Angriffe

Ein Hash Algorithmus gilt im kryptographischen Sinne als gebrochen, wenn es eine Methode oder einen Angriff gibt, der schneller eine Kollision herbeiführt als ein Brute Force Angriff.(vgl. Schneier 2000, S. 2)

2.3.1 Preimage-Angriff

Bei einem Preimage(Urbild)-Angriff wird versucht, eine Nachricht x zu finden, die einen dem Angreifer bekannten Hashwert z liefert, zum Beispiel um einen unterzeichneten Vertrag im Nachhinein zu ändern. (vgl. Petritsch 2006, S. 5)

$$h(x) = z$$

2.3.2 Second-Preimage-Angriff

Bei einem Second-Preimage-Angriff ist im Gegensatz zu Abschnitt 2.3.1 nicht nur der Hashwert z sondern auch die ursprüngliche Nachricht x_1, für die gilt $h(x_1) = z$, bekannt. Gesucht ist eine Nachricht $x_2 \neq x_1$, für die

$$h(x_2) = h(x_1)$$

gilt. (vgl. ebd., S. 6 f.)

2.3.3 Kollisionsangriff

Bei einem Kollisionsangriff werden zwei unterschiedliche Nachrichten $x_1 \neq x_2$ gesucht, die die Gleichung

$$h(x_1) = h(x_2)$$

erfüllen.
Ein sehr naiver Ansatz wäre es, ein zufälliges x_1 zu wählen und danach einen second- Preimage-Angriff durchzuführen. Dabei wären bei einem 160 Bit langen Hashwert rund 2^{160} Berechnungen nötig, um mit großer Wahrscheinlichkeit eine Kollision zu finden.
Mit Hilfe eines Geburtstagsangriffes kann mit weit weniger Berechnungen eine gleichgroße Erfolgswahrscheinlichkeit erreicht werden. Dabei werden n zufällige Nachrichten $x_1, \ldots, x_n$ gewählt und für jede der Hashwert $z = h(x_i)$ berechnet. Danach wird überprüft, ob zwei Nachrichten denselben Hashwert ergeben haben.

Analog zum Geburtstagsparadoxon lässt sich die Erfolgswahrscheinlichkeit bei m möglichen Hashwerten, wie folgt, berechnen.

$$p = 1 - \left(\frac{m}{m} * \frac{m-1}{m} \cdots \frac{m-n+1}{m} \right)$$

Um mit Wahrscheinlichkeit $\geq 1/2$ ein Kollision zu finden, sind $n \approx 1.17 * \sqrt{m}$ Berechnungen notwendig. Um mit großer Wahrscheinlichkeit eine Kollision zu finden, sind $\approx 2^{80}$ Berechnungen nötig. (vgl. Petritsch 2006, S. 5 f.)

3 Hashfunktionen im Vergleich

3.1 Message-Digest Algorithm 4 (MD4)

1990 veröffentlichte Ron Rivest MD4. Dieser orientiert sich am Merkel-Damgård-Prinzip, baut jedoch nicht auf anderen Bausteinen der Kryptographie (wie z.B. Blockchiffren) auf. Da MD4 nur einfache Bitmanipulationen mit 32-Bit-Operanden verwendet, eignet sich der Algorithmus für schnelle Softwareimplementierungen. Üblicherweise werden Bit-Strings, welche aus genau 32 Bit bestehen, als Wort bezeichnet. Weiters wird für Wörter x, y, $x + y$ als Addition modulo 2^{32} definiert. Die verwendeten Bit-Operatoren sind in Abschnitt 2.2 beschrieben.(vgl. Greveler 1998, S. 19 f.)
Die Funktionen f, g und h bilden auf Wörter ab und seien folgendermaßen definiert.

$$f(x, y, z) \coloneqq (x \wedge y) \vee ((\neg x) \wedge z) \tag{3.1}$$
$$g(x, y, z) \coloneqq (x \wedge y) \vee (x \wedge z) \vee (y \wedge z) \tag{3.2}$$
$$h(x, y, z) \coloneqq x \oplus y \oplus z \tag{3.3}$$

1. Schritt (1-0-Padding)

Da sich MD4 am Merkel-Damgård-Prinzip orientiert, wird die Nachricht zunächst um einige Bits erweitert. An das Argument der Hashfunktion werden ein Einser und danach, falls nötig, so viele Nullen angehängt, bis die Länge des Bit-Strings kongruent 448 modulo 512 ist. Sollte die Länge der Nachricht bereits vor diesem Schritt kongruent 448 modulo 512 sein, wird sie um 512 Bit erweitert.(vgl. ebd., S. 20)

2. Schritt (Längenpadding)

Anschließend wird die Länge der ursprünglichen Nachricht in Bit konkateniert[2]. Dafür wird ein 64-Bit-String gebildet, um insgesamt eine Kongruenz 0 modulo 512 zu erreichen. Dabei wird der 64-Bit-String in 2 Wörter geteilt und das niederwertigere Wort zuerst angehängt. Sollte die Länge nicht als 64-Bit-String darstellbar sein, wird stattdessen die Kongruenz modulo 2^{64} konkateniert.(vgl. ebd., S. 21)

3. Schritt (Initialisierung)

Der MD4 Hashalgorithmus verwendet einen 4-Wort-Puffer (A, B, C, D). Um die Werte A, B, C und D zwischen Durchläufen zu speichern, wird ein zweiter Puffer (AA, BB, CC, DD) benötigt. Die 32-Bit-Strings werden mit folgenden

[2] Als Konkatenation wird die Verkettung von Zeichen oder Zeichenketten bezeichnet.

Hexadezimal-Werten initialisiert: (vgl. Stinson 1995, S. 247 f.)

$$A = 67452301$$
$$B = efcdab89$$
$$C = 98badcfe$$
$$D = 10325476$$

4. Schritt (Argumentverarbeitung)

Da dank der ersten beiden Schritte sichergestellt ist, dass die Länge der Nachricht x (in Bit) ein ganzzahliges Vielfaches von 512 ist, kann nun der Vektor

$$M = M[0]M[1]\dots M[N-1]$$

so gebildet werden, dass $M[i]$ ein 32-Bit-String ist und dass $N \equiv 0 \mod 16$ ist. Anschließend wird die Nachricht in 512-Bit-Blöcken verarbeitet (je 16 Ms). Die 16 aktuell verwendeten Ms werden mit $X[0]\dots X[15]$ bezeichnet und die Werte von A, B, C und D in AA, BB, CC, DD gespeichert. Danach werden in drei sich voneinander unterscheidenden "Runden" je 16 Operationen durchgeführt (zu jedem X eine).(vgl. ebd., S. 247 f.)

Runde 1

Aus Platzgründen wird die Operation

$$a = (a + f(b,c,d) + X[k]) \lll s \tag{3.4}$$

im Folgenden durch [abcd k s] repräsentiert. Es wird

[ABCD 0 3],	[DABC 1 7],	[CDAB 2 11],	[BCDA 3 19],
[ABCD 4 3],	[DABC 5 7],	[CDAB 6 11],	[BCDA 7 19],
[ABCD 8 3],	[DABC 9 7],	[CDAB 10 11],	[BCDA 11 19],
[ABCD 12 3],	[DABC 13 7],	[CDAB 14 11],	[BCDA 15 19]

ausgeführt.

Runde 2

Im Folgenden wird die Operation

$$a = (a + g(b,c,d) + X[k] + 5a827999) \lll s \tag{3.5}$$

durch [abcd k s] repräsentiert. Im Gegensatz zu Runde eins wird nun eine additive Konstante, $5a827999$(hex), verwendet. Es wird

[ABCD 0 3],	[DABC 4 5],	[CDAB 8 9],	[BCDA 12 13],
[ABCD 1 3],	[DABC 5 5],	[CDAB 9 9],	[BCDA 13 13],
[ABCD 2 3],	[DABC 6 5],	[CDAB 10 9],	[BCDA 14 13],
[ABCD 3 3],	[DABC 7 5],	[CDAB 11 9],	[BCDA 15 13]

ausgeführt.

Runde 3
Im Folgenden wird die Operation

$$a = (a + h(b, c, d) + X[k] + 6ed9eba1) \lll s \tag{3.6}$$

durch [abcd k s] repräsentiert. Es wird eine additive Konstante, $6ed9eba1$(hex), verwendet. Es wird

[ABCD 0 3],	[DABC 8 9],	[CDAB 4 11],	[BCDA 12 15],
[ABCD 2 3],	[DABC 10 9],	[CDAB 6 11],	[BCDA 14 15],
[ABCD 1 3],	[DABC 9 9],	[CDAB 5 11],	[BCDA 13 15],
[ABCD 3 3],	[DABC 11 9],	[CDAB 7 11],	[BCDA 15 15]

ausgeführt.
Nach der dritten Runde werden die Pufferwerte A, B, C und D durch A + AA,

B + BB, C + CC und D + DD ersetzt. Diese drei Runden werden für jeden 512-Bit-Block der Nachricht wiederholt. (vgl. Rivest 1992, S. 4 f.)

5. Schritt (Ausgabe)

Nachdem alle Blöcke verarbeitet wurden, ergibt die Konkatenation $A \parallel B \parallel C \parallel D$ den 128-Bit-Hashwert.

3.1.1 Beispiel

Die Nachricht x sei durch den 56-Bit-String

01100010 01100011 01110010 01111001 01110000 01110100 00111111

repräsentiert.
1. Schritt
Die Nachricht x wird zu

01100010 01100011 01110010 01111001 01110000 01110100 00111111 1

erweitert.

2. Schritt

Die Länge L, der Nachricht x ist nach dem ersten Schritt 57. Es werden $K = 391$ Nullen angehängt, was folgenden Hexadezimal-Wert ergibt.

62637279 70743f80 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000

3. Schritt

Die 64-Bit-Darstellung von L = 56 ist 00000000 00000038(hex). Der 64-Bit-String wird an die Nachricht angehängt.

