Effizientes Programmieren in C

— Hashing —

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Paulus Böhme
E-Mail-Adresse:	1boehme@informatik.uni-hamburg.de
Matrikelnummer:	6313217
Studiengang:	Software- und Systementwicklung

Betreuer: Michael Kuhn

Hamburg, den 27.03.2014

Inhaltsverzeichnis

1 Einleitung

Das Thema Hashing ist in der Informatik weit verbreitet, wenn nicht sogar ein Muss für jeden Programmierer. Es wird in fast allen Sparten der Informatik genutzt: in der Datenbankverwaltung, Kryptographie und im Bereich der Prüfsummen.
Aus diesen Gründen habe ich dieses Thema für das Seminar Effizientes Programmieren in C gewählt. Ich will mit dieser Ausarbeitung die Wichtigkeit, Einsetzbarkeit und Einfachheit von Hashing in der Programmiersprache C näher erklären und zeigen, wie diese zu implementieren ist. Hierbei muss man beachten, dass es in den Standard-Bibliotheken von C keine Datenstrukturen wie z.B. Hashtabellen gibt, bzw. keine vordefinierten Hashfunktionen. Darum werde ich im Folgenden einerseits zeigen, wie man mit den gegebenen Mitteln von C Hashing betreiben kann und anderseits werde ich eine vorimplementierte Bibliothek näher erläutern bzw. werde ich versuchen zu zeigen, welcher der Implantationen und Datenstrukturen effizienter sind. Als Voraussetzung für diese Ausarbeitung lege ich Grundkenntnisse in der Informatik und vor allem in der Programmiersprache C fest und werde daher bekannte Datenstrukturen und Begriffe nur gelegentlich erklären.

2 Hashing

Zunächst wird der theoretische Teil des Hashings erläutert. Hierzu werden die mathematischen Seiten des Themengebietes veranschaulicht, um die Funktionsweise von Hashing verständlich zu machen.

2.1 Grundlegendes

Der Definition nach ist Hashing die Abbildung von beliebig langen Werten auf Werte mit einer festen Länge. Zudem kann man aber auch sagen, dass Hashing die Abbildung von Werten aus einem Universum auf einen Wert aus einer begrenzten Teilmenge ist. Dies bedeutet, dass wir beliebig viele Schlüssel haben, z.B. Zahlen, Wörter oder andere Datensätze, welche auf eine begrenzte Menge von natürlichen Zahlen abgebildet werden können(Die Zahl Null gehört in diesen Fall mit in die Menge der natürlichen Zahlen).

$$h = K \rightarrow S$$

h ist hierbei der Hashwert, K die Menge an Schlüsseln, d.h. die Werte, die gehasht werden sollen, und S die Menge an möglichen Hashwerten. Dabei ist immer zu beachten, dass

$$|K| \geq |S|$$

d.h. dass die Menge an möglichen Hashwerten immer kleiner gleich der Menge an vorhanden Schlüsseln ist. Bei dieser Abbildung ist es natürlich klar, dass einige Schlüssel auf den selben Wert abgebildet werden. Dies nennt man Kollision:

$$k \neq k' \text{ , } h(k) = h(k')$$

Für die Abbildung verwenden wir eine mathematische Formel, welche Hashfunktion genannt wird. Diese kann beliebig gewählt werden. Dabei kann man z.B. die Quersumme einer Zahl als Hashfunktion verwenden.
Um eine geeignete Funktion zu implementieren, muss diese einige Eigenschaften erfüllen. Selbstverständliche sollte eine Hashfunktion wenige Kollisionen verursachen, vor allem wenn man diese in den Bereichen Kryptographie und Datenbankverwaltung einsetzt. Eine perfekte Hashfunktion wäre hierbei injektiv, d.h. ein Schlüssel wird immer nur auf genau einen Hashwert abgebildet. Dies ist aber in der Praxis üblicherweise mit den gegebenen Ressourcen so gut wie unmöglich. Nur mit großem Aufwand gelingt dieses.[1]
Ein weiterer Punkt ist die Surjektivität der Hashfunktion, d.h. dass jeder Schlüssel immer auf mindestens einen Hashwert abgebildet werden muss, bzw. jeder Schlüssel hat

[1] Momentan sind die SHA-2- und SHA-3-Familien Kollisionsfrei

einen Hashwert. Zudem ist es wichtig, dass relativ gleich aussehende Schlüssel komplett verschiedene Hashwerte haben, so dass man nicht die Ähnlichkeit erkennen kann. Dies nennt man in der Mathematik Chaos. Als Letztes sollte darauf geachtet werden, dass der Speicherbedarf der Hashwerte kleiner als der Speicherbedarf der Eingabewerte ist, sonst wäre die Effektivität mancher Einsätze nicht gegeben.

