Hashing

— Seminararbeit: „Effiziente Programmierung in C“ —

Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von: Alexander Koglin
E-Mail-Adresse: alexanderkoglin@hotmail.de
Matrikelnummer: 6218451
Studiengang: Mathematik (B.Sc.)

Betreuer: Julian Kunkel
E-Mail-Adresse: kunkel@dkrz.de

Hamburg, den 29.03.2013

Inhaltsverzeichnis

1 Einleitung

1.1 Definition

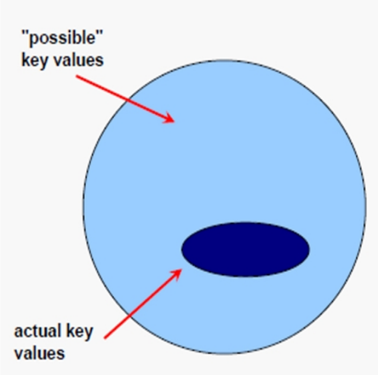

Abbildung 1.1: Mächtigkeitsveranschaulichung

Eine „Hashfunktion“[1] (hash (en.)=zerhacken (de)) oder auch „Streuwertfunktion“ ist im Allgemeinen ein Algorithmus, der Elemente einer Menge sogenannter „Schlüssel“ (engl.: keys) auf eine Menge sogenannter „Hashwerte“ (engl. Hashes) abbildet. Dabei hat die Menge der Hashes eine kleinere Mächtigkeit als die Menge der Schlüssel und zumeist haben die Hashwerte eine feste Länge. Die Hashwerte sind meist Element aus einer Teilmenge der natürlichen Zahlen. Da also unter Umständen zwei Schlüssel auf denselben Hash abgebildet werden – was man „Kollision“ nennt – ist die Abbildung nicht bijektiv (also nicht gleichzeitig injektiv [Injektivität: Jeder Hashwert wird höchstens einmal als Funktionswert angenommen.] und surjektiv=[Surjektivität: Jeder Hashwert wird mindestens einmal als Funktionswert angenommen]). Die meisten Hash-Algorithmen sind nicht injektiv und nicht notwendigerweise surjektiv. Diese Surjektivität zu beweisen, ist äußerst schwierig! Kollisionen sind nicht beliebt, da sie Probleme bei vielen Anwendungen bereiten können; deshalb gibt es verschiedene Möglichkeiten damit umzugehen. Am besten ist es natürlich, wenn die Hash-Funktion so für den jeweiligen Anwendungsfall konstruiert ist, dass bei Schlüsseln sehr selten Kollisionen auftreten können.

Eine gute Hash-Funktion muss nicht nur die Definition erfüllen, sondern zudem in der Regel noch einfach und schnell berechenbar sein. Idealerweise ist sie injektiv – dann nennt man sie ideale Hashfunktion, da keine Kollisionen auftreten können. Für bestimmte Schlüsselmengen existieren auch perfekte Hashfunktionen, d.h. ohne Kollisionen. Zudem zeichnen sich gute Hash-Funktionen durch einen gewissen Grad an Chaos aus: Eine kleine Änderung eines Schlüssels bewirkt eine große Änderung am Hashwert.

Bedenke: Hashfunktionen sind Algorithmen oder sogar Funktionen und erzeugen daher keine Zufallswerte. Allerdings unterscheiden sich die einzelnen Algorithmen durch verschiedenen Grade von Pseudozufall.

[1] http://de.wikipedia.org/wiki/Hashfunktion

1.2 Beispiel eines Hash-Algorithmus‘: DJBX33X

DJBX33X (auch bekannt als „cdb“ http://cr.yp.to/cdb/cdb.txt) ist ein vom bekannten Kryptographen Daniel J. Bernstein entwickelter Hash-Algorithmus, der für Datenbanken eingesetzt wird. Das erste „X“ steht für „mal“, das zweite für „XOR“. „(hash«5)+hash“ ist gerade „hash*33“. Man hat ersteres aus Effizienzgründen gewählt, da „x«5“ eine sehr schnelle Möglichkeit für die Multiplikation mit 2^5 ist. Gerade „33“ wurde aus statistischen Gründen genommen. „5381“ ist empirisch gewählt. Eine etwas schlechtere Variation benutzt statt der Exponentiation die Addition; dann heißt es DJBX33A. Andere bekannte Algorithmen sind PJW und K&R.

Listing 1.1: DJBX33X-Code

```
1  uint32_t hash(string str)
2  {
3      uint32_t hash = 5381;
4
5      for (int i = 0; i < str.Length; i++)
6      {
7          hash = ((hash << 5) + hash) ^ (int)str[i];
8      }
9      return hash;
10 }
```

1.3 Kryptographische Hash-Funktionen

Bekannte kryptographische Hash-Funktionen sind MD5 (von 1992; Message Digests Digest = Hashwert), SHA-1 (1995); jedoch sind diese nicht mehr sicher, obwohl sie noch vielfach eingesetzt werden. Wenn es sich für einen Angreifer lohnt, kann er absichtlich Kollisionen erzeugen oder auf ein Urbild bzw. einen Schlüssel schließen. Folglich sollte man zurzeit als sicher geltende Algorithmen benutzen, so etwa SHA-2 (welches eine Oberbezeichnung für SHA-224, SHA-256, SHA-384 und SHA-512 ist, wobei die Zahlen die Bitlänge des Hashwerts ist), WHIRLPOOL oder RIPEMD-160 (von 1996).

Die ideale kryptographische Hash-Funktion muss folgende Eigenschaften erfüllen: Eine kryptographische Hash-Funktion muss weitaus mehr darauf ausgelegt sein, chaotisch zu reagieren: Eine kleine Änderung am Schlüssel muss eine große Änderung am Hashwert bedeuten, damit man nicht Rückschlüsse auf den Schlüssel ziehen kann! Ferner muss die Einweg-Eigenschaft stark sein: Man darf keine Nachricht effizient erzeugen dürfen, die einen bestimmten Hash besitzt. Desweiteren dürfen Kollisionen nur schwierig zu erzeugen sein, damit nicht absichtlich oder unabsichtlich zwei Datenpakete den gleichen Hash haben (Kollsionsresistenz). Dies wird in der Kryptographie verwendet, um übertragene Nachrichten zu verifizieren. Jedoch gehen diese Forderungen mit einem erhöhten CPU-Rechenaufwand einher im Vergleich zu herkömmlichen Hash-Funktionen.