62637279 70743f80 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000038

4. Schritt

Da die Nachricht nun genau 512 Bit lang ist, wird nur ein Durchlauf der Hauptschleife benötigt. Die Nachricht wird in 16 32-Bit-Blöcke geteilt.

$$\begin{aligned} X[0] &= 62637279 \\ X[1] &= 70743f80 \\ &\vdots \\ X[14] &= 00000000 \\ X[15] &= 00000038 \end{aligned}$$

Nun sind die in Abschnitt 3.1 beschriebenen Operationen der ersten Runde auszuführen. Dafür wird in (3.4) eingesetzt.

$$A = (67452301 + f(efcdab89, 98badcfe, 10325476) + X[0]) \lll 3$$

Um den Rückgabewert von $f(efcdab89, 98badcfe, 10325476)$ zu berechnen, muss in (3.2) eingesetzt werden.

$$f(efcdab89, 98badcfe, 10325476) = (efcdab89 \wedge 98badcfe) \vee ((\neg efcdab89) \wedge 10325476)$$

Nun werden die einzelnen Bitmanipulationen durchgeführt. Dafür werden die Hexadezimal-Werte als 32-Bit-Strings dargestellt.
Die erste Klammer:

$$\begin{aligned} & 1110\ 1111\ 1100\ 1101\ 1010\ 1011\ 1000\ 1001 \\ \& \ & 1001\ 1000\ 1011\ 1010\ 1101\ 1100\ 1111\ 1110 \\ = \ & 1000\ 1000\ 1000\ 1000\ 1000\ 1000\ 1000\ 1000 \end{aligned}$$

Die zweite Klammer:

$$\begin{aligned} & 0001\ 0000\ 0011\ 0010\ 0101\ 0100\ 0111\ 0110 \\ \& \ & 0001\ 0000\ 0011\ 0010\ 0101\ 0100\ 0111\ 0110 \\ = \ & 0001\ 0000\ 0011\ 0010\ 0101\ 0100\ 0111\ 0110 \end{aligned}$$

Das Endergebnis:

$$\begin{aligned} & 1000\ 1000\ 1000\ 1000\ 1000\ 1000\ 1000\ 1000 \\ | \ & 0001\ 0000\ 0011\ 0010\ 0101\ 0100\ 0111\ 0110 \\ = \ & 1001\ 1000\ 1011\ 1010\ 1101\ 1100\ 1111\ 1110 \end{aligned}$$

Anschließend wird der errechnete Wert mit 67452301(hex) und $X[0]$ modulo 2^{32} addiert.

$$\begin{aligned} 67452301 + 98badcfe + 62637279 &= 162637279 \\ 162637279 &\equiv 62637279 \mod 100000000 \end{aligned}$$

Zuletzt wird nun eine zirkuläre Linksverschiebung um 3 Bit durchgeführt.

$$\begin{aligned} & 0110\ 0010\ 0110\ 0011\ 0111\ 0010\ 0111\ 1001 \lll 3 \\ = \ & 0001\ 0011\ 0001\ 1011\ 1001\ 0011\ 1100\ 1011 \end{aligned}$$

Der Pufferwert A wird durch

$$A = 131b93cb$$

ersetzt.
Wie zu sehen ist, sind die Startwerte des Puffers (A, B, C, D) so gewählt, dass der Wert A in der ersten Operation der ersten Runde durch den um 3 Bit zirkulär linksverschobenen Wert des Wortes $X[0]$ ersetzt wird.
Nach dem Durchlaufen aller drei Runden ergibt sich der Hashwert

$$h(x) = 993ec669ce8c97edf20d01c37f090538$$

3.1.2 Erfolgreiche Angriffe

Bereits kurze Zeit nach seiner Veröffentlichung wurden Schwächen im MD4 Algorithmus entdeckt, weshalb 1991 MD5 veröffentlicht wurde.

1995

1995 gelang es Hans Dobbertin einen Angriff auf zwei der drei Runden von RIPEMD[3] zu finden. Durch eine Abstraktion dieses Angriffes konnte er erstmalig einen erfolgreichen Angriff auf den gesamten MD4 Algorithmus demonstrieren. Der Angriff benötigt nur wenige Sekunden auf einem einfachen PC (1995). In seinem Paper stellt er sogar in Frage, ob MD4 eine Einwegfunktion ist, da es sehr schnell möglich ist, einen Urbild-Angriff auf die ersten beiden Runden von MD4 durchzuführen.(vgl. Dobbertin 1995, S. 1)

2004

2004 gelang es dem chinesischen Forscherteam rund um Xiaoyun Wang einen Weg zu finden, auch ohne Computer Kollisionen zu finden. Es ist lediglich die Berechnung von

$$M' = M + \Delta C, \Delta C = (0, 2^{31}, -2^{28} + 2^{31}, 0, 0, 0, 0, 0, 0, 0, 0, 0, -2^{16}, 0, 0, 0)$$

nötig, damit $MD4(M) = MD4(M')$ ist.(vgl. Wang u. a. 2004, S. 3)

M_1	4d7a9c83 56cb927a b9d5a578 57a7a5ee de748a3c dcc366b3 b683a020 3b2a5d9f c69d71b3 f9e99198 d79f805e a63bb2e8 45dd8e31 97e31fe5 2794bf08 b9e8c3e9
M_1	4d7a9c83 d6cb927a 29d5a578 57a7a5ee de748a3c dcc366b3 b683a020 3b2a5d9f c69d71b3 f9e99198 d79f805e a63bb2e8 45dc8e31 97e31fe5 2794bf08 b9e8c3e9
H	5f5c1a0d 71b36046 1b5435da 9b0d807a
M_2	4d7a9c83 56cb927a b9d5a578 57a7a5ee de748a3c dcc366b3 b683a020 3b2a5d9f c69d71b3 f9e99198 d79f805e a63bb2e8 45dd8e31 97e31fe5 f713c240 a7b8cf69
M_2	4d7a9c83 d6cb927a 29d5a578 57a7a5ee de748a3c dcc366b3 b683a020 3b2a5d9f c69d71b3 f9e99198 d79f805e a63bb2e8 45dc8e31 97e31fe5 f713c240 a7b8cf69
H	e0f76122 c429c56c ebb5e256 b809793

Tabelle 5: Je zwei Nachrichten, die unter MD4 kollidieren (Wang u. a. 2004, S. 3)

[3]Bei RIPEMD handelt es sich um eine von Hans Dobbertin entworfene Hashfunktion.

3.2 Message-Digest Algorithm 5 (MD5)

Bei MD5 handelt es sich um eine verbesserte Version von MD4. Als es Boer, Bosselaers und Merkel gelang, erfolgreich die ersten zwei Runden(Merkel) bzw. die letzten beiden Runden (Boer und Bosselaers) von MD4 anzugreifen, veröffentlichte Rivest 1991 MD5. Dies geschah noch bevor Angriffe auf MD4 bekannt waren.(vgl. Schneier 1996, S. 498)
Da bei MD5 (im Gegensatz zu MD4) vier Runden durchgeführt werden, werden vier nichtlineare Funktionen benötigt. Es sei

$$f(x, y, z) \coloneqq (x \wedge y) \vee ((\neg x) \wedge z) \tag{3.7}$$
$$g(x, y, z) \coloneqq (x \wedge y) \vee (y \wedge (\neg z)) \tag{3.8}$$
$$h(x, y, z) \coloneqq x \oplus y \oplus z \tag{3.9}$$
$$i(x, y, z) \coloneqq y \oplus (x \vee (\neg z)) \tag{3.10}$$

definiert.
Die ersten drei Schritte unterscheiden sich nicht von den in Abschnitt 3.1 beschriebenen.

1.Schritt (1-0-Padding)

2.Schritt (Längenpadding)

3.Schritt (Initialisierung)

4.Schritt (Argumentverarbeitung)

Wie in MD4 wird der Eingabetext in Blöcken zu je 512 Bit, aufgeteilt in 16 Teilblöcke der Länge 32 Bit, verarbeitet. Die 16 aktuellen Teilblöcke werden erneut mit $X[0] \dots X[15]$ bezeichnet. Weiters benötigt MD5 64 konstante Wörter, die im Folgenden mit $T[1] \dots T[64]$ bezeichnet werden. $T[i]$ bezeichnet den ganzzahlige Teil von $2^{32}|sin(i)|$, wobei i im Bogenmaß gemessen wird.
Die Werte von A, B, C und D werden in AA, BB, CC, DD gespeichert. Danach werden in vier sich leicht voneinander unterscheidenden "Runden" je 16 Operationen durchgeführt. (vgl. Rivest 1992, S. 4)

Runde 1
Die Operation

$$a = b + ((a + f(b, c, d) + X[k] + T[i]) \lll s)$$

wird durch [abcd k s i] repräsentiert. Es wird

[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]

ausgeführt.

Runde 2
Die Operation

$$a = b + ((a + g(b, c, d) + X[k] + T[i]) \lll s)$$

wird durch [abcd k s i] repräsentiert. Es wird

[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]

ausgeführt.

Runde 3
Die Operation

$$a = b + ((a + h(b, c, d) + X[k] + T[i]) \lll s)$$

wird durch [abcd k s i] repräsentiert. Es wird

[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]

ausgeführt.

Runde 4
Die Operation

$$a = b + ((a + i(b, c, d) + X[k] + T[i]) \lll s)$$

wird durch [abcd k s i] repräsentiert. Es wird

[ABCD 0 6 49], [DABC 7 10 50], [CDAB 14 15 51], [BCDA 5 21 52],
[ABCD 12 6 53], [DABC 3 10 53], [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]

ausgeführt.(vgl. Rivest 1992, S. 5)
Nach der dritten Runde werden die Pufferwerte A, B, C und D durch A + AA,

B + BB, C + CC und D + DD ersetzt.
Diese drei Runden werden für jeden 512-Bit-Block der Nachricht wiederholt.

5. Schritt (Ausgabe)

Nachdem alle Blöcke verarbeitet wurden, ergibt die Konkatenation $A \parallel B \parallel C \parallel D$ den 128-Bit-Hashwert.

3.2.1 Unterschiede zu MD4

Ron Rivest beschreibt die Änderungen von MD5 wie folgt(Schneier 1996, S. 502 f.):

1. Eine vierte Runde wurde hinzugefügt.
2. Jeder Schritt enthält jetzt eine eindeutige additive Konstante.
3. Die Funktion G in Runde 2 wurde von $(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$ zu $(x \wedge y) \vee (y \wedge (\neg z))$ geändert, um G weniger symmetrisch zu machen.[4]
4. In jedem Schritt wird jetzt das Ergebnis des vorherigen Schrittes addiert. Dies liefert einen schnelleren Lawineneffekt[5].
5. Die Reihenfolge, in der in den Runden 2 und 3 auf die Teilblöcke zugegriffen wird, wurde geändert, um ähnliche Muster zu vermeiden.
6. Die Beträge der zirkulären Linksverschiebungen in jeder Runde wurden optimiert, um einen schnelleren Lawineneffekt zu erreichen. Die vier Verschiebungen in den einzelnen Runden unterscheiden sich von denen in den anderen Runden.

[4] mit G ist g in (3.8) gemeint
[5] Bei einer minimalen Änderung der Eingabe, ändert sich die Ausgabe völlig.

3.2.2 Erfolgreiche Angriffe

1993

1993 gelang es Bert den Boer und Antoon Bosselaers einen Algorithmus zu entwickeln, um Kollisionen für die in MD5 verwendete Kompressionsfunktion der ersten beiden Runden zu finden. Der Angriff stützt sich auf eine der zwei Änderungen, die Rivest vornahm, um MD5 sicherer zu machen als MD4. (vgl. Boer und Bosselaers 1993, S. 1)

1996

Hans Dobbertin gelang es zwei 512-Bit-Strings zu finden, welche unter einer veränderten Version von MD5 kollidieren. Die Änderung beschränkt sich auf die Initialisierung der Puffer-Werte. Obwohl dies keine Auswirkungen auf den echten MD5 Algorithmus hat, zeigt es die Schwächen von MD5 auf.(vgl. Wang u. a. 2004, S. 1)

2004

Wang und seinem Team gelang es einen Weg zu finden, um Kollisionen zu errechnen. Dafür wird für eine 1024-Bit Nachricht (M, N_i)

$$M' = M + \Delta C_1, \Delta C_1 = (0, 0, 0, 0, 2^{31}, \ldots, 2^{15}, 0, 0, 2^{31}, 0)$$
$$N_i' = N_i + \Delta C_2, \Delta C_2 = (0, 0, 0, 0, 2^{31}, \ldots, -2^{15}, 0, 0, 2^{31}, 0)$$

berechnet, sodass $MD5(M, N_i) = MD5(M', N_i')$ ist.