2.2 Einsatz von Hashing

In der Praxis kann man den Einsatz von Hashing in drei Bereiche aufteilen: Datenbankverwaltung/Datenbankindex, Prüfsummen und Kryptographie. Der allgemeinste Fall ist die Datenbankverwaltung. Hierbei wird der Hashwert als Datenbankindex eingesetzt.
In der heutigen Zeit werden gigantische Mengen von Daten gespeichert und verwaltet. Hierbei werden verschiedenste Sortieralgorithmen und Datensätze verwendet, die alle ihre Vor- und Nachteile haben. Ein größer Nachteil der meisten dieser Möglichkeiten ist die Laufzeit. Wenn wir uns das simple Beispiel eines Arrays vor Augen halten, müssten wir im worst case scenario bei einer Suche eines Elements alle n[1] Indexwerte des Arrays betrachten. Dies gilt auch beim Einfügen eines Elements, wenn man dieses einfach nur nacheinander in das Array einfügen würde. Wird aber eine Hashtabelle und eine Hashfunkion benutzt, so kann man mit nur einem Zugriff das gewünschte Element einfügen, finden und löschen.

Ein anderer Bereich ist die Verwendung von Hashfunktionen als Prüfsumme. Als kurzes und verständliches Beispiel wird die ISBN10-Nummer betrachtet. Diese ist eine zehn stellige Buch-ID, wobei die zehnte Ziffer als Puffer genommen wird. Diese Puffer werden nun so gewählt, dass bei der Division mit Rest der Quersumme der ISBN immer Null ergibt. Wenn eine andere Zahl als Ergebnis herauskommt, kann davon ausgegangen werden, dass es einen Lese- oder Schreibfehler gab bzw. dass das Buch nicht existiert. Hierbei gibt es aber auch wieder zwei Unterbereiche. Normale Hashfunktionen werden zur Erkennung von Schreib- und Lesefehler benutzt. Diese Funktionen sind meistens simpel und vom Rechenaufwand gering. Will man nun aber Schutz vor gezielten Manipulationen haben, verwendet man komplexere Hashfunktionen als Prüfsumme. Diese sind aber in der Regel vom Rechenaufwand sehr hoch, sind dafür aber nur mit sehr hohem Aufwand und Rechenleistung zu hacken.

Der letzte Punkt beinhaltet die Kryptographie, d.h. die Verschlüsselung von Daten durch eine Hashfunktionen. Hierbei verwendet man komplexe, möglichst kollisionsfreie Funktionen und nutzt dabei eine weitere Eigenschaft der Hashfunktionen. Diese Hashfunktionen können nicht rückwärts angewendet werden. Man nennt sie deshalb auch Einwegfunktionen, d.h. sie sind nicht zurückverfolgbar. Möchte man nun herausfinden, welcher Schlüssel auf den Hashwert abgebildet wurde, müsste man dies durch das Ausprobieren aller Schlüssel auf die Hashfunktion testen[2].

[1] $n \in \mathbb{N}$, Array[1,...,n]
[2] Brute-Force-Angriff

2.3 Hashfunktionen

Wie im obigen Absatz beschrieben, gibt es sehr viele Hashfunktionen und sogar die Quersumme eines Wertes kann verwendet werden. Die Effektivität der Hashfunktion hängt dann von den jeweiligen Eigenschaften ab und ihrer Komplexität. In der Theorie, d.h. in den meisten Büchern und Internetseiten, werden hierbei drei Methoden beschrieben, welche im Folgenden kurz erläutert werden.
Zunächst gibt es die Divisionsmethode, die bekannteste Hashfunktion, welche auch in vielen Algorithmen verwendet wird. Hierbei wird im einfachsten Fall der Schlüssel Modulo mit einen festen Wert berechnet. D.h. der Schlüssel wird durch den Wert dividiert und man erhält als Ergebnis den Rest der Berechnung:

$$h(k) = k \bmod m$$
$$h(19876) = 19876 \bmod 11 = 10^{1}$$

m ist hierbei auch ein Element der natürlichen Zahlen und sollte stets eine Primzahl sein bzw. wird in der Praxis, z.B. bei der Datenbankverwaltung, m mit der Größe der Hashtabelle gleichgesetzt, damit auch ein gültiger Index errechnet wird.
Eine weitere Methode ist die Mitquadratmethode. Hierbei werden führende und endende Ziffern abgeschnitten, bis diese sich in einem gültigen Wertebereich befinden:

$$h(10278436) = 1027\ [8]\ 436 = 8^{1}$$

Die Zerlegungsmethode ist die letzte Methode, welche hier erläutert wird. In dieser Berechnung werden die Schlüssel solange zerteilt, bis ein gültiger Wertebereich erreicht wird. Hierbei gibt es zwei verschiedene Möglichkeiten. Eine ist, den gegeben Wert in einzelne Werte zu zerlegen und das Ergebnis, je nach Bedarf, weiter zu zerlegen:

$$h(135612) = [13]+[56]+[12] = 81 = [8]+[1] = 9^{1}$$

[1] Beispiele: http://openbook.galileocomputing.de/c_von_a_bis_z/022_c_algorithmen_005.htm 10.01.2014

2.4 Einleitung Hashtabelle

Hashtabellen wurden schon kurz erläutert, nun wird die Funktionsweise genauer durchleuchtet und die Implementation in C erklärt.

2.4.1 Offene Adressierung (geschlossenes Hashing)

Es gibt zwei unterschiedliche Arten von Hashtabellen. Eine davon ist die Hashtabelle mit offener Adressierung. Hierbei hat die Tabelle eine begrenzte Zahl an Adressen, was bedeutet, dass jede Adresse nur einen Wert speichern kann. Wenn man nun einen Schlüssel in die Tabelle speichern will, muss man zunächst die Hashadresse errechnen, d.h. den Index der Tabelle in der der Wert gespeichert werden soll. Dazu hasht man den Schlüssel und verwendet diesen Hashwert dann als Hashadresse. Nun kann es aber zu einer Kollision kommen. Wenn dieser Fall eintritt und die Tabelle noch nicht komplett befüllt ist, werden die sogenannten Kollisionsauflösungsstrategien verwendet. Hierbei gibt es wieder mehrere Möglichkeiten. Im Folgenden werden drei Strategien näher erläutert: lineare Sondierung, quadratische Sondierung und doppeltes Hashen.
Bei der linearen Sondierung wird der Hashwert und somit die Hashtabellenadresse mit eins addiert. Dies wird solang durchgeführt bis eine freie Adresse gefunden wurde, in welche man den Wert schreiben kann.
Wird die quadratische Sondierung genutzt, wird der Hashwert quadriert und wenn nötig Modulo der Größe der Hashtabelle genommen. Dies wird wiederum solange durchgeführt, bis eine leere Adresse gefunden wurde.
Die letzte Strategie, die hier erläutert wird, ist das doppelte Hashen. Wenn eine Kollision entdeckt wurde, wird der Hashwert noch einmal gehasht, so dass man einen neuen Hashwert erhält.
Diese Strategien werden beim Einfügen, Suchen und sämtlichen anderen Operationen genutzt, falls Kollisionen entdeckt werden.

2.4.2 Adressierung mit verketteten Listen

Eine andere Art eine Hashtabelle zu erstellen ist, dass ihre Indexwerte eine verkettete Liste beinhalten. Soll nun ein Wert eingefügt werden, wird dieser ans Ende der verketten Liste angeheftet. Je nach Implementation kann dieser Wert auch an den Anfang der Liste geheftet werden. Dies hat den Vorteil, dass bei einer Kollision nicht eine neue Adresse ausgerechnet werden muss, sondern der Wert einfach an die Liste angeheftet wird. Außerdem hat die Tabelle theoretisch unendlich viel Platz, was bei dynamischen Größen von Daten eine großen Vorteil mitbringt. In der Praxis wird dies meistens so gelöst, dass man ein Array erstellt und jedem Indexwert des Arrays einen Linked List übergibt. Dieses und andere Implementationen werden im folgenden Kapitel genauer erklärt.