2 Effizienzvergleich von kryptographischen Hashfunktionen

Um einen Effizienzvergleich anzustellen, ist es zwingend, festzulegen, was „Effizienz“ hier bedeutet. Sicherlich spielt die „Geschwindigkeit“ dabei eine große Rolle. Das Problem hieran ist, dass man nicht ohne weiteres die Geschwindigkeit der einzelnen Hash-Algorithmen miteinander vergleichen kann. Dies wäre unseriös, da sie von weiteren variablen Größen abhängt: [1] Zunächst wäre da die CPU: Diese lässt sich abermals in Architektur, Anzahl der Kerne, und Cache-Größe unterteilen. Nachfolgend werden die Geschwindigkeitsunterschiede für bekannte Hash-Algorithmen (bis auf CRC32 alles kryptographische Algorithmen) auf drei verschiedenen CPUs (Intel Core 2 1.83 GHz unter 32-bit Vista; AMD Opteron 8354 2.2 GHz unter Linux; Intel Pentium 4 (Prescott) 2.93 GHz) dargestellt. Dabei sind die Messgrößen zum einen „Datenvolumen pro Zeit“ und „Prozessortakte pro Byte“. Nur ein Kern wurde jeweils benutzt.[2]

	Core 2		Opteron		P4	
Algorithmus	**MiB/s**	**Takte/Byte**	**MiB/s**	**Takte/Byte**	**MiB/s**	**Takte/Byte**
CRC32	253	6.9	382	5.5	381	7.3
MD5	255	6.8	335	6.3	352	7.9
SHA-1	153	11.4	192	10.9	138	20.3
SHA-256	111	15.8	139	15.0	123	22.7
SHA-512	99	17.7	154	13.6	77	36.5
Tiger	214	8.1	328	6.4	182	15.3
Whirlpool	57	30.5	77	27.3	65	43.3
RIPEMD-160	106	16.5	143	14.6	101	27.8
RIPEMD-320	110	15.9	153	13.6	87	32.1
RIPEMD-128	153	11.4	210	10.0	211	13.2
RIPEMD-256	158	11.1	234	8.9	176	15.9

Tabelle 2.1: Unterschiede bei bekannten Hash-Algorithmen

Dann ist da noch das zu hashende Datenvolumen: Dabei treten Geschwindigkeitsunterschiede der Algorithmen beispielsweise beim „Padding“ (engl. „Auffüllen“) auf, was

[1] http://cr.yp.to/talks/2008.06.05/slides.pdf
[2] http://www.cryptopp.com/benchmarks.html

ein Auffüllen der Blöcke bei kleinen Datenvolumina.

Doch dies (CPU + Datenvolumen) reicht nicht zur Identifizierung einer effizienten Hash-Funktion aus! Denn es gibt sicherlich eine kryptographisch schwache Funktion, die diese Aufgaben besser erfüllt als eine stärkere. (Meist stehen langsamere Algorithmen nicht so stark in der „Schusslinie“ wie schnelle.)

Oft werden die Schlüssel mehrere tausend Male hintereinander gehasht, um eine höhere Sicherheit zu erlangen. Deshalb definieren wir, dass eine Hashfunktion als „sicher“ gilt, wenn die Hälfte der Runden nicht geknackt wurden.

Die folgende Abbildung vom eBASH-Projekt (ECRYPT Benchmarking of All Submitted Hashes) von Daniel J. Bernstein demonstriert die Effizienzunterschiede der jeweiligen Implementationen (verschiede Farben) auf verschiedenen Prozessoren bzw. Architekturen anhand des SHA-256-Hashes. Beim eBASH-Projekt kann jeder eine neue Hash-Funktion bzw. eine neue Implementation hochladen, die dann automatisch auf den CPUs getestet wird. Die Tests der einzelnen Implementationen einer bestimmten Funktion auf verschiedenen Systemen gibt es hier[3], den Vergleich der unterschiedlichen Hash-Funktionen/Implementationen der Takte pro Byte für verschiedene Datenvolumina auf einem bestimmten Prozessor gibt es hier [4].

2.1 Gedankenexperiment

Es existiert seit März 2008 eine Befehlssatzerweiterung für die Verschlüsselungschiffre AES (Advanced Encryption Standard) für Intel- und AMD-Prozessoren. Diese Erweiterung bringt einen enormen Geschwindigkeitsgewinn (falls die entsprechenden Befehle verwendet werden). Die folgende Graphik (ebenfalls von der eBASH Seite) für echo-256 [5] , einem auf AES basierenden Hash-Algorithmus[6], zeigt eine Reduzierung der Rechenzeit auf weniger als ein Viertel bei Verwendung der Erweiterungsbefehle! Wenn nun eine Erweiterung für einen bestimmten Hash-Algorithmus eingeführt werden sollte (, da man nicht alle Hash-Algorithmen mit den AES-Befehlen darstellen kann), veranschaulicht dies den ungefähren Effizienzgewinn! Dies wäre für Serverprozessoren wichtig, da Server-Mainbords zwar PCI-Express-x8-Anschlüsse für Grafikkarten besitzen, aber x16-Anschlüsse (auch PCI Express for Graphics (PEG) genannt) auf Desktopsystemen Standard sind, wodurch die Transferrate zwischen Grafikchip und Hauptspeicher entsprechend reduziert ist. Ferner ist nicht ausgeschlossen, dass das BIOS die Karte nicht richtig einbindet, da es als Server oft keine erwartet. Nicht zuletzt wäre dann noch das Problem der Stromversorgung: Die

[3] http://bench.cr.yp.to/primitives-hash.html

[4] http://bench.cr.yp.to/results-hash.html

[5] http://crypto.rd.francetelecom.com/ECHO/

[6] http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/documents/presentations/PEYRIN_ECHO-SHA3Conference2010_slides.pdf

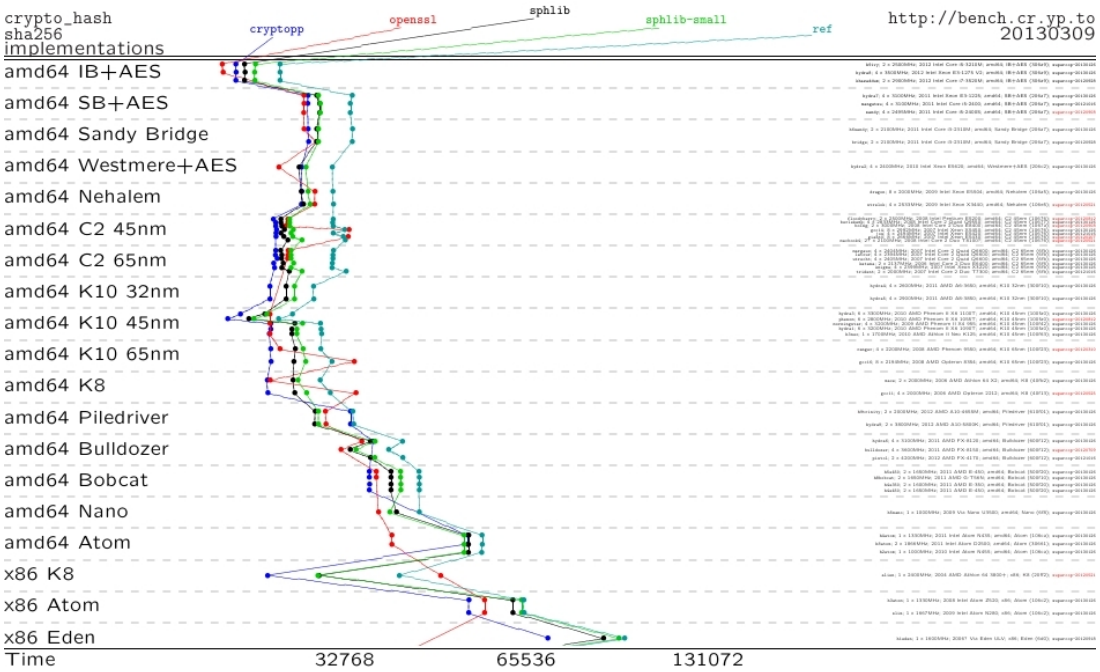

Abbildung 2.1: SHA-256: Effizienz verschiedener Implementationen auf verschiedenen CPUs [Quelle: http://bench.cr.yp.to/impl-hash/sha256.html, gekürzt]

x16-Anschlüsse liefern bis zu 75 Watt, während die x8er nur bis zu 25 Watt liefern. Somit ist der Einsatz von Highend-Grafikkarten, die mit ihrer Parallelleistung das 20 bis 30 fache an Hashes im Vergleich zu Multi-Core-CPUs erbringen, ausgeschlossen.[7]