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
X_1	M'	2dd31d1 c4eee6c5 69a3d69 5cf9af98 7b5ca2f ab7e4612 3e580440 897ffbb8 634ad55 2b3f409 8388e483 5a41f125 e8255108 9fc9cdf7 72bd1dd9 5b3c3780
	N_1	d11d0b96 9c7b41dc f497d8e4 d555655a 479a7335 cfdebf0 66f12930 8fb109d1 797f2775 eb5cd530 baade822 5c154c79 ddcb74ed 6dd3c55f 580a9bb1 e3a7cc35
H		9603161f f41fc7ef 9f65ffbc a30f9dbf

Tabelle 6: Zwei Nachrichten, die unter MD5 kollidieren (Wang u. a. 2004, S. 3)

3.3 Secure Hash Algorithm (SHA-1)

Von der NSA in Zusammenarbeit mit dem NIST entwickelt, sollte SHA-1 die im SHS (Secure Hash Standard) verwendete Hashfunktion sein. SHA-1 ist wie MD5 eine Verbesserung/Weiterentwicklung des MD4 Algorithmus. SHA-1 erzeugt für jede Eingabe $< 2^{64}$ Bit einen 160 Bit langen Hashwert. Es werden vier Runden verwendet. Es seien dabei f_1, f_2, f_3, f_4 wie folgt definiert:(vgl. Schneier 1996, S. 504)

$$
\begin{aligned}
f_1(x,y,z) &\coloneqq (x \wedge y) \vee ((\neg x) \wedge z) \\
f_2(x,y,z) &\coloneqq x \oplus y \oplus z \\
f_3(x,y,z) &\coloneqq (x \wedge y) \vee (y \wedge (\neg z)) \\
f_4(x,y,z) &\coloneqq x \oplus y \oplus z
\end{aligned}
$$

1.Schritt (1-0-Padding)

Dieser Schritt unterscheidet sich nicht von dem in Abschnitt 3.1 beschriebenen.

2.Schritt (Längenpadding)

Die Länge der ursprünglichen Nachricht in Bit wird als 64-Bit-String angehängt. Im Gegensatz zu MD4 wird jedoch das höherwertige Wort zuerst konkateniert. (vgl. Greveler 1998, S. 26 f.)

3.Schritt (Initialisierung)

Da SHA-1 einen 160 Bit langen Hashwert liefert, werden zwei fünf 5-Wort-Puffer benötigt. Die fünf Puffervariablen werden wie folgt initialisiert. (vgl. Schneier 1996, S. 505 f.)

$$
\begin{aligned}
A &= 67452310 \\
B &= efcdab89 \\
C &= 98badcfe \\
D &= 10325476 \\
E &= c3d2e1f0
\end{aligned}
$$

4.Schritt (Argumentverarbeitung)

Der Eingabetext wird in 512-Bit-Blöcken verarbeitet. Jeder Block wird in 16 Teilblöcke gespalten, welche mit $M[0] \ldots M[15]$ bezeichnet werden. Die 16 Ms werden in 80 32-Bit-Strings umgewandelt.

$$
\begin{aligned}
W_t &= M_t \qquad \text{für } t = 0,\ldots,15 \\
W_t &= (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \lll 1 \qquad \text{für } t = 16,\ldots,79
\end{aligned}
$$

Weiters werden vier additive Konstanten benötigt.

$$K_1 = \frac{\sqrt{2}}{4} = 5a827999$$
$$K_2 = \frac{\sqrt{3}}{4} = 6ed9eba1$$
$$K_3 = \frac{\sqrt{5}}{4} = 8f1bbcdc$$
$$K_4 = \frac{\sqrt{10}}{4} = ca62c1d6$$

Die Hauptschleife besteht aus vier Runden zu je 20 Operationen und wird für jeden 512-Bit-Block durchlaufen.
Zunächst werden die Variablen A, B, C, D und E in a, b, c, d und e gespeichert. Danach werden die vier Runden durchgeführt:

Runde 1

Für $t = 0, \ldots, 19$

$$TEMP = (a \lll 5) + f_1(b, c, d) + e + W_t + K_1$$
$$e = d$$
$$d = c$$
$$c = b \lll 30$$
$$b = a$$
$$a = TEMP$$

Runde 2

Für $t = 20, \ldots, 39$

$$TEMP = (a \lll 5) + f_2(b, c, d) + e + W_t + K_2$$
$$e = d$$
$$d = c$$
$$c = b \lll 30$$
$$b = a$$
$$a = TEMP$$

Runde 3

$$
\begin{aligned}
&\text{Für } t = 40, \ldots, 59 \\
&\qquad TEMP = (a \lll 5) + f_3(b, c, d) + e + W_t + K_3 \\
&\qquad\qquad e = d \\
&\qquad\qquad d = c \\
&\qquad\qquad c = b \lll 30 \\
&\qquad\qquad b = a \\
&\qquad\qquad a = TEMP
\end{aligned}
$$

Runde 4

$$
\begin{aligned}
&\text{Für } t = 60, \ldots, 79 \\
&\qquad TEMP = (a \lll 5) + f_4(b, c, d) + e + W_t + K_4 \\
&\qquad\qquad e = d \\
&\qquad\qquad d = c \\
&\qquad\qquad c = b \lll 30 \\
&\qquad\qquad b = a \\
&\qquad\qquad a = TEMP
\end{aligned}
$$

Die Variablen a, b, c, d und e werden zu A, B, C, D und E addiert und die Hauptschleife wird für den nächsten Block wiederholt.(vgl. Schneier 1996, S. 505 f.)

Durch die Verschiebung der Variablen erreicht der Algorithmus das Gleiche wie MD5 durch die Verwendung verschiedener Variablen an verschiedenen Stellen.(ebd., S. 506)

5. Schritt (Ausgabe)

Nachdem alle Blöcke verarbeitet wurden, ergibt die Konkatenation $A \parallel B \parallel C \parallel D \parallel E$ den 160-Bit langen Hashwert.

3.3.1 Vergleich zu MD5

Sowohl bei MD5 als auch bei SHA handelt es sich um Weiterentwicklungen des MD4 Algorithmus. Die Verbesserungen unterscheiden sich in folgenden Punkten: (vgl. Schneier 1996, S. 507)

1. Sowohl bei MD5 als auch bei SHA wurde eine vierte Runde hinzugefügt.
2. Während in MD5 jeder Schritt eine eindeutige additive Konstante enthält, behält SHA das Schema von MD4 bei. Je eine Konstante pro Runde.
3. SHA behält die alte Funktion g bei. MD5 hat diese durch $((X \wedge Y) \vee (Y \wedge (\neg Z)))$ ersetzt.
4. Sowohl bei SHA als auch bei MD5 wird in jedem Schritt das Ergebnis des vorherigen Schrittes addiert. SHA verwendet dafür eine fünfte Variable, was den Angriff von Boer und Bosselaers auf MD5 verhindert.
5. "Die Reihenfolge, in der in Runde zwei und drei auf die Teilblöcke zugegriffen wird, wurde geändert, um ähnliche Muster zu vermeiden" SHA ist völlig anders, da es einen zyklisch fehlerkorrigierten Code benutzt.(ebd., S. 507)
6. Bei MD5 wurden die Beträge der Linksverschiebung für einen schnelleren Lawineneffekt optimiert. SHA verschiebt in jeder Runde um einen konstanten Betrag.

3.3.2 Erfolgreiche Angriffe

2005

2005 gelang es den Wissenschaftlern Xiaoyun Wang, Yiqun Lisa Yin und Hongbo Yu erstmals SHA-1 zu brechen. Kollisionen können mit nur 2^{69} Berechnungen gefunden werden. Das ist mehr als 2000-mal schneller als Bruteforce. (vgl. Schneier 2005)

2015

2015 ist erstmals ein nicht nur theoretischer Angriff auf SHA-1 gelungen. Die Forscher Marc Stevens, Pierre Karpman und Thomas Peyrin haben eine Freestart-Kollision, für die in SHA-1 verwendete Kompressionsfunktion, gefunden.
(vgl. Schneier 2015)

Bis heute wurde keine Kollision für SHA-1 veröffentlicht.

4 Angriffe auf Passwörter

In der Praxis werden Hashfunktionen oft für die Sicherung von Passwörtern verwendet. Ziel ist es, die beim Hack einer Datenbank gestohlenen Passwörter für Angreifer unbrauchbar zu machen, da nur der Hashwert gespeichert wird. Dieser kann nicht für das login verwendet werden. Der Versuch Passwörter zu schützen, ist ausgesprochen wichtig, da viele Nutzer dasselbe Passwort für mehr als einen Account verwenden (z.B. für Onlinebanking und Facebook).
Da die beschriebenen Hashfunktionen aber eigentlich nicht für die Sicherung von Passwörtern entworfen wurden, stellt sich die Frage, ob sie sich dennoch dafür eignen. Um zu überprüfen, ob sich Hashfunktionen für die Sicherung von Passwörtern eignen, muss die Vorgehensweise eines Angreifers untersucht werden. In diesem, dem empirischen Teil meiner Arbeit, habe ich deshalb versucht, gehashte Passwörter zu entschlüsseln. Ich habe verschiedene Angriffsmethoden getestet und untereinander verglichen.

4.1 Vorgehensweise

Um ein realistisches Angriffsszenario nachzustellen, habe ich den Inhalt von zwei gehackten Datenbanken angegriffen[6]. Dabei handelt es sich um die 2016 veröffentliche Yahoo-Liste und die 2009 veröffentlichte Rockyou-Liste. Diese enthalten die Passwörter von 450 000 bzw. 14 000 000 Nutzern im Klartext. Da Menschen ihre Passwörter nicht wie ein Zufallsgenerator wählen, ist es sehr wichtig, mit Passwörtern zu testen, die tatsächlich von Menschen gewählt und verwendet wurden.
Für einige Angriffe ist die Analyse von bereits bekannten Passwörtern notwendig, weshalb beide Listen in jeweils ein Trainingsset und ein Testset geteilt wurden. Dadurch wird sichergestellt, dass Wörter, die erraten werden sollen, nicht analysiert werden, da dies in einem echten Angriff nicht möglich wäre.
Da es sich bei den verwendeten Hashfunktionen um Einwegfunktionen handelt, ist es rechnerisch nicht möglich, die Passwörter zu entschlüsseln. Um dennoch herauszufinden, welches Passwort den gesuchten Hashwert liefert, muss ein beliebiges Wort geraten werden. Anschließend wird der Hashwert berechnet und geprüft, ob er mit dem gesuchten Hash des Passwortes übereinstimmt. Ist dies der Fall, handelt es sich bei dem geratenen Wort entweder um das Passwort oder, weit weniger wahrscheinlich, um eine Kollision. Beides kann für das Login verwendet werden.

[6]Der Inhalt der Datenbanken ist frei im Internet verfügbar.

4.2 Verwendete Programme

Im Mittelpunkt jedes von mir getesteten Angriffes steht John the Ripper (JtR), ein Programm, das es erlaubt, Daten mittels Pipe (stdio) von einem Script/Programm zu erhalten, den Hashwert zu bilden und anschließend mit einer Liste abzugleichen, um Übereinstimmungen zu finden.

Bei den Scripten/Programmen handelt es sich um von mir geschriebenen Code, der je nach Methode entsprechend vorgeht. Aufgrund der Geschwindigkeit und Einfachheit der Anwendung ist C die Programmiersprache der Wahl. Sämtlicher verwendeter Sourcecode ist in den folgenden Kapiteln angeführt und erklärt. Alle Tests wurden aufgrund der weitaus besseren Shell unter Linux durchgeführt.