3 Implementation

Im nun folgenden Kapitel wird die Implementierung von Hashfunktionen und Hashtabellen genauer erläutert. Die folgenden Funktionen wurden in C geschrieben, hierbei ist zu beachten, dass in der Standardbibliothek von C keine Datenstrukturen oder vordefinierte Funktionen im Bereich Hashing vorhanden sind. Daher wird erst erklärt, wie man diese mit der Standardbibliothek entwickelt und danach wird eine vordefinierte Bibliothek gezeigt, welche als Open Source Code frei im Internet zur Verfügung steht.

3.1 Standardbibliothek

3.1.1 Hashfunktion

Es gibt viele verschieden Hashfunktion, diese werden zusätzlich noch unterschieden in normale und kryptografische Hashfunktionen. Zudem unterscheiden sich die jeweiligen Funktionen in Form, Komplexität, Zeitaufwand und die Arte der mathematischen Methode. Im Folgenden wird eine sehr simple aber verbreitete Hashfunktionen gezeigt, die in der Datenbankverwaltung angewendet wird.

Zunächst werden die benötigten Bibliotheken eingebunden, danach wird eine globale Variable MAX_HASH definiert. Diese Variable repräsentiert die Größe der Hashtabelle. Dies ist sofern wichtig, da man das Ergebnis der Hashfunktion als Adresse der Hashtabelle interpretieren wird und man daher einen zulässigen Wert haben will.
Im Kopf der Funktionen legt man fest, dass die Funktion einen Integer-Wert zurückgeben und dass ihr ein Array aus Chars übergeben werden soll. Dieses Array dient als String, d.h. jeder Indexwert dieses Arrays hat als Inhalt einen Buchstaben bzw. einen Char des Wortes/Strings.

```
1    int hashfunction(char *string)
```

In der Funktion werden zwei lokale Variablen erstellt: Einen unsigned Interger-Wert namens hash_add, welcher die Adresse der Hashtabelle repräsentieren soll und einen unsigned Char-Pointer. Wir nehmen unsigned-Werte, da wir nur positive Werte als Ergebnis erhalten wollen.

```
2    {
3        unsigned int hash_add;
4        unsigned char *pointer;
```

Danach initialisieren wir den Wert hash_add mit Null und übergeben dem lokalen Pointer den String. Nun durchläuft der Pointer eine While-Schleife, die solange durchlaufen wird bis der Pointer den Wert '\0' findet, was bedeutet, dass der String zu Ende ist. In der While-Schleife wird nun die Hashadresse errechnet, indem die aktuelle Hashadresse mit 33 multipliziert wird und dann der Wert des aktuellen Pointers. D.h. der Wert des aktuellen Buchstabens des String wird addiert. Danach wird der Pointer inkrementiert, so dass dieser auf den nächsten Buchstaben/Char des Strings zeigt.
Wenn nun die While-Schleife ihre Bedingung erfüllt und das Ende des Strings erreicht hat, wird der Wert Modulo MAX_HASH genommen und wiedergegeben.

```
5         hash_add = 0;
6         pointer = (unsigned char *) string;
7         while(*pointer != '\0')
8         {
9             hash_add = 33 * hash_add + *pointer;
10            pointer++;
11        }
12        return hash_add % MAX_HASH;
13    }
```

Im späteren Kapitel Test wird die selbst implementierte Hashfunktion mit der Hashfunktion der glib-Bibliothek verglichen. Daher wird diese Hashfunktion genommen:

$$h(k) = k * 33 \bmod MAX\_HASH$$

Diese Hashfunktion wird auch in der glib-Bibliothek verwendet. Zusätzlich muss beachtet werden, dass hash_add später nicht mit Null, sondern mit 13845163 initialisiert wird.

3.1.2 Hashtabelle mit offener Adressierung

Auch in diesem Punkt gibt es verschiedene Möglichkeiten, eine Hashtabelle zu implementieren. Hierbei wurde das folgende Beispiel so gewählt, dass es einfach zu verstehen und zu implementieren ist und die Grundeigenschaften einer Hashtabelle besitzt. Zudem werden in diesem Beispiel nur Strings in der Tabelle gespeichert. Selbstverständlich können alle Arten von Daten gehasht werden und Hashtabellen gespeichert werden, jedoch wurde dies aus Verständnisgründen außer Acht gelassen.