[7] http://www.heise.de/ct/hotline/Grafikkarte-auf-dem-Serverboard-961527.html

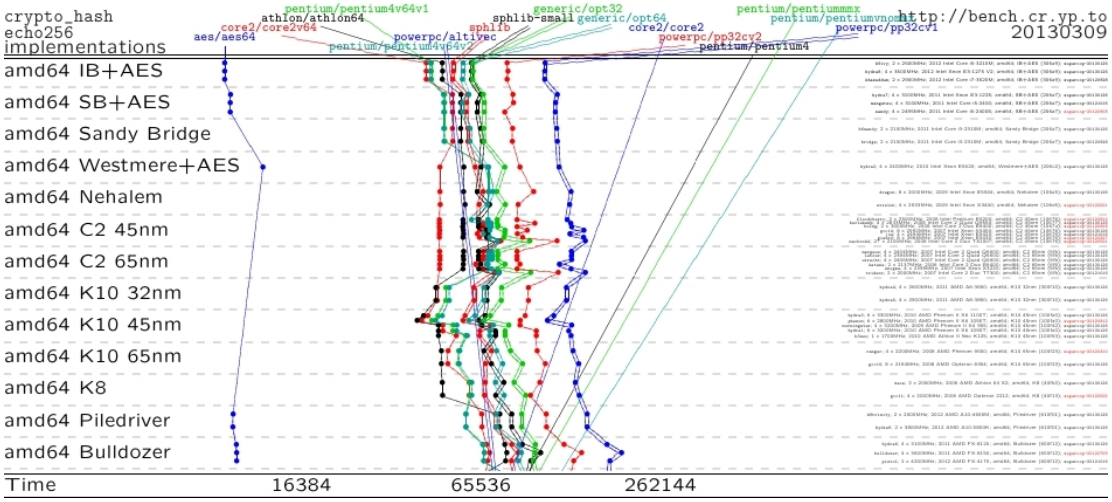

Abbildung 2.2: Echo-256: Veranschaulichung der AES-Erweiterungsstärke [Quelle: http://bench.cr.yp.to/impl-hash/echo256.html, gekürzt]

3 Anwendungen

Es gibt vielfältige Anwendungsmöglichkeiten für Hashing. Es kommt bei Prüfsummen zum Einsatz, in der Kryptographie für (Pseudo-)Zufallsgeneratoren, bei Hash-Listen und /-Bäumen und bei Hash-Tabellen etwa in Datenbanken.

Anwendung/Zweck	**Schlüssel**	**Typ**	**in Liste**
Duplikate löschen		dedup	Duplikate
Rechtschreibprüfung/ Fehler finden	Wort	Ausnahme	Wörterbuch
Browser/besuchte Seiten markieren	URL	Nachschlagen	besuchte Seiten
Schach/Remis feststellen	Brett	Nachschlagen	Positionen
Spamfilter/Spam löschen	IP-Adresse	Ausnahme	Spam
Whitelist-Filter	URL	Nachschlagen	Whitelist
Kreditkarten	Zahl	Ausnahme	gestohlene Karten
Data Mining/Datei-Vergleich	Zeilen der Datei	Vergl./Nachschl.	geminete Dateien

Tabelle 3.1: Einige Anwendungen, bei denen Hashing effizient angewendet werden kann

3.1 Prüfsummen

Prüfsummen erlauben es, Datenänderungen effizient zu bemerken und Dateien effizient zu identifizieren (Oft wird in diesem Zusammenhang der Hash „Fingerprint“ (engl. „Fingerabdruck“) genannt, da ein Hash die Daten fast verwechselungsfrei identifiziert, so wie der Fingerabdruck einen Menschen).

Dies spielt vor allem bei der Überprüfung der Datei-/Daten-Integrität eine wichtige Rolle, etwa um festzustellen, ob Daten bei der Übermittlung im Netzwerk beschädigt wurden oder Dateien im Speicher/der Festplatte verändert wurden. Jedoch werden Prüfsummen auch bei Mensch-Maschine-Interaktionen verwendet: Hier sind zum Beispiel ISBN (International Standard Book Number) bei Büchern und EAN bei Artikeln zu nennen! Manchmal muss ein Kassierer/eine Kassiererin die Artikelnummer in die Kasse eingeben. Hier können schon mal Zahlendreher vorkommen, welche schlimmstenfalls dazu führen, dass die Nummer eines anderen Artikels eingegeben wird! Besonders im Deutschen

wird die Einerziffer vor der Zehnerziffer genannt; dies kann dazu führen, dass man zuerst die Einerziffer eingibt. Durch Verwendung eines herkömmlichen Prüfsummenalgorithmus, der die einzelnen Summanden mit speziellen Gewichten versieht und aufsummiert, können die Dreher und Veränderungen im Allgemeinen effizient erkennen.

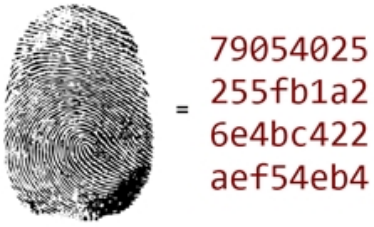

Abbildung 3.1: Fingerprint codinghorror.com/blog/2012/04/speed-hashing.html

Achtung: Herkömmliche Prüfsummen sind keine kryptographischen Verfahren! Das bedeutet, dass sie gegen Manipulationen nicht gerüstet sind. Deshalb sollte man sie nur dort verwenden, wo kein absichtlicher Schaden angerichtet werden kann. Nicht jede Hashfunktion ist für diesen Einsatz geeignet. Deshalb sollte man ansonsten kryptographische Hash-Algorithmen, die sich durch ihre Einweg-Eigenschaft auszeichnen, benutzen. Komplexere Prüfsummen-Algorithmen, wie beispielsweise CRC (zyklische Redundanzprüfung), das statt einfacher Summe von gewichteten Summanden Polynomdivision verwendet, erlauben bessere Erkennung von (zufälligen) Fehlern, denn aufgrund des Charakters von Hashing, nämlich der Reduktion der Mächtigkeit geht immer eine gewisse Ungenauigkeit einher, dass Kollisionen erzeugt werden. Einige komplexe Prüfsummen-Algorithmen erlauben sogar eine gewisse Fehlerkorrektur.