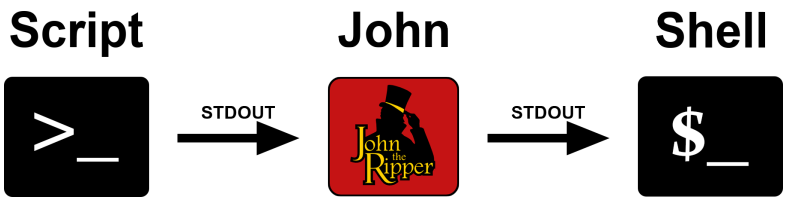

Abbildung 4: JtR Workflow

4.3 Bruteforce

Bei einem Bruteforce-Angriff handelt es sich um eine sehr naive Methode, Passwörter zu erraten. Dabei werden alle möglichen Kombinationen getestet um einen Wert zu finden, der den gewünschten Hash liefert. Die Erfolgswahrscheinlichkeit eines Bruteforce-Angriffes liegt bei 100%. Da die Zahl der möglichen Passwörter und damit die benötigte Zeit exponentiell mit der Länge des Wortes steigt, eignet sich diese Methode jedoch nur bedingt um lange Passwörter zu knacken.
Programmiertechnisch ist ein Bruteforce-Angriff leicht umzusetzen.

```
1  void getGuess()
   {
3      const char chars[CHAR_COUNT+1] = "abcdefghijklmnopqrstuvwxyz";
       int i, j;
5      int guessc[MAX_SIZE] = {0};
       char guess[MAX_SIZE+1];
7
       for(i = 1; i < MAX_SIZE ; guessc[i++] = -1);
9      for(i = 1; i <= MAX_SIZE ; guess[i++] = '\0');

11     while(1)
       {
13         i=0;
           while(guessc[i]==CHAR_COUNT)
15         {
               guessc[i]=0;
17             guessc[++i]+=1;
           }
19
           for(j=0;j<=i;++j)
21         {
               if(j < MAX_SIZE)
23                 guess[j]=chars[guessc[j]];
           }
25
           printf("%s\n",guess);
27
           ++guessc[0];
29     }
   }
```

bruteforce.c

Zunächst wird jedem verwendeten Zeichen eine Zahl zugeordnet.
Die Funktionsweise des Algorithmus ist mit der eines Fahrradschlosses (Zahlenschloss) zu vergleichen. Das Fahrradschloss hat n Scheiben, wobei auf jeder statt der Ziffern von 0 bis 9 alle zuvor zugeordneten Zahlen stehen. Nun wird die erste

Scheibe um eine Stelle weitergedreht. Sobald eine Scheibe eine ganze Umdrehung abgeschlossen hat, sodass wieder die erste Zahl zu sehen ist, wird auch die nächste Scheibe um eine Position verdreht.
Es wird vor jeder Bewegung der ersten Scheibe ein Wort gebildet, indem jede Zahl durch das zuvor zugewiesene Zeichen ersetzt wird. Da die Zahl der sich ändernden

Zahlen-kombination	25	0	7	0	13	0
zugeordnetes Zeichen	z	a	h	a	n	a

Tabelle 7: Beispiel einer Zuordnung

Zahlen-kombination	0	1	7	0	13	0
zugeordnetes Zeichen	a	b	h	a	n	a

Tabelle 8: Tabelle 7 nach einer weiteren Rotation

Zeichen gleich der Anzahl gedrehter Scheiben k ist, reicht es, die ersten k Zeichen des Arrays zu ersetzen. Dadurch kann die Geschwindigkeit stark gesteigert werden.

Zahl der Versuche	Benötigte Zeit
$10,000,000$	$0.014s$
$100,000,000$	$0.138s$
$1,000,000,000$	$1.383s$
$10,000,000,000$	$11.152s$

Tabelle 9: Geschwindigkeit des vorgestellten Algorithmus

4.4 Markow-Ketten

Um einen weitaus effektiveren Angriff nachzustellen, habe ich nun jedes Zeichen eines Passwortes als Teil einer Markow-Kette betrachtet. Dieser Angriff wird repräsentativ für eine Gruppe von Angriffen, die sich auf die Analyse bereits bekannter Passwörter stützt durchgeführt. Im Gegensatz zu einem Bruteforce-Angriff werden dabei nicht alle möglichen Kombinationen ausprobiert, sondern nur jene, die auch im Trainingsset vorgekommen sind. Um ein n-stelliges Passwort zu erhalten, werden n Zeichen aneinander gereiht, wobei die bedingte Wahrscheinlichkeit, dass das Zeichen x_j gewählt wird, von dem vorangegangenen Zeichen x_i abhängig ist. Es gilt $p_{ij} \coloneqq \mathbb{P}(aktuelles\ Zeichen = x_j | vorangegangenes\ Zeichen = x_i)$.

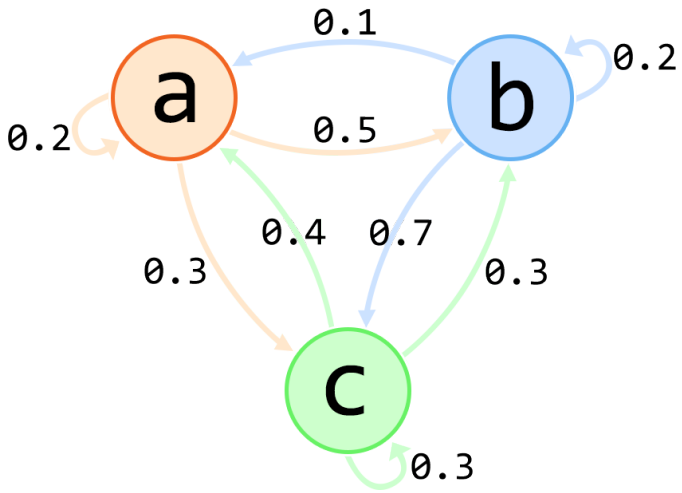

Abbildung 5: Schematische Darstellung einer Markow-Kette

x_i und x_j sind Elemente der Menge $I = \{0, 1, 2, \ldots, 126, 127\}$, die alle 128 möglichen ASCII-Zeichen beinhaltet. Sämtliche Wahrscheinlichkeiten werden in der Übergangsmatrix (stochastische Matrix) P angegeben. Da es sich bei den Einträgen um Wahrscheinlichkeiten handelt, gilt $0 \leq p_{ij} \leq 1, \forall i, j \in I$. Für die Zeilensumme der Übergangsmatrix gilt $\sum_{j \in I} p_{ij} = 1, \forall i \in I$.

$$P = \begin{pmatrix} p_{11} & p_{12} & \cdots & p_{1j} \\ p_{21} & p_{22} & \cdots & p_{2j} \\ \vdots & \vdots & \ddots & \vdots \\ p_{i1} & p_{i2} & \cdots & p_{ij} \end{pmatrix} \tag{4.1}$$

Da es für das Zeichen x_0 keinen Vorgänger gibt, wird es mit Hilfe der Anfangsverteilung λ gewählt. So gibt λ_0 an, mit welcher Wahrscheinlichkeit das erste Zeichen

des Wortes NUL ist.(vgl. Brzeźniak und Zastawniak 1999, S. 85 ff.)

$$\lambda = (\lambda_i, i \in I) \tag{4.2}$$

$$0 \leq \lambda_i \leq 1 \tag{4.3}$$

Die Anfangsverteilung (4.2) der Yahoo-Liste sieht wie folgt aus:

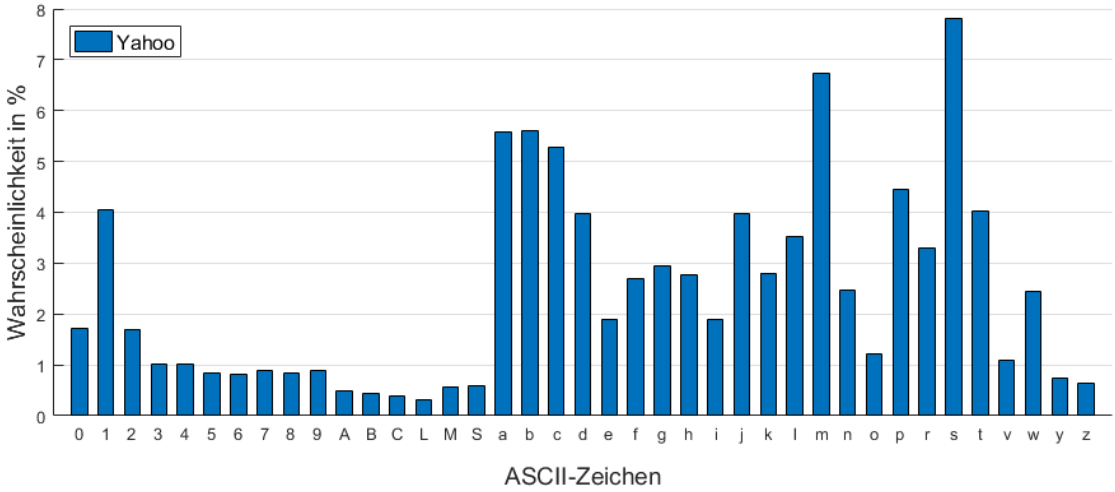

Abbildung 6: Anfangsverteilung der Yahoo-Liste

Alle Einträge mit $p \leq 0.005$ wurden aus Platzgründen aus dem Diagramm entfernt, sind aber in Anhang B zu finden und wurden in allen Tests berücksichtigt. Aus den gesammelten Daten lässt sich nun ablesen, welche Zeichen mit welcher Wahrscheinlichkeit an der erster Stelle des Wortes vorkommen.

4.4.1 Markow-Ketten verschiedener Ordnung

Man unterscheidet Markow-Ketten unterschiedlicher Ordnung. Bei einer Markow-Kette erster Ordnung hängt die Zukunft nur von der Gegenwart ab. Bei einer Markow-Kette zweiter Ordnung hängt die Zukunft von der Gegenwart und der Vergangenheit ab.
Um die beiden vergleichen zu können, habe ich jeweils 500 000 000 Mal ein 7-stelliges Passwort geraten und mit dem Rockyou-Testset abgeglichen. In Abb. 7 ist zu sehen, wie viele der 7 000 000 Passwörter entschlüsselt werden konnten.

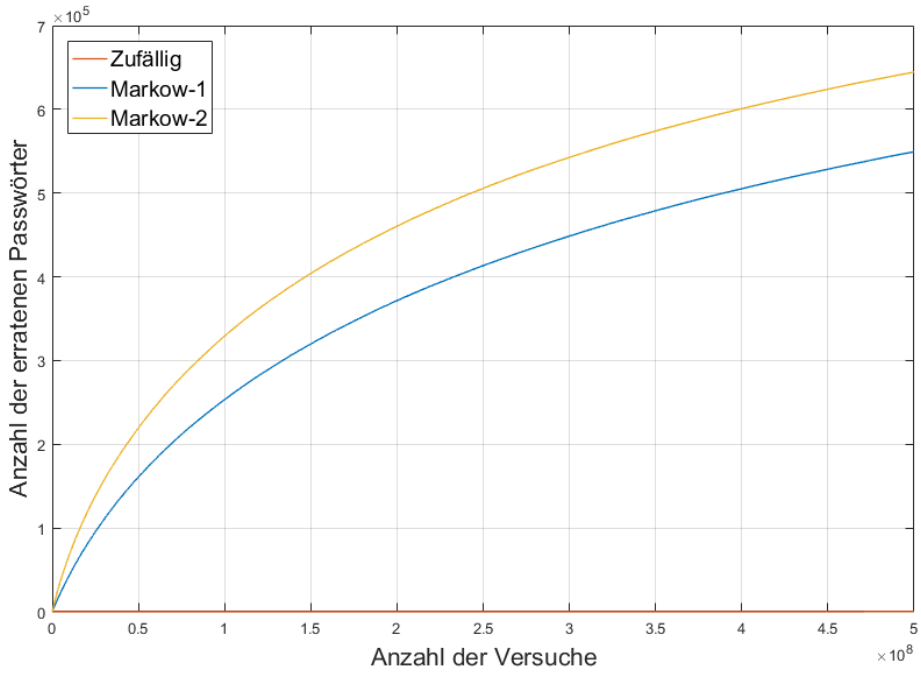

Abbildung 7: Markow-Ketten verschiedener Ordnung

Das rein zufällige Raten hat dabei mit 180 erratenen Wörtern am schlechtesten abgeschnitten, während mithilfe einer Markow-Kette erster Ordnung 549 362 Passwörter entschlüsselt werden konnten. Mit einer Markow-Kette zweiter Ordnung konnte ich sogar 644 568 Hashes knacken.