Zunächst wird zusätzlich zu MAX_HASH eine weitere globale Variable definiert: MAX_STRING. Dieser Wert legt die erlaubte Größe für die Strings fest, welche in die Tabelle eingefügt werden soll.

```
1    #define MAX_HASH 100
```

Nun wird ein zweidimensionales Array aus Chars erstellt. Die Größe der Tabelle legt hierbei MAX_HASH fest und die Größe der jeweiligen Indexwerte der ersten Dimension des Arrays MAX_STRING. D.h. jeder Behälter des Array besitzt wiederum ein Array aus Chars, welche die Strings repräsentieren sollen.

```
2    char hash_table[MAX_HASH][MAX_STRING];
```

Des Weiteren wird nun die Operation Insert, d.h. das Einfügen eines Elements in die Tabelle, erklärt. Die Operationen Löschen und Suchen werden kurz erläutert, aber nicht genau erklärt, da die Implementierung dieser Operationen sehr ähnlich sind und daher leicht ableitbar. Im Kopf der Funktion wird festgelegt, dass die Funktion nichts zurückgeben soll und daher mit void initialisiert wird. Der Funktion wird ein String übergeben.
Danach wird im Rumpf der Funktion die lokale Variabel hashad initialisiert, welche die Hashadresse repräsentieren soll. Hierbei rufen wir die Funktion zu Hashwerterrechnung auf und übergeben dieser den String. Man erhält dadurch einen Integer-Wert, also die Hashadresse.

```
3    void insertOpAd(char *string)
4    {
5        int hashad = hashFunction(string);
```

Nun wird eine While-Schleife solange durchlaufen, bis ein freier Platz in der Tabelle gefunden wird. Wenn dies nicht der Fall ist, wird die Hashadresse um eins inkrementiert und Modulo MAX_HASH genommen. Dies dient als Kollisionsauflösungsstrategie. D.h. wenn eine Kollision vorliegt und in dem Index der Tabelle schon ein Element gespeichert ist, geht man die Tabelle solange durch, bis ein freier Platz gefunden wird.

```
6           while(hash_table_open[hashad][0] != 0)
7           {
8               hashad = (hashad + 1) % MAX_HASH;
9           }
```

Wenn nun ein freier Platz gefunden wurde, wird der String in die Hashtabelle geschrieben an dem geeigneten Index. Dies passiert in diesem Beispiel durch die Funktion strncpy(char *Ziel, const char *Quelle, size_t num):

```
10      strncpy(hash_tabl[hashad],string,MAX_STRING);
11      }
```

Die Operationen Suchen und Löschen funktionieren nach dem selben Prinzip, nur wird hier der gesuchte Wert mit dem Wert in der Tabelle vergleichen, um zu erkennen, ob es sich um den richtigen Wert handelt, falls das nicht der Fall ist, wird die While-Schleife der Kollisionsauflösungsstrategie verwendet, um das richtige Objekt in der Tabelle zu finden.

3.1.3 Hashtabellen mit verketteten Listen (Linked Lists)

Nun wird die Funktionsweise von Hashtabellen erläutert, die als Inhalt ihres Indexwertes verkette Listen besitzen. D.h. die Elemente, die in die Tabelle verwaltet werden sollen, werden am jeweiligen Index/Hashadresse in die verkettete Liste eingefügt.

Als einfaches und überschaubares Beispiel wird das Szenario verwendet, in dem eine Person ihren Namen, Vornamen und Alter in einen Tabelle verwalten will.
Hierzu wird als Erstes eine neue Datenstruktur angelegt, welche als perso bezeichnet wird. Dieses Struktur bekommt als Komponenten ein char-Array vname, welches die Vornamen repräsentieren soll, ein char-Array nname, welches die Nachnamen repräsentieren soll, ein Integer age, welche das Alter repräsentieren soll und einen Pointer next, der auf seinen Nachfolger weisen soll.

```
1       struct perso
2       {
3           char vname[255];
4           char nname[255];
5           int age;
6           struct perso *next;
7       };
```

Nun werden drei Struktur-Variablen deklariert: start, current und last, welche als nützliche Zeiger dienen. Sie repräsentieren das Anfangs- und Schlusselement sowie das aktuelle Element der verketten Liste in dem jeweiligen Index der Hashtabelle. Zusammen nennen wir diese Struktur-Variable perso_list.

```
8     typedef struct{
9         struct perso *start;
10        struct perso *current;
11        struct perso *last;
12    }perso_list;
```

Danach wird ein Array erstellt, welches mit perso_list initialisiert wird und die Größe MAX_HASH hat. Dies ist dann die Hashtabelle, deren Inhalt aus verketteten Listen besteht.

```
13    perso_list *hash_table[MAX_HASH];
```

Kurz zusammengefasst wurde ein Array, bestehend aus der Struktur-Variable perso_list, bestehend aus den Struktur-Variable perso, welche aus den Komponenten vname(char), nname(char), age(int) und next(perso) bestehen, erstellt, welche nun eine Hashtabelle repräsentiert, die als Indexwert eine verkettet Liste besitzt.