Anwendungsbeispiele: Im Peer2Peer-Bereich werden Prüfsummen zur Verifizierung und Identifizierung von Dateien eingesetzt. Ebenfalls als Session ID in Webanwendungen werden Hashes verwendet, wobei als Eingangsparameter typischerweise IP Adresse, Zeitmarke, etc. einfließen. Bei elektronischer Signalübertragung werden Prüfbits an die Pakete angehängt. Es wird die Prüfsumme der Daten berechnet und mit der mitgeschickten Prüfsumme verglichen; sind sie identisch, liegt höchstwahrscheinlich kein Übertragungsfehler vor, andernfalls muss das Paket noch einmal angefordert werden.

3.2 Statistik/Data Mining

Zu statistischen Zwecken reicht es nicht aus, zu wissen, welche Dateien identisch sind. Es ist von Interesse, ähnliche Dateien zu finden und/oder zu vergleichen.

Wir beschränken uns hier auf Dateien, die eine feste Struktur dahingehend aufweisen, dass eine Datei keine eingefügte Zeile gegenüber der anderen aufweist, da sonst Schwierigkeiten bei folgender Methode auftreten können (Es gibt überwiegend Dateien, die einen Zeilenumbruch oder kleine Änderungen im bestehenden Text aufweisen, was darin resultiert, dass die Hashwerte sehr unterschiedlich sind und sich bei eingefügten neuen Zeilen

verschieben. Das zeigt nicht nur die Schwächen dieser einfachen Methode, sondern auch bei der Benutzung von Hashing im Allgemeinen auf: Es bedarf größerer Rechenleistung, um eine höhere Genauigkeit zu erreichen (Abwägung: Effizienz vs. Effektivität).).

Ein einfacher Ansatz dies effizient durchzuführen ist, den Hashwert jeder Zeile der (Plaintext-)Datei zu berechnen. Dies ergibt einen Hash-Vektor. Wie wir vielleicht noch wissen, kann der Winkel zwischen zwei Vektoren bzw. der Kosinus des Winkels mithilfe des Skalarproduktes ermittelt werden: $cos(\alpha) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| * \|\vec{b}\|}$

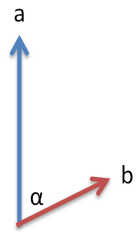

Abbildung 3.2: Vektoren und Skalarprodukt

Damit immer der kleinere Winkel genommen wird, was intuitiv als „Winkel zwischen den Vektoren“ bezeichnet wird, steht immer Nenner des Bruches das Produkt aus den (euklidischen) Normen der Vektoren. Dadurch liegt der Winkel zwischen 0 und 180 Grad bzw. 0 und Pi. Wenn nun der Kosinus nahe Null ist, sind die Vektoren sehr verschieden und damit vermutlich auch die Dateien; wenn der Kosinus nahe Eins ist, so sind die Vektoren ähnlich und damit auch die Dateien.

Es eine weitere Schwierigkeiten, die bei diesem einfachen Ansatz auftreten kann; die Dateien unterscheiden sich höchstwahrscheinlich in der Anzahl der Zeilen. Dies würde zu abweichenden Dimensionen der Vektoren führen. Sind beispielsweise zwei Dateien bis zum Ende von einer der beiden identisch, der anderen wurde allerdings noch etwas angehängt, so werden diese (unterschiedlichen) Dateien als identisch angesehen, wenn man einfach die Restdimensionen des kürzeren Vektors mit Nullen auffüllt. Die Lösung hierfür ist relativ einfach; man nimmt statt der Nullen einen gewissen Penalty-Wert, wodurch gewährleistet wird, dass das Skalarprodukt nicht Eins wird.

3.3 Hashing bei Passwörtern

Passwörter unverschlüsselt zu speichern, ist ein großes Sicherheitsrisiko! Oftmals verwenden Benutzer aus Faulheit die gleiche Kombination aus Benutzername/E-Mail-Adresse und Passwort bei mehreren Diensten. Dies bedeutet, wenn einer von solchen Diensten gehackt wird und die Angreifer die unverschlüsselten Kombinationen erlangen, können sie sich auch Zutritt zu anderen Diensten verschaffen; sie müssen nur noch herausfinden, welche diese sind. Wenn allerdings die Passwörter serverseitig gehasht vorliegen, kann ein Angreifer sich nicht ohne weiteres Zugang zu anderen Diensten verschaffen, da ihm das Klartextpasswort nicht bekannt ist.

Typischerweise werden kryptographische Hashfunktionen verwendet, die sich u.a. durch ihre Einwegseigenschaft auszeichnen. Ein Problem besteht darin, dass es unter Umständen nicht ausreicht, die Passwörter zu hashen: Wenn die Benutzer schwache Passwörter (die man erraten kann) oder etwa Wörter, die sich in Wörterbüchern auffinden lassen, verwenden, kann ein Angreifer per Brute Force alle Wörter bzw. Wörter, bei denen bekannt ist (z.B. aus gehackten nicht gehashten Datenbanken), dass sie oft als Passwort eingesetzt werden, durchprobieren/hashen und mit den Hashes aus der erbeuteten Datenbank vergleichen.

Ein weiteres Werkzeug, das Hacker anwenden können, ist der sogenannte Rainbowtable. Er ist eine Datenstruktur, die es ermöglicht, effizient zu einem gegebenen Hashwert ein Urbild (Es können ja mehrere Urbilder zum selben Hash führen) bestimmter Länge (und nur dieser Länge) zu finden. Diese Tabelle ist vorher erzeugt worden, mitunter sogar kollaborativ. Die Alternative, für alle möglichen Klartexte einer bestimmten Länge die Hashes zu berechnen und diese mit dem Hash in der Datenbank zu vergleichen, ist sehr ineffizient und bietet keine Möglichkeit, die aufgewendete Rechenleistung für zukünftige Versuche per Speichern und Abgleichen zu nutzen, da der benötigte Speicherbedarf immens ist (Groß- und Kleinschreibung und Ziffern ergeben bereits 62 Zeichen; bei einer Länge von 8 Zeichen haben wir dann 62^8= 218.340.105.584.896 Möglichkeiten und somit 62^8*(2*8) Bytes, was ungefähr 3,5 Petabytes sind.). Um einen Rainbowtable zu erstellen, wird eine zufällige Zeichenkette mit bestimmter Länge genommen und von dieser der Hashwert berechnet. Wie wir erinnern, ist die Länge des Hashes höchstwahrscheinlich unterschiedlich lang verglichen mit der Zeichenkette. Durch eine Reduktionsfunktion wird der erhaltene Hash in eine neue Zeichenkette umgewandelt, die wieder die gleiche Länge wie die ursprüngliche Zeichenkette besitzt. Diese beiden Schritte können nun sukzessive beliebig fortgesetzt werden. Schließlich bilden die einzelnen Werte eine Kette (engl.: Chain), wovon sowohl Anfangs- als auch Endwert gespeichert werden; die anderen Werte werden verworfen. So reduziert sich der Speicherbedarf, die Hashes einer Kette lassen sich trotzdem noch mit den beiden Werten reproduzieren. Damit wird der Hash Table gebildet. Dieser weist aus genannten Gründen einen relativ geringen Speicherbedarf auf; es wird dennoch ein vertretbarer Rechenaufwand in Echtzeit benötigt, um ein Urbild zu finden. Damit stellt der Hash Table eine gute Abwägung zwischen Rechenaufwand und Speicherbedarf dar. Diese Abwägung ist nicht eindeutig und kann deshalb schwanken, wobei je nach Kettenlänge entweder mehr Rechenaufwand und weniger Speicherbedarf (kurze Kette) oder weniger Rechenaufwand und mehr Speicherbedarf (lange Kette) benötigt wird. Bei einer Kettenlänge von n werden n Hash-Berechnungen benötigt, um das Urbild zu finden. Wenn der Angreifer nun einen Hash hat, zu dem er das ein Urbild finden möchte, muss er zuerst diesen Hash sukzessive per Reduktionsfunktion reduzieren und weiterhashen, bis ein Endhash einer Kette erreicht wird; nun weiß man welche Kette es ist und kann angefangen mit dem korrespondierenden Startwert wieder bis zum Wert vor dem gegebenen Hash die Kette reproduzieren: Dieser Wert ist das gesuchte Urbild und wird als Passwort funktionieren. Es ist keinesfalls sicher, dass man ein Urbild, geschweige denn ein Passwort, findet. Dies hängt von der Reduktionsfunktion ab, da nur die Hashes, die von der Reduktionsfunktion erzeugt werden, wiedergefunden werden

können.