Um die Effektivität dieser Methode zu erklären, wird untersucht, mit welcher Häufigkeit jedes Zeichen in den beiden Listen vorkommt. Betrachtet man nun in Abb. 8 die Ergebnisse, lässt sich eine große Ähnlichkeit zwischen den beiden Listen feststel-

len. Dieses Ergebnis ist durchaus erstaunlich, da zwischen den Veröffentlichungen der beiden Listen rund sieben Jahre liegen und sie von unterschiedlichen Webseiten stammen.
Unter der Annahme, dass alle Passwortlisten mit genügend vielen Einträgen eine sehr ähnliche Verteilung der Zeichen haben, werden häufige Zeichen öfter geraten als weniger häufige.
Programmiertechnisch ist solch ein Angriff wie folgt realisierbar:

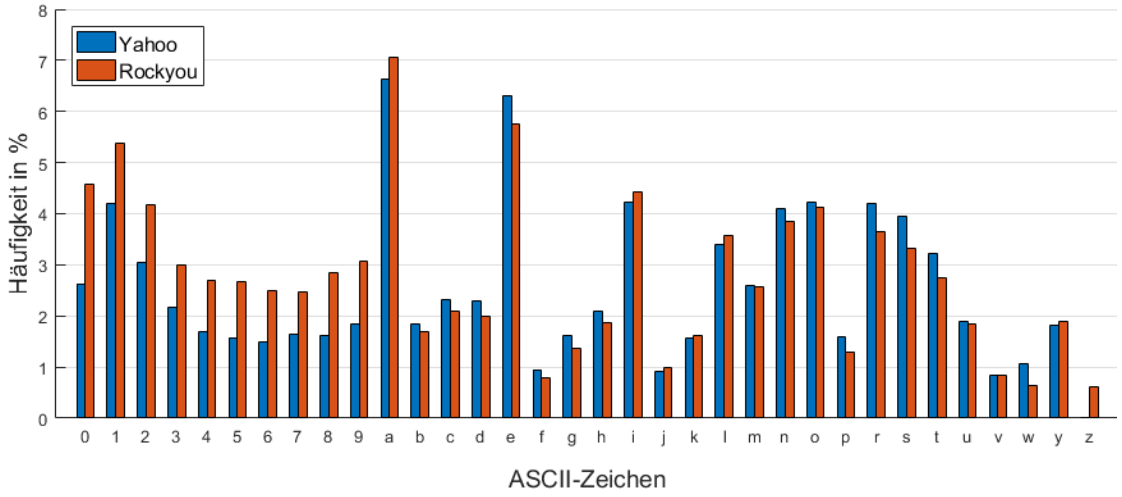

Abbildung 8: Zeichen mit $h_n(A) > 0.5\%$

```
for(k=1;k<length;++k)
{
    x=(rand()/RAND_MAX);

    for(j=0;j<128;++j)
    {
        if(x<=(stochMatrix[i][j]+totalProb))
        {
            guess[k]=j;
            i=j;
            totalProb=0;
            break;
        }
        totalProb+=stochMatrix[i][j];
    }
}
```

markov.c

Aus Gründen der Lesbarkeit wurde die Deklaration und Initialisierung der Variablen, sowie die Wahl des ersten Zeichens x_0 aus dem Quellcode gestrichen.
Die erste for-Schleife wiederholt die Wahl von Zeichen solange, bis das Wort die gewünschte Länge hat. In Zeile drei wird eine Zufallszahl $0 \leq x \leq 1$ gebildet. Nun wird der ASCII-Zeichensatz durchgegangen und die stochastische Matrix an der jeweiligen Stelle geprüft. Solange die Variable $totalProb < x$ ist, werden die Einträge der Matrix addiert. Da $\sum_{j \in I} p_{ij} = 1, \forall i \in I$ ist und $x \leq 1$ ist, wird x sicherlich kleiner gleich $totalProb$ werden. Wenn dies der Fall ist, wird dem Vektor $guess$ an der Stelle k, der 8-Bit-Integer j hinzugefügt, der aktuelle Wert j für den nächsten Durchlauf in i gespeichert und $totalProb$ zurückgesetzt. Da in C jedes Zeichen als 8-Bit-Integer dargestellt wird, wird die Zählvariable j zum entsprechenden ASCII-Zeichen konvertiert.

4.4.2 Weitere Verbesserungen

Um die Methode weiter zu verbessern, habe ich einige kleine Änderungen vorgenommen, weshalb es sich nun nicht mehr um eine Markow-Kette handelt.

(i) Bisher wurden nur 7-stellige Passwörter geraten. Durch Analyse der Listen wird nun auch die Länge des Passwortes richtig verteilt. Dieser Vorgang ist mit der Anfangsverteilung zu vergleichen.

(ii) Bisher wurde für jede Stelle bis auf die erste die Übergangsmatrix P genutzt. Nun wird jeder Stelle i eine eigene Übergangsmatrix P_i zugewiesen. Da sich die stochastische Matrix nun von Schritt zu Schritt ändert, kann nicht mehr von einer Markow-Kette gesprochen werden.

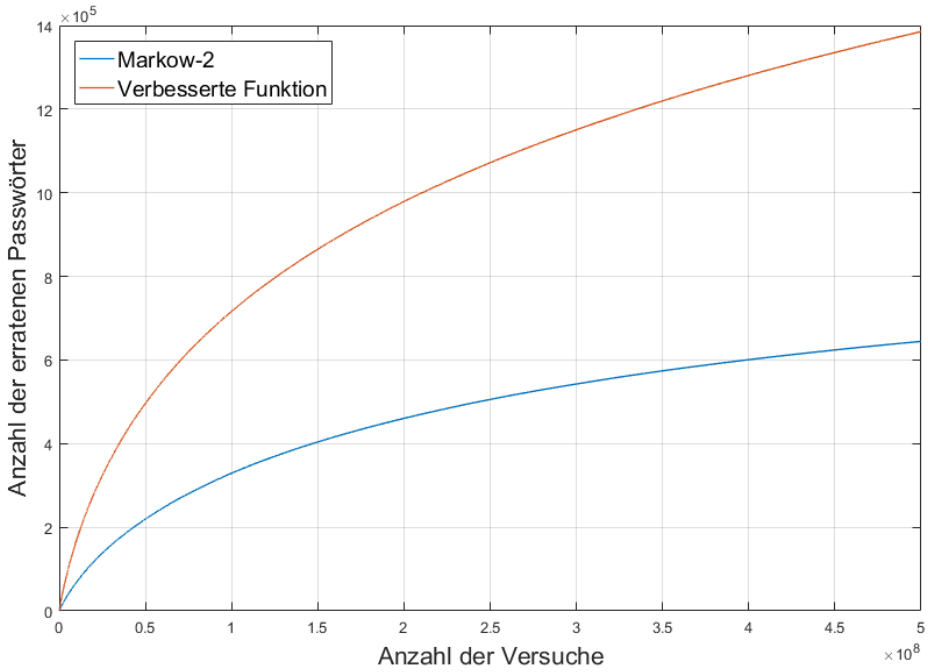

Abbildung 9: Markow-Kette vs. verbesserte Funktion

Wie in Abb. 9 zu sehen ist, haben diese kleinen Änderungen gereicht, die Effektivität einer Markow-Kette zweiter Ordnung mehr als zu verdoppeln. Es wurden 1 385 589 der Hashes entschlüsselt.
Damit ist diese Methode 7 697-mal so effektiv wie rein zufälliges Raten.

4.5 Gezielte Angriffe

Bisher habe ich versucht, möglichst effizient, möglichst viele Passwörter der beiden Listen zu erraten. Nun möchte ich gezielt ein einzelnes Passwort angreifen, um einen Angriff auf ein Unternehmen zu simulieren. Da die Schäden, die jährlich durch Internetkriminalität entstehen, auf 24,3 Milliarden Euro geschätzt werden (vgl. Dörner 2011), ist es für Unternehmen essentiell zu wissen, wie Passwörter nicht gewählt werden sollten. Durch die Simulation eines solchen Angriffes lassen sich vorbeugende Maßnahmen treffen.

Als Passwort werde ich *Mil2013!* verwenden. Dieses Passwort stammt aus der Yahoo-Liste und wurde gezielt gewählt, um einige Methoden zu demonstrieren.

Da es 94 ASCII-Zeichen gibt, die für ein Passwort in Frage kommen, gibt es 94^n Möglichkeiten ein n-stelliges Passwort zu bilden. Unter der Annahme, dass das gesuchte Wort zufällig gewählt wurde und es sich um eine diskrete Gleichverteilung handelt, gilt für den Erwartungswert

$$\mathbb{E}[X] = \frac{1}{n} \sum_{i=1}^{n} x_i. \tag{4.4}$$

Durch Einsetzen in (4.4) und mithilfe der Gaußschen Summenformel erhält man

$$\mathbb{E}[X] = \frac{94^n + 1}{2}. \tag{4.5}$$

Mit 10 000 000 Versuchen pro Sekunde würde ein einfacher Bruteforce-Angriff auf das 8-stellige Beispielpasswort also 9.6 Jahre dauern[7]. Die benötigte Zeit $t(x)$ lässt sich als Exponentialfunktion darstellen.

$$t(x) = \frac{94^x + 1}{2 \cdot 10^7} \tag{4.6}$$

[7] Es handelt sich um die durchschnittlich benötigte Zeit.

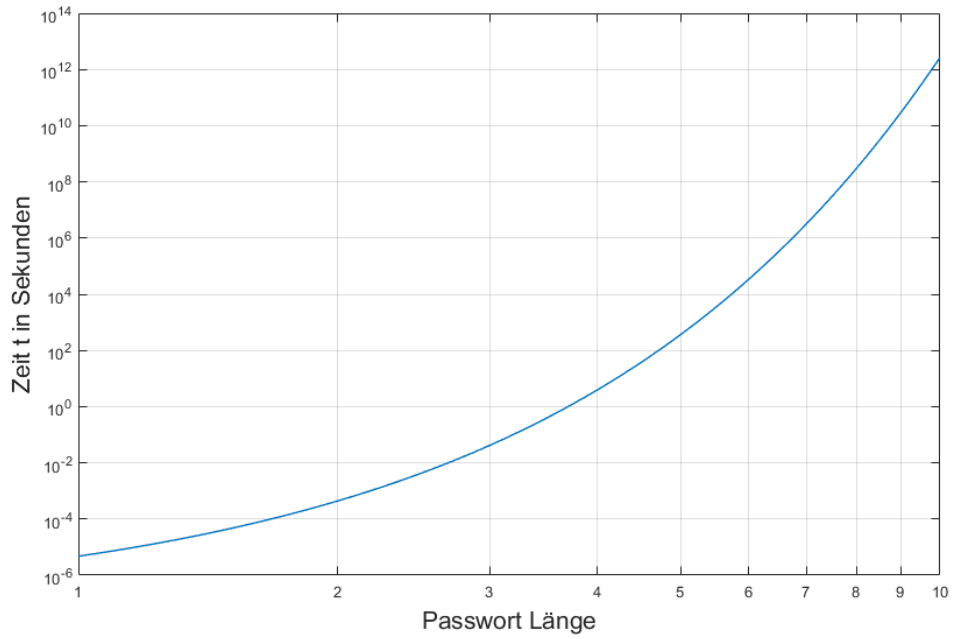

Abbildung 10: Plot zu (4.6)

Um die benötigte Zeit zu senken, muss versucht werden, das Passwort zu verkürzen. Gelingt es, das Passwort auf 7 Zeichen zu reduzieren, dauert der Angriff nur noch 37.5 Tage. Damit dies möglich ist, muss mindestens ein Zeichen des Passwortes bekannt sein. Anschließend wird ein Bruteforce-Angriff durchgeführt, wobei jedoch nur Kombinationen getestet werden, die das bekannte Zeichen beinhalten. Dabei ist es irrelevant, an welcher Stelle sich dieses befindet.