Da nun eine Hashtabelle erstellt wurde, werden die Operationen näher erläutert. In diesem Fall werden wieder die Funktion Einfügen genauer betrachtet und Suchen und Löschen kurz erklärt, da auch hier wieder die Funktionsweise ableitbar ist.

Weiterhin wird das Szenario verwendet, dass eine Person in einer Hashtabelle Name, Vorname und Alter verwalten will. D.h. wir übergeben der Funktion drei Variablen und entscheiden zunächst, dass die Funktion nichts zurückgeben soll.

```
1     void insert(char *nname, char *vname, int age)
2     {
```

Die lokale Variable hash_ad erhält über die Funktion HashFunction einen Hashwert bzw. die Hashadresse. Hierbei wird nur der Nachname, also nname, gehasht. Welche Komponente gehasht werden soll, hängt einerseites vom Entwickler ab, anderseits von der Aufgabenstellung. Danach wird ein lokaler Pointer list erstellt, welcher die Pointeradresse der Hashtabelle übergeben wird. D.h. list zeigt nun auf die verkettete Liste des Indexwertes des Arrays.

```
3         int hash_ad = hashFunction(nname);
4         perso_list *list = &hash_table[hash_ad];
```

Um das Element nun in die Tabelle schreiben zu können, muss vorher noch überprüft werden, ob eine Kollision vorliegt. Bei einer Kollision bleibt die Hashadresse zwar gleich, doch muss geprüft werden, ob sich schon Elemente in der verketteten Liste befinden. Dies wird in diesem Beispiel durch eine If-Abfrage getestet. Als Erstes wird geprüft, ob die Struktur-Variable start der perso_list list NULL ist. Falls dies der Fall ist, wird erst Speicher reserviert und dann wird angegeben,dass der Listenanfang gleichzeitig das Listenende ist.

```
5           if(list->start == NULL){
6               list->start = malloc(sizeof(struct perso));
7               list->last = list->start;
```

Falls aber beim Fallunterschied false herauskommen sollte, wird der Else-Fall aufgegriffen. In diesem wird auf die Struktur-Variable next des letzten Elementes der Liste zugegriffen. Diesem wird Speicher alloziert und es wird festgelegt, dass dieser nun auch das letzte Element der Liste ist.

```
8           } else {
9               list->last->next = malloc(sizeof(struct perso));
10              list->last = list->last->next;
11          }
```

Da nun alle Fälle überprüft worden sind, werden die passenden Inhalt in die Tabelle bzw. in die verkettete Liste eingetragen.

```
12          if(!list->last) assert("Memmory Error");
13
14          strcpy(list->last->nname, nname);
15          strcpy(list->last->vname, vname);
16          list->last->age = age;
17      }
```

Die Operationen Suchen und Löschen funktionieren ähnlich. Allerdings wird in diesen Szenarien eine While-Schleife verwendet, die an der ausgerechneten Hashadresse die verkettete Liste durchgeht, um das gesuchte Element zu finden.

3.1.4 Adler32

Bei Adler32 handelt es sich um einen Prüfsummen-Algorithmus, welcher von Mark Adler entwickelt wurde und unter anderem in der zlib-Bibliothek verwendet wird. Dieser Algorithmus wird im Folgenden kurz erläutert. Er dient als gutes Beispiel, um die Funktionsweise von Hashing im Bereich Prüfsummen zu demonstrieren, da dieser relativ simpel ist. Zur Erinnerung: Diese Art von Prüfsummen werden zur Erkennung von Schreib- und Lesefehlern verwendet.

```
1   uint32_t adler(unsigned char *data, size_t datalength)
2   {
3       uint32_t s1 = 1;
4       uint32_t s2 = 0;
5       size_t n;
6
7       for(n = 0; n < datalength; n++)
8       {
9           s1 = (s1 + data[n]) % 65521;
10          s2 = (s2 + s1) % 65521;
11      }
12
13      return(s2 << 16) | s1;
14  }
```