3.3.1 Gegenmaßnahmen: Mache den Hash einzigartig und langsam

Eine Maßnahme, um einen Erfolg mittels Rainbow Tables zu vereiteln, ist die Benutzung eines „Salt“; dies macht die Erzeugung eines Rainbow Tables unrentabel, da dem Passwort vor dem Hashvorgang eine zufällige Zeichenkette, die natürlich zusammen mit dem Hash gespeichert werden muss, angehängt wird; dies ist das Salt. Es sollte mindestens 64 Bits lang sein! Nun müsste für jedes Salt ein eigener Rainbow Table erstellt werden. Da nicht mehr ein Rainbow Table verwendet werdet werden kann, um alles Passwörter abzudecken, ist es für einen Angreifer nicht rentabel, da die benötigten Rainbow Tables großen Rechenaufwand und auch einen nicht vernachlässigbaren Platzaufwand nach sich ziehen.

Achtung: Ein Salt erhöht bei kurzen Passwörtern die Sicherheit nicht erheblich; nur der Rechenaufwand wird vergrößert.

Eine weitere Maßnahme besteht darin, dass die Größe des Rainbow Tables in Zusammenhang mit der Länge der Passwörter und wie oben schon erwähnt mit dem Hash-Algorithmus steht; man muss also den Benutzern eine Mindestlänge (8 Zeichen bei allen möglichen Symbolen und 12 Zeichen bei Groß- und Kleinbuchstaben und Zahlen) verbindlich vorschreiben. Dabei sollte man bedenken, dass allein schon pures Brute Force für Passwörter, die aus Groß- und Kleinbuchstaben, Zahlen und anderen Symbolen bestehen, mindestens 9 Zeichen lang sein.

Zudem kann man mit einer Hash-Kette den Hash-Vorgang hinreichend langsam gestalten. Wenn ein Angreifer die Datenbank noch nicht erlangt hat, kann man durch Erhöhung der Runden (s.o.) die Sicherheit erhöhen, ohne den Benutzer zum Passwort-Wechsel zu zwingen.

Es sollte auf keinen Fall bei sicherheitskritischen (Web-)Anwendungen versucht werden, die Hashalgorithmen selbst zu implementieren bzw. eigene Algorithmen zu entwickeln, da sie Schwachstellen aufweisen können.

3.3.2 Implementationen: bcrypt und PBKDF2

bcrypt und PBKDF2 sind Implementationen bzw. weitverbreitete Standards für kryptographische Hash-Funktionen speziell zum Passwort-Hashing. Der Hauptunterschied zu MD5 oder SHA ist, dass letztere mit dem Ziel entwickelt wurden, sehr effizient zu sein, da sie besonders zum Prüfen der Datei-Integrität benutzt werden. Dies bedeutet, dass Brute Force und Rainbow Tables sehr effizient sind. Ein 6 –stelliges Passwort (alle Symbole),

das mit MD5 gehasht wurde, zu knacken, dauert auf aktuellen Grafikkarten (Die ein Vielfaches an Hashes pro Sekunde berechnen können! AMD 7970: über 8 Giga-Hashes pro Sekunde! vs. Zwei-stellige Mega-Hashes pro Sekunde bei CPUs)[1] unter eine Minute, bei Groß- und Kleinschreibung und Zahlen sogar nur drei Sekunden!

Deshalb sind bcrypt und PBKDF2 darauf ausgelegt, das Hashen hinreichend ineffizient zu gestalten: Es gibt einen vom Benutzer einstellbaren Kosten-Faktor bzw. einstellbare Anzahl an Hashvorgängen. Falls in der Zukunft die CPU/GPU-Rechenleistung steigt, kann der Kosten-Faktor nachträglich angepasst werden.

Anwendungen, die bcrypt oder PBKDF2 benutzen:

- WPA und WPA2
- MacOS X Mountain Lion für die Benutzerpasswörter
- Dateisystemverschlüsselung bei Android und iOS
- 1Password und LastPass
- OpenOffice
- WinZip

3.3.3 Hash-Chain/Hash-Kette

Bei der Hash-Kette wird, wie die Bezeichnung bereits andeutet, eine kryptographische (Einweg-)Hash-Funktion auf eine Zeichenkette angewendet. Dies eröffnet die Möglichkeit, Einmalpasswörter (engl.: one-time passwords) aus einem gegebenen Schlüssel zu erstellen. Diese kann man dann in einer unsicheren Umgebung, beispielsweise um sich als Benutzer einem Server zu authentifizieren, anwenden: Hierzu speichert der Server den tausendsten Hash (999 malige Hintereinanderanwendung des Hashes auf den Hash vom Schlüssel). Der Benutzer authentifiziert sich, indem den 999sten Hash übermittelt. Dies stellt in der Tat eine sichere Authentifizierung aufgrund der Einweg-Eigenschaft der kryptographischen Hashfunktionen dar. Dieser 999ste Hash wiederum wird danach vom Server verwendet, den gespeicherten 1000sten Hashwert zu ersetzen. Das nächste Mal muss der Benutzer den 998sten Hash übermitteln, um sich zu authentifizieren. Derart fortgesetzt ergeben sich insgesamt 999 verschiedene Einmalpasswörter. Ein Angreifer kann nicht durch Mitschneiden der Übertragung des Hashes vom Benutzer zum Server effizient Zugang zum System erlangen, da bereits ein anderer Hash aktiv ist. Es schützt jedoch nicht vor Man-in-the-Middle-Angriffen; dafür ist zusätzlich eine verschlüsselte Verbindung erforderlich.

[1] http://www.codinghorror.com/blog/2012/04/speed-hashing.html

3.4 Hash-Liste und (binärer) Hash-Baum

3.4.1 Hash-Liste

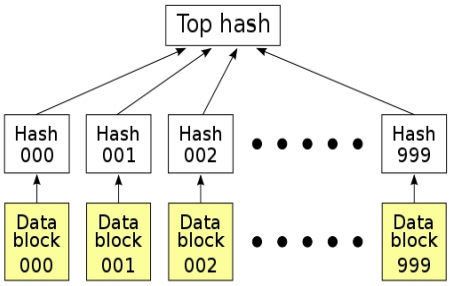

Abbildung 3.3: Hash-Liste
(David Göthberg)

Eine Hash-Liste[2] ist[3] eine Datenstruktur ist eine Liste aus den jeweiligen Hashes aus Blöcken einer Datei. Meistens wird zusätzliche noch ein sogenannter „Top Hash“ gespeichert, der den Hash der Hash-Liste darstellt. Die Hashliste ist eine Weiterentwicklung des normalen Hashings: Die Datei-Integrität kann nicht nur überprüft werden, sondern es kann auch festgestellt werden, welche Blöcke eventuell beschädigt sind.