Sollte kein Zeichen bekannt sein, können Wortteile den gewünschten Erfolg bringen. Dabei wird ähnlich vorgegangen wie bei einem Bruteforce-Angriff. Anstelle eines Char-Arrays wird jedoch ein String-Array verwendet. Der Zeichensatz wird dabei durch Wörter ergänzt, in der Hoffnung, dass eines der hinzugefügten Wörter tatsächlich im Passwort enthalten ist.

Um diese Vorgehensweise besser zu verstehen, betrachte man das Beispielpasswort *Mil2013!*. Ich nehme an, dass, wie in vielen Firmen üblich, auch das Beispielpasswort in regelmäßigen Abständen geändert wird. Man könnte also das aktuelle Jahr als Wort zu dem verwendeten Zeichensatz hinzufügen. Für $\hat{t}(x)$ ergibt sich also

$$\hat{t}(x) = \frac{95^x + 1}{2 \cdot 10^7} \tag{4.7}$$

Grundsätzlich steigt die benötigte Zeit dadurch an. Da das Passwort allerdings tatsächlich das aktuelle Jahr beinhaltet, wurde aus dem 8 Zeichen Passwort ein 5 Zeichen langes Wort, weil die 4-stellige Jahreszahl als einzelnes Zeichen geraten wurde. Für die benötigte Zeit ergibt sich demnach

$$\hat{t}(5) = \frac{95^5 + 1}{2 \cdot 10^7} = 366.9520s. \tag{4.8}$$

Da es sich in (4.7) um eine Exponentialfunktion handelt, kann der Zeichensatz stark vergrößert werden. Solange dadurch erreicht wird, dass der Exponent kleiner wird, sinkt die benötigte Zeit dramatisch. Um herauszufinden wie viele Wörter dem verwendeten Zeichensatz hinzugefügt werden dürfen, damit die benötigte Zeit sinkt, muss in die Ungleichung

$$94^n > (x + 94)^{n-m} \tag{4.9}$$

eingesetzt werden. Dabei ist n die Länge des zu erratenden Passwortes und m die Zahl der Zeichen, um die das Wort verkürzt wurde.
In dem zuvor verwendeten Beispiel könnten also neben '2013' noch 1340 andere Wörter beliebiger Länge hinzugefügt werden.

Der verwendete Code orientiert sich stark an dem in Abschnitt 4.3 beschriebenen Algorithmus. Anstelle eines Char-Arrays wird nun allerdings ein C++ String-Array verwendet. Dieses wird in Zeile 14 um die vermuteten Wörter, die im Array *charlist* gespeichert sind, erweitert. Da die Einträge des String-Arrays nicht notwendiger Weise gleich lang sind, muss nach jedem Versuch der komplette String *guess* erneuert werden.

```
void getGuess(const vector<string>& charlist){
    vector<string> chars{"a", "b", "c", "d", "e", "f", "g", "h", "i",
                         "j", "k", "l", "m", "n", "o", "p", "q", "r",
                         "s", "t", "u", "v", "w", "x", "y", "z", "A",
                         "B", "C", "D", "E", "F", "G", "H", "I", "J",
                         "K", "L", "M", "N", "O", "P", "Q", "R", "S",
                         "T", "U", "V", "W", "X", "Y", "Z", "0", "1",
                         "2", "3", "4", "5", "6", "7", "8", "9", "!",
                         "@", "#", "$", "%", "^", "&", "*", "(", ")",
                         "-", "_", "+", "=", "~", "`", "[", "]", "{",
                         "}", "|","\\", ":", ";","\"", "'", "<", ">",
                         ",", ".", "?", "/", " "};

    for(unsigned long k=0; k<charlist.size(); chars.push_back(
    charlist[k++]));

    int i;
    int guessc[MAX_SIZE] = {0};
    string guess{""};

    for(i = 1; i < MAX_SIZE ; guessc[i++] = -1);

    while(1)
    {
        guess="";

        i=0;
        while(guessc[i]==CHAR_COUNT)
        {
            guessc[i]=0;
            guessc[++i]+=1;
        }

        i=0;
        while(guessc[i]!=-1 && i< MAX_SIZE)
        {
            guess+=chars[guessc[i]];
            ++i;
        }

        printf("%s\n",guess.c_str());

        ++guessc[0];
    }
}
```

cforce.cpp

4.6 Ergebnis

Obwohl sich die drei vorgestellten Angriffe in ihrer Vorgehensweise stark von einander unterscheiden, wird in allen drei die gleiche Schwäche des verwendeten Hashalgorithmus ausgenutzt. Dabei handelt es sich um die Geschwindigkeit mit der es möglich ist Hashwerte zu berechnen.

"[...] because MD4 was designed to be exceptionally fast, it is 'at the edge' in terms of risking successful cryptanalytic attack." (Rivest 1992, S. 20)

Die extrem hohe Geschwindigkeit ist eine der größten Schwächen der MD4-Familie. Anstatt einen Angriff zu erschweren, ermöglicht sie es, sehr billig und schnell erfolgreiche Angriffe durchzuführen. Auch eine Parallelisierung stellt für einen Angreifer keine Herausforderung dar.
Viele Webseiten, darunter auch Dropbox, verwenden dennoch bis heute[8] den einerseits gebrochenen und andererseits wegen seiner Geschwindigkeit ungeeigneten MD5 Algorithmus um Passwörter zu sichern. Eine weit besser Wahl wäre bcrypt oder scrypt[9]. Beide wurden speziell für das Hashen und Speichern von Passwörtern entwickelt. Sie bieten viele Vorteile gegenüber MD5:

1. Der Rechenaufwand ist sehr hoch.
2. Es wird automatisch ein salt[10] verwendet um die Verwendung von Rainbowtables[11] zu verhindern.
3. Um eine Parallelisierung zu erschweren, wird bei scrypt standardmäßig 16MB Arbeitsspeicher angefordert. bcrypt fordert 4KB RAM an, scheitert jedoch daran, eine Parallelisierung effektiv zu verhindern.

Da die Berechnung eines scrypt-Hashwertes wesentlich mehr Rechenleistung erfordert als die eines MD5-Hashwertes, ist bessere Hardware nötig, um in gleicher Zeit gleich viele Berechnungen durchführen zu können. In Tabelle 10 sind die Kosten der benötigten Hardware ersichtlich, um in einem Jahr ein n-stelliges Passwort zu entschlüsseln (Bruteforce).(vgl. Percival 2009)

[8] 25.01.2017
[9] Bei scrypt handelt es sich, im Gegensatz zu bcrypt, um eine Passwort-basierte Schlüsselableitungsfunktion.
[10] Bei einem salt handelt es sich um eine der Eingabe hinzugefügte Zufallszahl.
[11] Sammlung bereits berechneter Hashwerte, die eine sehr schnelle Entschlüsselung aller gespeicherten Werte erlaubt

KDF	8 Buchstaben	8 Zeichen	10 Zeichen
DES CRYPT 1	<$1	<$1	<$1
MD5 1	<$1	<$1	$1.1Tsd.
MD5 CRYPT 1	<$1	$130	$1.1Mio.
PBKDF2 (100ms) 1	<$1	$18Tsd.	$160Mio.
bcrypt (95ms) 1	$4	$130Tsd.	$1.2Mrd.
scrypt (64ms) 1	$150	$4.8Mio.	$43Mrd.
PBKDF2 (5.0s) 1	$29	$920Tsd.	$8.3Mrd.
bcrypt (3.0s) 1	$130	$4.3Mio.	$39Mrd.
scrypt (3.8s)	$610Tsd.	$19Mrd.	$175Bio.

Tabelle 10: Geschätzte Hardwarekosten um ein Passwort in einem Jahr zu entschlüsseln (vgl. Percival 2009, S. 19)

5 Schluss

Resümee
Hashfunktionen sind dank ihrer nützlichen Eigenschaften ein wichtiger Bestandteil der Kryptographie und Informatik im Allgemeinen. Für sie gibt es unzählige Einsatzmöglichkeiten, was ein Problem darstellen kann. Je nach Einsatz muss eine Hashfunktion unterschiedliche Kriterien erfüllen. Wie Rivest bei der Beschreibung seiner Ziele für MD4 erläutert, sollte eine Funktion, die für die Signaturen von Daten verwendet wird, möglichst schnell Arbeiten. Abschnitt 4 zeigt, dass aber eben jene an sich gute Eigenschaft der Funktion in fast allen Angriffen auf Passwörter ausgenutzt wird. Durch die hohe Geschwindigkeit des Algorithmus ist es möglich viele Millionen Passwörter pro Sekunde zu testen. Für die Sicherung von Passwörtern sollte, wie zu sehen ist, eine speziell dafür designete Funktion, wie zum Beispiel bcrypt oder scrypt, zum Einsatz kommen.
Auch im LAN spielen Hashwerte keine unerhebliche Rolle. So verwendet Windows seit Windows NT den NTLM Hash, um eine Authentifizierung zu ermöglichen. Auch ein NTLM Hash ist schnell zu berechnen und damit nur bedingt für die Sicherung von Passwörtern geeignet. Dem in Abschnitt 4 beschriebenen Sinn eine Passwortliste zu hashen widersprechend, ist es bei Windows jedoch nicht nötig den Hash vor dem Login zu entschlüsseln. Die als “pass the hash” bekannte Methode nutzt diese Schwäche aus, um sich nur mit dem Hashwert bei einem Service zu authentifizieren. Sie ist die Grundlage vieler gezielter Angriffe auf Unternehmen, da sie die Kontrolle des gesamten Netzwerks stark vereinfacht/ermöglicht.

Wie zu sehen ist, stellt oft nicht die Hashfunktion selbst sondern ihre Implementierung eine Schwachstelle dar.

Ausblick
Auch in Zukunft wird es möglich sein, als sicher geltende Hashfunktionen zu brechen. Das ist jedoch kein Problem, da es mit relativ geringem Aufwand möglich ist, gebrochene Hashfunktionen auszutauschen, bevor Schäden entstehen. Da Computer immer schneller werden, muss wohl auch die Länge künftiger Hashwerte steigen. Bereits heute werden 512 Bit lange Hashwerte verwendet. Ein MD5-Hashwert hat hingegen nur eine Länge von 128 Bit. Wie Quantencomputer diese Entwicklung beeinflussen werden, ist noch nicht abzusehen.

Literatur

Boer, Bert den und Antoon Bosselaers (1993). *Collisions for the compression function of MD5*. URL: https://www.esat.kuleuven.be/cosic/publications/article-143.pdf (besucht am 21. 01. 2017) (siehe S. 23).

Brzeźniak, Zdzisław und Tomasz Zastawniak (1999). *Basic Stochastic Processes* (siehe S. 33).