Zunächst werden der Funktion die Daten in Form von unsigned chars und deren Größe als size_t-Variable übergeben. Danach werden im Rumpf zwei lokale uint32_t-Variablen erstellt mit den Namen s1 und s2. s1 wird auf 1 initialisiert und s2 auf 0. Diese beiden Variablen haben folgende Funktion:
s1 zählt alle Datenbytes zusammen, dies ist in Line 9 zu sehen. In der for-Schleife werden alle Datenbytes durchlaufen, daher wird der Funktion auch die Größe bzw. Länge der Daten übergeben. Der aktuelle Wert von s1 wird dann mit dem aktuelle Datenbyte addiert und anschließend mit Modulo 65521 berechnet.
s2 addiert danach die Summe aller s1-Werte und berechnet auch hier Modulo 65521.
Am Ende der Funktion, wenn alle Datenbytes betrachten wurden, wird s2 mit 16 geshiftet ($x << 16 \equiv x * 2^{16}$) und als uint32_t-Wert ausgegeben.

Kryptografische Prüfsummen und Verschlüsselungstechniken wurden auf Grund ihrer Komplexität und Zeitaufwandes in dieser Ausarbeitung nicht näher betrachtet. Dieser Bereich ist so umfangreich, dass man ihm eine eigene Ausarbeitung zuschreiben kann.

3.2 glib

glib ist eine vom GTK+-Team implementierte Ergänzung zu den Standardbibliotheken von C. Es werden hierbei zusätzliche Datenstrukturen und Funktionen ergänzt, um es dem Anwender/Entwickler leichter zu machen. Zusätzlich handelt es sich um Open Source Code, d.h. es ist frei im Internet verfügbar und darf unter gewissen Regeln genutzt werden.

3.2.1 Einleitung glib

Um die glib zu benutzen, müssen wir die Bibliothek glib.h per include-Befehl in das Programm einbinden. Vorher müssen auch Voreinstellungen getroffen werden, z.B. im Compiler und im Linker.
In diesem Projekt wurde unter einem Windows-Betriebssystem gearbeitet mithilfe der IDE wx-Dev Cpp[1].

Bei der Verwendung von glib werden nun keine Arrays benutzt, sondern die von glib vordefinierten Datenstrukturen GHashtable. Um nun eine Hashtabelle zu erstellen, instantiieren wie die Tabelle:

```
1    GHashtable hash_table = *g_hash_table_new(g_str_hash,
       ↪ g_str_equal);
```

Mit der Funktion g_hash_table_new() wird eine neue Hashtabelle erstellt. Übergeben wird ihr hierbei eine Hashfunktion, die die Art der Hashfunktion festlegt und eine Vergleichsfunktion, die für das Vergleichen der Objekte zuständig ist, falls bei einem Suchbefehl unter der Adresse, mehrere Elemente vorliegen.
Es gibt mehrere Hashfunktionen. In diesen Beispiel wurde g_str_hash verwendet. Diese Funktion hasht Strings, hierbei nutzt diese die selbe mathematische Formel, welche im Punkt 3.1.1 genutzt wurde

$$h(k) = 33 * k \bmod m$$
oder
hash_add = 33 * hash_add + *pointer;

Einen Unterschied gibt es jedoch, bei der Funktion in 3.1.1 wurde hash_add mit 0 initialisiert. Bei glib wird dies mit 5381 gemacht. Außerdem verwendet glib für ihre Größe der Tabelle eine minimale Größe von 11 und eine maximale Größe von 13845163.

[1] http://wxdsgn.sourceforge.net/, 06.03.2014

Folgende Hash- und Vergleichsfunktionen stehen bei glib zur Verfügung:

Hashfunktionen	Vergleichfunktionen
g_direct_hash()	g_direct_equal()
g_int_hash()	g_int_equal()
g_int64_hash()	g_int64_equal()
g_double_hash()	g_double_equal()
g_str_hash()	g_str_equal()