3.4.2 (binärer) Hash-Baum

Ein Hash-Baum[4] (engl.: hash tree) ist ebenfalls eine Datenstruktur, die vom Kryptologen Ralph Merkle erfunden wurde und die wie der Name bereits andeutet, baumartig/gestuft aufgebaut ist: Daten werden in bestimmte (meist gleich große) Blöcke aufgeteilt. Für diese wird jeweils der Hash berechnet; es ergibt sich eine Hashliste (s.o.), welche sukzessive abermals in Teile aufgeteilt wird, von denen jeweils der Hash berechnet wird. Schließlich erhält man einen so genannten „Top Hash“/ „Root Hash“. Hash-Bäume können verwendet werden, um Datei- bzw. Block-Integrität zu überprüfen; dazu wird bei Dateisystemen der Top Hash zu einer bestimmten Datei gespeichert. Bei Downloads wird der Top Hash getrennt von einer vertrauenswürdigen Quelle übermittelt; durch Vergleich mit dem selbst errechneten Top Hash kann man effizient die Integrität überprüfen. Zum Einsatz kommt eine kryptographische Hash-Funktion, etwa SHA-1, Whirlpool oder Tiger, um möglichst gut geschützt vor Übertragungsfehlern und böswilligen Manipulationen von Blöcken zu sein. Letzteres kann ein Problem bei Peer2Peer-Netzwerken sein: Ein Angreifer kann Blöcke erzeugen, die gleichen Hashes haben wie die Originalblöcke. Dominiert dieser Angreifer das Netzwerk,

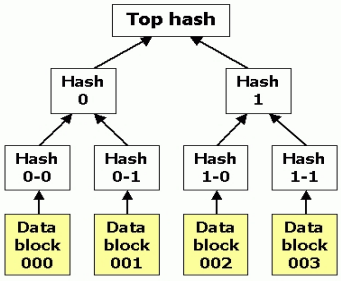

Abbildung 3.4: binärer Hash-Tree
(David Göthberg)

[2] http://en.wikipedia.org/wiki/Hash_list
[3] http://en.wikipedia.org/wiki/File:Hash_list.svg
[4] http://de.wikipedia.org/wiki/Hash_tree

hat beispielsweise eine hohe zur Verfügung stehende Bandbreite, werden die fehlerhaften Blöcke in die heruntergeladene Datei integriert; sie ist beschädigt und womöglich sogar total unbrauchbar. Ein weiterer Vorteil ist, dass der Top Hash im Idealfall (also bei Benutzung einer idealen kryptographischen Hashfunktion) eindeutig ist. Damit lassen sich effizient Dateien auf der Festplatte bzw. allgemein im Netzwerk identifizieren. [5] Die meisten Implementierungen benutzen eine binäre Struktur.

Die Anwendungen im Überblick:

- Peer2Peer: Integrität bei erhaltenen Datei-Blöcken überprüfen
- ZFS-Dateisystem von Sun Microsystems
- Git
- Bitcoin
- einige NoSQL-Systeme (etwa Apache Cassandra)
- allgemein Suche von Worten in Texten

3.5 Hash-Tabelle

3.5.1 Das Wörterbuchproblem

Beim Wörterbuchproblem haben wir eine Menge von Objekten, die über einen Schlüssel eindeutig identifiziert werden können. Das Problem besteht nun darin, eine Datenstruktur zu finden, die die effiziente Ausführung der drei wesentlichen Operationen „Suchen“, „Einfügen“ bzw. „Hinzufügen“ und „Löschen“ gewährleistet. Die Lösung des Problems hängt von verschiedenen Faktoren ab: Nicht nur von der Speicherart, d.h., ob die Datenstruktur im Arbeitsspeicher angelegt wird oder ob sie sich auf der Festplatte befindet, sondern auch von der Art der angewendeten Operationen, d.h., ob überwiegend gesucht wird oder mehr „Einfügen“ und „Löschen“ im Vordergrund stehen oder die Operationenarten etwa gleichverteilt sind. Weitere Faktoren sind die Ausführungsreihenfolge der Operationen und die Möglichkeit, weitere Operationen durchzuführen, wie etwa „Vereinigung“ oder „Schnitt“.

[5] http://de.wikipedia.org/wiki/Tiger-Tree_Hash

3.5.2 Hash-Tabelle, ein assoziatives Array

Die Hash-Tabelle [6] ist eine Datenstruktur, bei der Schlüsselwerte/Urbilder auf Indizes eines Arrays abgebildet werden; dies erfolgt durch Anwendung eines Hash-Algorithmus'. Sie zeichnet sich durch ihre hohe Zugriffsgeschwindigkeit, da keine Vergleichsoperationen durchgeführt werden, und im Vergleich zu Bäumen einfachen Programmierbarkeit aus. Allerdings: Hash-Tabellen basieren auf Arrays und sind deshalb schwierig erweiterbar! Hinzu kommt, dass es keinen brauchbaren Ansatz gibt, die Einträge in der Tabelle in einer bestimmten Reihenfolge zu erreichen! In diesem Fall wären Bäume besser geeignet.

Falls keine Kollisionen vorkommen, haben wir eine konstante Komplexität, d.h. die Zugriffszeit dürfte auch konstant sein.

Das Anwendungsgebiet liegt beim Datenmanagement, etwa beim „verteilten Hash Table“ (engl.: distributed hash table (DTH)): Hier können große Datenbanken auf ein Netzwerk verteilt werden. Ein weiteres Beispiel ist die Verteilung von Ressourcen (Traffic und Speicherbedarf) auf mehrere Server. Um einen Cache (z.B. Browsercache) zu implementieren, werden meist Hash-Tabellen genommen. Auch bei Compilern unter anderem zum Zweck der Bildung von Symboltabellen, die jedem Symbol/Bezeichner im Quelltext den Ort, den Datentyp und ggf. einen Pointer zuordnet, wird der Hash-Table verwendet.

3.5.3 Kollisionen

Wir erinnern uns: Eine Kollision tritt auf, wenn zwei Urbilder auf den gleichen Hashwert gemappt werden. Bei einem Hash-Table werden zwei Urbilder auf den gleichen Index des Arrays abgebildet. Mit zunehmendem Füllgrad, erhöht sich die Anzahl der Kollisionen.

„Geburtstagsproblem“

„Wie hoch ist die Wahrscheinlichkeit, dass zwei Personen im gleichen Raum am gleichen Tag Geburtstag haben?“

Auf das Hashing bezogen, werden Kollisionen sicherlich auftreten, wenn hinreichend viele Hashings durchgeführt werden. Man braucht also viel Speicher, wenn man möglichst viele Kollisionen vermeiden möchte. Deshalb sollte man einen effizienten Weg suchen, wie man mit ihnen umgeht.