Dobbertin, Hans (1995). *Cryptanalysis of MD4*. URL: https://pdfs.semanticscholar.org/e878/3ef34f73773fb679aef58d987839313e1b3c.pdf (besucht am 21. 01. 2017) (siehe S. 19).

Dörner, Stephan (2011). *Hacker auf dem Rückzug*. URL: http://www.handelsblatt.com/technik/it-internet/it-sicherheit-16-4-milliarden-euro-schaden-durch-internetkriminalitaet/4604434-2.html (besucht am 15. 01. 2017) (siehe S. 38).

Greveler, Ulrich (1998). *Kryptoanalyse der MD4-Familie*. URL: https://www.ruhr-uni-bochum.de/nds/greveler/papers/DiplomArbeit.pdf (besucht am 28. 01. 2017) (siehe S. 14, 24).

Percival, Colin (2009). *scrypt: A new key derivation function*. URL: http://www.tarsnap.com/scrypt/scrypt-slides.pdf (besucht am 28. 01. 2017) (siehe S. 42 f.).

Petritsch, Helmut (2006). *Aktuelle Angriffe auf SHA und MD5*. URL: http://petritsch.co.at/download/Attacken_auf_MD5uSHA1.pdf (besucht am 28. 01. 2017) (siehe S. 12 f.).

Rivest, Ronald L. (1992). *MD5 Message-Digest Algorithm*. RFC 1320. RFC Editor. URL: https://www.rfc-editor.org/rfc/rfc1320.txt (siehe S. 16, 20, 22, 42).

Schneier, Bruce (1996). *Angewandte Kryptographie*. Freiburg: Prof. Dr. Günther Müller (siehe S. 4, 6, 20, 22, 24, 26 f.).

— (2000). *A self-study course in block-cipher cryptoanalysis*. URL: https://www.schneier.com/academic/paperfiles/paper-self-study.pdf (besucht am 21. 01. 2017) (siehe S. 12).

— (2005). *SHA-1 Broken*. URL: https://www.schneier.com/blog/archives/2005/02/sha1_broken.html (besucht am 25. 01. 2017) (siehe S. 27).

— (2015). *SHA-1 Freestart Collision*. URL: https://www.schneier.com/blog/archives/2015/10/sha-1_freestart.html (besucht am 25. 01. 2017) (siehe S. 27).

Stinson, Douglas R. (1995). *Cryptography: theory and practice*. Florida: CRC Press (siehe S. 4, 6 f., 15).

Wang, Xiaoyun, Dengguo Feng, Xuejia Lai und HongboYu (2004). *Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD*. URL: https:

//eprint.iacr.org/2004/199.pdf (besucht am 28. 01. 2017) (siehe S. 19, 23).
Weisstein, Eric W. (2007). *Hash Function.* URL: http://mathworld.wolfram.com/HashFunction.html (besucht am 11. 12. 2016) (siehe S. 5).

Abbildungsverzeichnis

Alle Abbildungen und Grafiken wurden selbst mit den Programmen MATLAB und Photoshop erstellt.

Tabellenverzeichnis

A Passwortstatistiken

yahoo-stats.txt

```
Total words:         440712
Unique words:         341233 (77.43 %)

Word frequency, sorted by count, top 20
+------------------------------+
|   Word    | Count | Of total |
+------------------------------+
| 123456    |  1655 | 0.3755 % |
| password  |   775 | 0.1759 % |
| welcome   |   434 | 0.0985 % |
| ninja     |   333 | 0.0756 % |
| abc123    |   249 | 0.0565 % |
| 123456789 |   221 | 0.0501 % |
| 12345678  |   207 | 0.047 %  |
| sunshine  |   204 | 0.0463 % |
| princess  |   202 | 0.0458 % |
| qwerty    |   172 | 0.039 %  |
| writer    |   162 | 0.0368 % |
| monkey    |   162 | 0.0368 % |
| freedom   |   161 | 0.0365 % |
| michael   |   160 | 0.0363 % |
| 111111    |   159 | 0.0361 % |
| iloveyou  |   140 | 0.0318 % |
| password1 |   138 | 0.0313 % |
| shadow    |   134 | 0.0304 % |
| baseball  |   133 | 0.0302 % |
| tigger    |   132 | 0.03 %   |
+------------------------------+


Base word (len>=3) frequency, sorted by count, top 20
+------------------------------+
|    Word    | Count | Of total |
+------------------------------+
| password   |  1264 | 0.2868 % |
| welcome    |   520 | 0.118 %  |
| abc        |   455 | 0.1032 % |
| qwerty     |   441 | 0.1001 % |
| monkey     |   407 | 0.0924 % |
| love       |   394 | 0.0894 % |
| money      |   380 | 0.0862 % |
| ninja      |   378 | 0.0858 % |
| jesus      |   371 | 0.0842 % |
```

freedom	363	0.0824 %
writer	357	0.081 %
sunshine	343	0.0778 %
princess	329	0.0747 %
michael	324	0.0735 %
angel	303	0.0688 %
jordan	282	0.064 %
shadow	276	0.0626 %
blue	274	0.0622 %
associated	271	0.0615 %
dragon	260	0.059 %

Length frequency, sorted by length, full table

Length	Count	Of total
1	115	0.0261 %
2	66	0.015 %
3	298	0.0676 %
4	2741	0.6219 %
5	5314	1.2058 %
6	79180	17.9664 %
7	65269	14.8099 %
8	118219	26.8245 %
9	65174	14.7883 %
10	53921	12.235 %
11	21070	4.7809 %
12	21583	4.8973 %
13	2682	0.6086 %
14	1561	0.3542 %
15	931	0.2112 %
16	710	0.1611 %
17	429	0.0973 %
18	376	0.0853 %
19	301	0.0683 %
20	342	0.0776 %
21	64	0.0145 %
22	41	0.0093 %
23	36	0.0082 %
24	43	0.0098 %
25	21	0.0048 %
26	24	0.0054 %
27	19	0.0043 %
28	29	0.0066 %
29	18	0.0041 %
30	15	0.0034 %
31	10	0.0023 %

```
|     32 |     13 | 0.0029 %  |
|     33 |      7 | 0.0016 %  |
|     34 |      9 | 0.002 %   |
|     35 |      9 | 0.002 %   |
|     36 |      7 | 0.0016 %  |
|     37 |      3 | 0.0007 %  |
|     38 |      8 | 0.0018 %  |
|     39 |      4 | 0.0009 %  |
|     40 |      3 | 0.0007 %  |
|     41 |      3 | 0.0007 %  |
|     42 |      3 | 0.0007 %  |
|     43 |      4 | 0.0009 %  |
|     44 |      2 | 0.0005 %  |
|     45 |      1 | 0.0002 %  |
|     46 |      2 | 0.0005 %  |
|     47 |      4 | 0.0009 %  |
|     48 |      4 | 0.0009 %  |
|     49 |      5 | 0.0011 %  |
|     50 |      1 | 0.0002 %  |
|     51 |      3 | 0.0007 %  |
|     52 |      2 | 0.0005 %  |
|     53 |      1 | 0.0002 %  |
|     54 |      1 | 0.0002 %  |
|     55 |      1 | 0.0002 %  |
|     56 |      2 | 0.0005 %  |
|     59 |      3 | 0.0007 %  |
|     61 |      2 | 0.0005 %  |
|     71 |      1 | 0.0002 %  |
|     75 |      1 | 0.0002 %  |
|     76 |      1 | 0.0002 %  |
+----------------------------+
```

```
Charset frequency, sorted by count, full table
+-------------------------------------------------------------------------+
|           Charset            | Count  | Of total  |   Count/keyspace    |
+-------------------------------------------------------------------------+
| lower-upper-numeric-symbolic | 438655 | 99.5333 % |   4617.421052631579 |
| lower-upper-numeric          | 428104 | 97.1392 % |   6904.903225806452 |
| lower-numeric-symbolic       | 401820 | 91.1752 % |   5823.478260869565 |
| lower-numeric                | 394599 | 89.5367 % |  10961.083333333334 |
| lower-upper-symbolic         | 154999 | 35.1701 % |  1823.5176470588235 |
| lower-upper                  | 152676 | 34.643 %  |   2936.076923076923 |
| lower-symbolic               | 147686 | 33.5108 % |  2503.1525423728813 |
| lower                        | 145810 | 33.0851 % |   5608.076923076923 |
| upper-numeric-symbolic       |  31495 | 7.1464 %  |   456.4492753623188 |
| upper-numeric                |  31105 | 7.0579 %  |   864.0277777777778 |
| numeric-symbolic             |  26111 | 5.9247 %  |   607.2325581395348 |
| numeric                      |  25943 | 5.8866 %  |              2594.3 |
```

```
| upper-symbolic               |   1817 | 0.4123 %  |  30.796610169491526 |
| upper                        |   1769 | 0.4014 %  |   68.03846153846153 |
| symbolic                     |     12 | 0.0027 %  | 0.36363636363636365 |
+------------------------------------------------------------------------+

Charset frequency, sorted by count/keyspace, full table
+------------------------------------------------------------------------+
|           Charset            | Count  | Of total  |   Count/keyspace    |
+------------------------------------------------------------------------+
| lower-numeric                | 394599 | 89.5367 % |  10961.083333333334 |
| lower-upper-numeric          | 428104 | 97.1392 % |   6904.903225806452 |
| lower-numeric-symbolic       | 401820 | 91.1752 % |   5823.478260869565 |
| lower                        | 145810 | 33.0851 % |   5608.076923076923 |
| lower-upper-numeric-symbolic | 438655 | 99.5333 % |   4617.421052631579 |
| lower-upper                  | 152676 | 34.643 %  |   2936.076923076923 |
| numeric                      |  25943 | 5.8866 %  |              2594.3 |
| lower-symbolic               | 147686 | 33.5108 % |  2503.1525423728813 |
| lower-upper-symbolic         | 154999 | 35.1701 % |  1823.5176470588235 |
| upper-numeric                |  31105 | 7.0579 %  |   864.0277777777778 |
| numeric-symbolic             |  26111 | 5.9247 %  |   607.2325581395348 |
| upper-numeric-symbolic       |  31495 | 7.1464 %  |   456.4492753623188 |
| upper                        |   1769 | 0.4014 %  |   68.03846153846153 |
| upper-symbolic               |   1817 | 0.4123 %  |  30.796610169491526 |
| symbolic                     |     12 | 0.0027 %  | 0.36363636363636365 |
+------------------------------------------------------------------------+


Hashcat mask frequency, sorted by count, top 20
+-------------------------------------------------------------------------+
|           Mask           | Count | Of total |     Count/keyspace     |
+-------------------------------------------------------------------------+
| ?l?l?l?l?l?l             | 40476 | 9.1842 % | 0.00013102600496518507 |
| ?l?l?l?l?l?l?l?l?l?l?l?l | 32257 | 7.3193 % |  3.380210905412972e-13 |
| ?l?l?l?l?l?l?l           | 28987 | 6.5773 % | 3.6090245367870606e-06 |
| ?l?l?l?l?l?l?d?d?l?l?l?l | 20162 | 4.5749 % | 1.4282365098383907e-12 |
| ?l?l?l?l?l?l?l?l?l       | 16086 | 3.65 %   |  2.962701740545428e-09 |
| ?d?d?d?d?d?d             | 12517 | 2.8402 % |               0.012517 |
| ?l?l?l?l?l?l?l?l?d?d     | 12485 | 2.8329 % |  5.978631182385002e-10 |
| ?l?l?l?l?l?l?l?d?l?l?l?l | 10550 | 2.3939 % | 2.8743895940564167e-13 |
| ?l?l?l?l?l?l?l?l?l?l     | 10261 | 2.3283 % |  7.268691016492276e-11 |
| ?l?l?l?l?l?l?l?d?d       | 10206 | 2.3158 % |  1.270697361660356e-08 |
| ?l?l?l?l?d?d?d?d?l?l?l?l |  8347 | 1.894 %  |  3.997087263065087e-12 |
| ?l?l?l?l?l?d?d           |  8133 | 1.8454 % | 6.8451667550963796e-06 |
| ?l?l?l?l?l?l?d           |  7083 | 1.6072 % |  2.292857973041817e-06 |
| ?l?l?l?l?l?d?d?d?l?l?l?l |  7077 | 1.6058 % | 1.3034340555663305e-12 |
| ?l?l?l?l?l?d             |  6416 | 1.4558 % |  5.400047940575233e-05 |
| ?l?l?l?l?l?l?d?d?d       |  6053 | 1.3735 % | 1.9594337584105773e-08 |
| ?l?l?l?l?l?l?l?l?d       |  6008 | 1.3632 % |  2.877021717562602e-09 |
| ?l?l?l?l?l?l?d?d?d?d     |  5823 | 1.3213 % | 1.8849798075705917e-09 |
```

```
| ?l?l?l?l?d?d             |  5718 | 1.2974 % |  0.0001251269213262841 |
| ?l?l?l?l?l?d?d?d?d       |  5658 | 1.2838 % |  4.762074695725478e-08 |
+--------------------------------------------------------------------+
Words that didn't match any ?l?u?d?s mask: 2057 (0.4667 %)


Charset distribution of characters in beginning and end of words (len>=6)
+-------------------------------------------------------------------------------+
| Charset\Index | 0 (first char) |     1     |     2     |    -3     |    -2    |
+-------------------------------------------------------------------------------+
| lower         | 79.4688 %      | 83.8378 % | 83.3835 % | 69.6585 % | 54.2577 |
| upper         | 6.7023 %       | 2.0805 %  | 1.9888 %  | 1.6456 %  | 1.4156  |
| digits        | 13.5951 %      | 13.8699 % | 14.4682 % | 28.2925 % | 43.9022 |
| symbols       | 0.2339 %       | 0.2118 %  | 0.1595 %  | 0.4033 %  | 0.4245  |
+-------------------------------------------------------------------------------+


Total characters:         3654043
Unique characters:         91
Top 50 characters:         aeio1rnslt20mcd3hub9y478gp5k6wfjvzxqASMELRNTBC!DIP

Character frequency, sorted by count, top 20
+-------------------------------+
| Character | Count  | Of total |
+-------------------------------+
| a         | 271119 | 7.4197 % |
| e         | 258645 | 7.0783 % |
| i         | 172879 | 4.7312 % |
| o         | 172669 | 4.7254 % |
| 1         | 171921 | 4.705 %  |
| r         | 171521 | 4.694 %  |
| n         | 167971 | 4.5969 % |
| s         | 161790 | 4.4277 % |
| l         | 138939 | 3.8023 % |
| t         | 131587 | 3.6011 % |
| 2         | 124344 | 3.4029 % |
| 0         | 107723 | 2.948 %  |
| m         | 106098 | 2.9036 % |
| c         |  95087 | 2.6022 % |
| d         |  94190 | 2.5777 % |
| 3         |  88686 | 2.4271 % |
| h         |  85940 | 2.3519 % |
| u         |  77109 | 2.1102 % |
| b         |  75338 | 2.0618 % |
| 9         |  75068 | 2.0544 % |
+-------------------------------+


Symbol frequency, sorted by count, top 20
```