Kollisionsbehebung

Es gibt viele Wege, Kollisionen zu beheben. Im Folgenden befindet sich eine Auswahl:

- **Open Addressing/Hashing mit offener Adressierung** Zum einen wäre da das „Open Addressing“, das bereits 1953 bei IBM entwickelt wurde. Wie auch hier

[6] http://en.wikipedia.org/wiki/Hash_table

die Bezeichnung andeutet, werden einfach leere Speicherstellen dazu verwendet, die Schlüssel, deren Hashes auf Speicherstellen zeigen, die bereits von anderen Schlüsseln verwendet werden, aufzunehmen. Es gibt verschiedene Arten des Open Adressing, die sich in Hinsicht auf das Suchen freier Speicherstellen unterscheiden:

  - Beim **„Linear Probing/Linearen Sonderieren“** untersucht man ab der Speicherstelle, auf die der Hash zeigt, jede folgende Speicherstelle darauf, ob sie schon einen Schlüssel trägt oder noch frei ist.
  - Beim **„Quadratic Probing/Quadratischen Sondieren“** unterscheidet sich gegenüber dem „Linear Probing“ in dem Punkt, dass anstatt jeder nachfolgenden Speicherstelle nur die $h + i^2 - te$ (i=1,2,3,...) überprüft wird, wobei h die Stelle ist, auf die der Hash zeigt. Das Problem hierbei ist, dass mit zunehmendem Füllgrad die Schlüsselballungen größer werden, was in durchschnittlich sehr langen Sondierlängen resultiert.

- **Seperate Chaining/Hashing mit Verkettung** Dann gibt es da noch das „Seperate Chaining“/„Hashing mit Verkettung“ , ebenfalls von IBM 1953 entwickelt. Speicheradressen können mehrfach vergeben werden; sie heißen hier ´"Buckets“ (engl. Behälter). Man verwendet Liste für Schlüssel gleichen Hashes. Dadurch kann der Füllgrad 100 überschreiten. Es bedeutet aber auch, dass zwei Suchen durchgeführt werden müssen: Einmal, um den richtigen Bucket zu finden, und dann, um das richtige Element in der Liste zu finden. Schlimmstenfalls landen alle Schlüssel im gleichen Bucket, was Ineffizienz beim Suchen nach sich zieht. Separate Chaining braucht mehr Speicher als Open Addressing, aber jenes unterstützt kein effizientes Löschen. Wenn die Anzahl der Elemente nicht bekannt ist, sollte man Seperate Chaining Open Adressing vorziehen, da der Füllgrad höher liegen kann.

- **Double Hashing** Das Double Hashing funktioniert wie das Open Addressing; nur wird zum Sondieren eine weitere Hash-Funktion genommmen.

- **Dynamisches Hashing** Wie gesagt, wird mit zunehmendem Füllgrad die Wahrscheinlichkeit für Kollisionen größer. Beim Dynamischen Hashing wird als Lösung die Größe des Tables erhöht. Es werden spezielle Hash-Funktionen verwendet, die sich bei einer Änderung der Donmain so verhalten, dass die alten Adressen beibehalten werden.

3.5.4 Effizienzvergleich

Wie ersichtlich ist, ist beim Hash Table die Effizienz am höchsten und es besteht eine konstante Komplexität bei allen drei Operationen unabhängig von der Anzahl der gespeicherten Schlüssel, solange man keine Kollisionen miteinbezieht. Dies erklärt die große Verwendungshäufigkeit des Hash Tables.

Zustandekommen der Komplexität:

	Suchen	Einfügen	Löschen
ungeordnetes Array	$\mathrm{O}\big(n\big)$	$\mathrm{O}\big(n\big)$	$\mathrm{O}\big(n\big)$
geordnetes Array	$\mathrm{O}\big(log(n)\big)$	$\mathrm{O}\big(n\big)$	$\mathrm{O}\big(n\big)$
ungeordnete Liste	$\mathrm{O}\big(n\big)$	$\mathrm{O}\big(n\big)$	$\mathrm{O}\big(n\big)$
geordnete Liste	$\mathrm{O}\big(log(n)\big)$	$\mathrm{O}\big(log(n)\big)$	$\mathrm{O}\big(log(n)\big)$
balancierter binärer Baum	$\mathrm{O}\big(log(n)\big)$	$\mathrm{O}\big(log(n)\big)$	$\mathrm{O}\big(log(n)\big)$
Hash Table	$\mathrm{O}\big(1\big)$	$\mathrm{O}\big(1\big)$	$\mathrm{O}\big(1\big)$

Tabelle 3.2: Effizienzvergleich bei Speicherstrukturen

- **ungeordnete Liste:** Es handelt sich um lineares Suchen: Bestenfalls ist das erste Element der Liste das gesuchte, schlechtestenfalls ist es das letzte. Somit ist die Komplexität $n/2$, also $\mathrm{O}\big(n\big)$. Beim Einfügen und Löschen hängt es nicht von n ab, bleibt also bei $\mathrm{O}\big(n\big)$.

- **geordnete Liste:** Das intuitive Verfahren mit der sukzessiven Halbierung des passenden Intervalls impliziert eine logarithmische Komplexität, d.h. $\mathrm{O}\big(log(n)\big)$. Hinweis: Im Vortrag wurde die Komplexität der geordneten Liste mit der der verlinkten Liste verwechselt und O(n) angemerkt.

- **ungeordnetes Array:** verhält sich wie ungeordnete Liste

- **geordnetes Array:** verhält sich beim Suchen wie geordnete Liste, beim Einfügen und Löschen jedoch müssen alle nachfolgenden Elemente verschoben werden. Damit ergibt sich $\mathrm{O}\big(n\big)$.

- **balancierter binärer Baum:** Der Algorithmus beginnt an der Wurzel und vergleicht bei jedem Knoten, ob der gesuchte Schlüssel gefunden wurde, andernfalls folgt er den Verzweigungen. „Balanciertheit“ bedeutet, dass der Baum gleichmäßig gefüllt ist, d.h. der Weg von der „Wurzel“ zu den „Blättern“ ist gleich lang. Im schlimmsten Fall kann der Baum entartet sein, d.h. nur eine Kette/Liste (s.o.) bilden. „binär“ bedeutet, dass es immer nur zwei Verzweigungen pro Knoten gibt. Damit verhält sich der balancierte binäre Baum wie eine geordnete Liste.

- **Hash Table:** Hier hängt das Suchen, Einfügen und Löschen nicht von der Anzahl der gespeicherten Schlüssel ab, da die Speicherstelle mit der Hash-Funktion berechnet wird. Jedoch sollte man beachten, dass man keine überdimensionierte verwendet. Mit Kollisionen hängt die Zugriffszeit beim Seperate Chaining von den resultierenden Sondierungslängen ab; die Zeit ist proportional zur Probenlänge zuzüglich einem konstanten Summanden, nämlich die Hashzeit. Damit nähert sich die Komplexität der beim binären Baum.

3.5.5 Komplexitätsangriff

Auch hier kann ein böswilliger Angriff erfolgen: Dadurch, dass die Behebung von Kollisionen mitunter viel Zeit in Anspruch nehmen kann, könnte ein Angreifer, dem die benutzte Hash-Funktion bekannt ist, absichtlich einen kollidierenden Schlüssel vom Server anfordern, wodurch die CPU schon bei kleiner Bandbreite stark ausgelastet werden kann; es handelt sich also um einen „Denial of Service“-Angriff (DoS).