Symbol	Count
!	3868
@	2921
_	1691
*	1598
#	1402
-	1082
;	644
&	609
.	527
%	356
	183
?	102
/	85
,	68
(	49
)	43
:	27
]	26
[	23
\	5

B Anfangsverteilung

$$\lambda^{yahoo} = \{0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000,$$
$$0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000,$$
$$0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000,$$
$$0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000,$$
$$0.000000, 0.060130, 0.000000, 0.034717, 0.060357, 0.005219, 0.008169, 0.000000,$$
$$0.002723, 0.000227, 0.046969, 0.003404, 0.001135, 0.007034, 0.004311, 0.001588,$$
$$1.709960, 4.051626, 1.701565, 1.005645, 1.004057, 0.838870, 0.808011, 0.886747,$$
$$0.831609, 0.885159, 0.000227, 0.001361, 0.000000, 0.002950, 0.000000, 0.001588,$$
$$0.047650, 0.495108, 0.437928, 0.385966, 0.332417, 0.159061, 0.196273, 0.268202,$$
$$0.240747, 0.173583, 0.358511, 0.222821, 0.313357, 0.570214, 0.215560, 0.091443,$$
$$0.370537, 0.058542, 0.292708, 0.582013, 0.351704, 0.067618, 0.094166, 0.169272,$$
$$0.039708, 0.073744, 0.072610, 0.002496, 0.000227, 0.000227, 0.003177, 0.004538,$$
$$0.000681, 5.589818, 5.605929, 5.283950, 3.963813, 1.902830, 2.694050, 2.936839,$$
$$2.761667, 1.902830, 3.963132, 2.807275, 3.529970, 6.739095, 2.468506, 1.207591,$$
$$4.453929, 0.600619, 3.300568, 7.818485, 4.036877, 0.520975, 1.085516, 2.437193,$$
$$0.337862, 0.747200, 0.641916, 0.000000, 0.000000, 0.000000, 0.003404, 0.000000\}$$

$$\lambda^{rockyou} = \{0.000000, 0.000000, 0.000000, 0.000000, 0.000007, 0.000000, 0.000000, 0.000000,$$
$$0.000021, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000,$$
$$0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000,$$
$$0.000000, 0.000000, 0.000007, 0.000000, 0.000000, 0.000000, 0.000000, 0.000000,$$
$$0.002037, 0.052929, 0.007498, 0.052992, 0.046317, 0.004736, 0.005105, 0.001932,$$
$$0.052197, 0.002511, 0.155593, 0.012394, 0.011759, 0.026377, 0.042816, 0.013251,$$
$$6.321085, 4.405412, 2.887085, 1.510863, 1.547605, 1.452460, 1.096687, 1.057135,$$
$$1.137257, 1.492542, 0.004171, 0.008320, 0.017624, 0.007477, 0.001276, 0.006403,$$
$$0.059345, 0.565745, 0.561448, 0.530475, 0.457083, 0.220036, 0.233253, 0.272142,$$
$$0.255139, 0.223935, 0.497500, 0.351030, 0.504963, 0.717118, 0.265516, 0.093436,$$
$$0.388929, 0.039357, 0.355535, 0.687351, 0.440979, 0.039956, 0.104372, 0.136880,$$
$$0.032598, 0.091442, 0.060719, 0.017164, 0.002999, 0.002490, 0.003836, 0.024376,$$
$$0.006416, 4.893538, 4.384446, 4.461367, 3.482039, 1.972612, 1.927962, 2.206695,$$
$$2.101563, 1.984476, 3.976965, 3.209353, 3.983019, 6.032706, 2.375441, 0.909319,$$
$$3.351875, 0.274046, 2.897721, 5.834855, 3.558165, 0.392333, 0.947643, 1.288212,$$
$$0.337583, 0.892190, 0.614071, 0.003152, 0.000879, 0.000251, 0.018078, 0.000000\}$$

Begleitprotokoll

Name des Schülers: Sebastian Hirnschall
Thema der Arbeit:
Funktionsweise und Schwachstellen von kryptographischen Hashfunktionen
Name der Betreuungsperson: Mag. Christian Filipp

Datum	Vorgangsweise, ausgeführte Arbeiten, verwendete Hilfsmittel, aufgesuchte Bibliotheken,...
01.12.2016	Empirischer Teil der Arbeit fertiggestellt
05.12.2016	Abschnitt 2.1 Kryptographische Hashfunktionen -Definition -Beweis -Literatur: Stinson und Schneier -Bibliothek: TU Wien
06.12.2016	Änderung an Abschnitt 2.1
08.12.2016	Abschnitt 2.1 Kryptographische Hashfunktionen -Theorem -Beweis -Abschnitt 2.1 fertiggestellt
10.12.2016	-Bruteforce Code optimiert -Änderungen an Abschnitt 2.1
12.12.2016	-Literaturverzeichnis mit Biblatex
15.01.2017	Abschnitt 3.1 MD4 -Beschreibung des MD4 Algorithmus
16.01.2017	Abschnitt 3.1 MD4 -Angriffe -Schwachstellen
17.01.2017	Abschnitt 3.2 MD5 -MD5 Schritte -MD5 Änderungen zu MD4 (Rivest-rfc1320)
18.01.2017	Abschnitt 3.1 MD4 -Beispielrechnung - händisch

22.01.2017	Eine frühe Fassung -SHA -Bitoperatoren -Abschnitt 2 geändert -Unterschiede SHA MD5 -Markow-Kette Änderung -Bruteforce Code erklärt -Anhang -Kapitel Verwendung gestrichen um 60Tsd. Zeichen nicht zu überschreiten

Datum	Besprechungen mit der betreuenden Lehrperson, Fortschritte, offene Fragen, Probleme, nächste Schritte
22.06.2016	über Sommer: -Theorieteil (Bücher aus TU und Vorträge) -praktischer Teil über Vorträge Aufbau: 1. Hashfunktionen (was? + Programmcode, Funktionen, Schwachstellen) 2. Angriffsmethoden (Vergleich, Muster der PW) Analye bereits im Laufen ca.50:50 (Theorie - Praxis)
29.09.2016	-Vorstellen der LaTex-Vorlage (selbst erstellt) – ist O.K. -Besprechen der Zitierweise: direkte Zitate (eingerückt und kursiv) =>genaues Zitieren in Literaturverzeichnis -Indirekte Zitate: mit Zusatz („Vergleiche“) -selbst erstellte Abbildungen + Code mit Hinweis darauf -In Kopfzeile reicht Hauptkapitel -Formulierung mit „man“ und „ich“ möglichst vermeiden (ausgenommen mathematische Erklärungen) -Zeichenzählen von PDF zu normalem Text (da sonst Sourcecode nicht mitgezählt werden würde)

12.01.2017	-Definitionen, Beweis und Protokoll (5 Schritte) direkt aus Buch übernommen (Hinweis darauf in Fußnote) -bereits besprochen: kryptographischen Hashfunktionen und praktischer Teil (Hash entschlüsseln + Theorie zu Markow-Ketten) -noch zu erledigen: Funktionsweise von Hash-Funktionen und Hashfunktionen im Vergleich -bis 31.1.: Endfassung ->Rückmeldung bis 3.2. -letzter Abgabetermin: 17.2. (in 3 fach gebundener Ausfertigung) -Termin zur Besprechung der Präsentation: 2.3. / 13: 20 (Columbus)

Die Arbeit hat eine Länge von 57 890 Zeichen.

Datum, Ort

Sebastian Hirnschall

Selbstständigkeitserklärung

Ich versichere, dass ich diese Vorwissenschaftliche Arbeit selbstständig angefertigt, keine anderen als die angegebenen Hilfsmittel benutzt und alle aus ungedruckten Quellen, gedruckter Literatur oder aus dem Internet im Wortlaut oder im wesentlichen Inhalt übernommenen Formulierungen und Konzepte gemäß den Richtlinien wissenschaftlicher Arbeiten zitiert, durch Fußnoten gekennzeichnet bzw. mit genauer Quellenangabe kenntlich gemacht habe.

Datum, Ort

Sebastian Hirnschall