Obwohl dieser Angriffsvektor schon lange Zeit bekannt ist, sind einige Programmiersprachen immer noch verwundbar, obwohl der Angriff mit einfachen Mitteln erschwert werden kann, etwa durch eine Rechenzeitbegrenzung für die einzelne Hash-Berechnung inklusive Kollisionsbehebung. [7]

[7] http://events.ccc.de/congress/2011/Fahrplan/attachments/2007_28C3_Effective_DoS_on_web_application_platforms

4 Implementierungen

Viele Bibliotheken existieren sowohl für die Verwendung von Hash-Alogorithmen als auch für die Erstellung von Hash-Tabellen und ähnlichem. Neben der Glib (s.u.) für Hash Tables gibt es noch andere Bibliotheken, etwa „ghthash“ oder „CC's hashtable“ (in C), und viele weitere in C++ geschriebene[1]. Da die Glib wohl die bekannteste ist, werde ich sie exemplarisch vorstellen.

4.1 LibTomCrypt (libtom.org)

Die LibTomCrypt [2] ist eine Bibliothek von vielen kryptographischen Methoden, darunter sowohl Einweg-Hash-Funktionen wie MD4, MD5, SHA-1, SHA-224/256/512, TIGER-192, RIPE-MD 128/160/256/320 und WHIRLPOOL als auch Pseudo-Zufallsgeneratoren, Public-Key-Algorithmen und den PKCS #5 Standard zur Passwort-Verschlüsselung.

4.2 Glib

Die Glib ist eine bekannte Bibliothek, die nicht nur auf Hash-Algorithmen (wie MD5, SHA-1 und SHA-2569 und Datenstrukturen (Bäume, Listen, Hash Table [Die Funktion dafür lautet GHashTable]) beschränkt ist: Sämtliche Funktionen, deren Implementationen einen großen Aufwand erfordern, sind enthalten, beispielsweise zum Arbeiten mit Dateien, Strings, Threads.

Bei der Verwendung von GHashTable muss man bedenken, dass weder Schlüssel noch Hashwerte kopiert werden, wenn sie eingefügt werden. Deshalb sollten temporäre Zeichenketten mit g_strdup() kopiert werden, bevor sie eingefügt werden. Desweiteren sollte man überprüfen, wenn dynamisch alloziierte Werte gelöscht bzw. überschrieben werden, ob der Speicher auch freigegeben wird.

[1] http://attractivechaos.wordpress.com/2008/08/28/comparison-of-hash-table-libraries/
[2] http://libtom.org/?page=features&newsitems=5&whatfile=crypt

Beispiel für die Anwendung von Glib:
http://www.ibm.com/developerworks/linux/tutorials/l-glib/section5.html

Listing 4.1: Glib-Beispiel

```
1  #include <glib.h>
2  int main(int argc, char** argv)
3  {
4  GHashTable* hash = g_hash_table_new(g_str_hash,
       ↪ g_str_equal);
5  g_hash_table_insert(hash, "Virginia", "Richmond");
       ↪ g_hash_table_insert(hash, "Texas", "Austin");
6  g_hash_table_insert(hash, "Ohio", "Columbus");
7  printf("There are %d keys in the hash\n",
       ↪ g_hash_table_size(hash));
8  printf("The capital of Texas is %s\n",
       ↪ g_hash_table_lookup(hash, "Texas"));
9  gboolean found = g_hash_table_remove(hash, "Virginia");
10 printf("The value 'Virginia' was %sfound and removed\n",
       ↪ found ? "" : "not ");
11 g_hash_table_destroy(hash);
12 return 0;
13 }
14 ***** Output *****
15 There are 3 keys in the hash
16 The capital of Texas is Austin
17 The value 'Virginia' was found and removed
```

Listing 4.2: Bsp.: Einfügen und Ersetzen von Werten

```
1
2  #include <glib.h>
3  static char* texas_1, *texas_2;
4  void key_destroyed(gpointer data) {
5  printf("Got a key destroy call for %s\n", data == texas_1 ?
       ↪ "texas_1" : "texas_2");
6  }
7  int main(int argc, char** argv) {
8  GHashTable* hash = g_hash_table_new_full(g_str_hash,
       ↪ g_str_equal, (GDestroyNotify)key_destroyed, NULL);
9  texas_1 = g_strdup("Texas");
10 texas_2 = g_strdup("Texas");
11 g_hash_table_insert(hash, texas_1, "Austin");
12 printf("Calling insert with the texas_2 key\n");
13 g_hash_table_insert(hash, texas_2, "Houston");
14 printf("Calling replace with the texas_2 key\n");
```

```
15 g_hash_table_replace(hash, texas_2, "Houston");
16 printf("Destroying hash, so goodbye texas_2\n");
17 g_hash_table_destroy(hash);
18 g_free(texas_1);
19 g_free(texas_2);
20 return 0; }
21
22 ***** Output *****
23 Calling insert with the texas_2 key
24 Got a key destroy call for texas_2
25 Calling replace with the texas_2 key
26 Got a key destroy call for texas_1
27 Destroying hash, so goodbye texas_2
28 Got a key destroy call for texas_2
```

- **g_hash_table_insert:** Wenn der Schlüssel bereits existiert, wird der Wert ersetzt, aber nicht der Schlüssel.
- **g_hash_table_replace:** Beide (Schlüssel und Wert) werden ersetzt.

5 Fazit

Abschließend lässt sich sagen, dass Hashing erlaubt, bei vielen seiner Anwendungen eine größere Effizienz zu erreichen. Diese Anwendungen reichen von der Integritätsprüfung bei Dateien/Daten/Nachrichten und unter Umständen Fehlerkorrektur über Dateiidentifizierung und Pseudozufallszahlengeneratoren zur sicheren Speicherung von Passwörtern und Speicherung in Datenstrukturen (Hash-Liste, Hash-Baum und Hash-Table) im Allgemeinen. Dabei helfen Implementationen wie LibTomCrypt und die Glib weiter. Dabei ist vor allem der Hash-Table als in Datenbankensystemen weit verbreitetes Instrument, das uns also im täglichen Leben unbewusst vielfach begegnet, hervorzuheben. Jedoch muss man auf Kollisionen achtgeben.

(weitere Literatur: http://www.cs.princeton.edu/courses/archive/fall08/cos521/hash.pdf
http://www.ma.rhul.ac.uk/static/techrep/2009/RHUL-MA-2009-18.pdf http://www.hit.bme.hu/ but
mac-sig.pdf www.latech.edu/ box/ds/chap11.ppt http://www.cs.princeton.edu/ rs/AlgsDS07/10Hashi
http://en.wikipedia.org/wiki/Cryptographic_hash_function http://de.wikipedia.org/wiki/Rainbow_Ta

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst und keine anderen, als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internetquellen – benutzt habe, die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Optional: Ich bin mit der Einstellung der Bachelor-Arbeit in den Bestand der Bibliothek des Fachbereichs Informatik einverstanden.

Hamburg, den 29.03.2013 .....