Mit einem kleinen Beispielcode werden nun einige Funktionen und Operationen kurz und bündig erläutert. Es werden in die Hashtabelle mehrere Namen eingefügt. Dies wird mit der Funktion g_hash_table_insert(*hash_table, key, value) verarbeitet. Dieser Funktion werden drei Komponenten übergeben: g_Hashtablle ist die Tabelle, in der das Element eingefügt werden soll, key ist das Element, was gehasht werden soll, und value ist das Element, was in die Adresse der Tabelle geschrieben werden soll.
Danach ersetzen wir ein Element in der Tabelle mit der Funktion g_hash_table_replace(*hash_table, key, value). Hierbei wird in der Hashadresse das sich dort befindende Element durch das Element value ersetzt.
Zusätzlich wird ein Element aus der Tabelle gelöscht via g_hash_table_delete(*hash_table, key, value).
Mit der Funktion g_hash_table_lookup(*hash_table, key) werden danach die Elemente der Hashtabelle ausgegeben.

```
1  int main()
2  {
3      GHashTable *hash_table =
          ↪ g_hash_table_new(g_str_hash,g_str_equal);
4
5      g_hash_table_insert(hash_table,"Boehme","Paulus");
6      g_hash_table_insert(hash_table,"Simpson","Homer");
7      g_hash_table_insert(hash_table,"Muster","Max");
8      g_hash_tbale_replace(hash_table,"Simpson","Bart");
9
10     g_hash_table_remove(hash_table,"Muster");
11
12     int size = g_hash_table_size(hash_table);
13
14     printf("%s\n",g_hash_table_lookup(hash_table,"Boehme"));
15     printf("%s\n",g_hash_table_lookup(hash_table,"Simpson"));
16     printf("%s\n",g_hash_table_lookup(hash_table,"Muster"));
17     printf("Size: %i \n\nDone...",size);
18
19     getchar();
20
```

```
21      return 0; }
```

Dieser Beispielcode hat nun die folgende Ausgabe:

3.2.2 kryptografische Prüfsummen

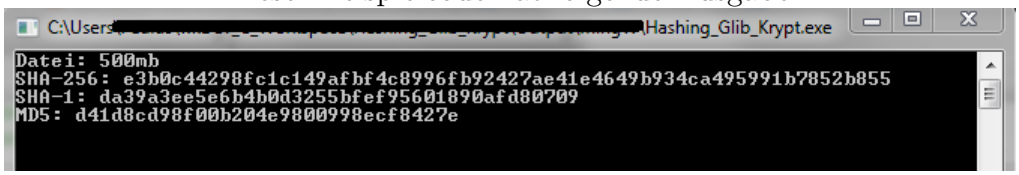
Zusätzlich zu den Standard Hashfunktionen und Datenstrukturen, die glib mitbringt, bietet die Bibliothek auch eine kryptografische Hashfunktion zur Erstellung von Prüfsummen an. Hierbei ist zu unterscheiden, dass Prüfsumme und Verschlüsselung zwei verschiedene Themen sind. Auch wenn beide zum Thema Kryptographie gehören, werden kryptografische Prüfsummen zur Erkennung von Manipulationen an Daten verwendet. Anders als der Einsatz von normalen Prüfsummenfunktionen, wie z.B. Adler32, welche sich um die Erkennung von Lese- und Schreibfehlern kümmern, wird hier auf den gezielten Angriff auf eine Datei geachtet.
Das folgende Beispiel zeigt die vorhanden Funktionen, die glib anbietet. Hierbei werden folgende Algorithmen angewendet: SHA-1, SHA-256 und MD5. Diese unterscheiden sich in der Länge der Ausgabewerte und ihrer Komplexität. MD5 gibt Wörter der Länge 16 Bit zurück, SHA1 Wörter der Länge 20 Bit und SHA256 Wörter der Länge 32 Bit. Es wurde für den Test einen 500mb große Testdatei erstellt, welche mit Nullen befüllt wurde[1].

```
1  int main()
2  {
3  gchar *sha1, *sha256, *md5;
4  FILE data;
5  data = fopen("Testfile","r");
6
7  sha256 =
     ↪ g_compute_checksum_for_data(G_CHECKSUM_SHA256,data,-1);
8  sha1 = g_compute_checksum_for_data(G_CHECKSUM_SHA1,data,-1);
9  md5 = g_compute_checksum_for_data(G_CHECKSUM_MD5,data,-1);
10
11 printf("Datei: 500mb \n");
12 printf("SHA-256-Hashwert: %s \n",sha256);
13 printf("SHA-1-Hashwert: %s \n",sha1);
14 printf("MD5-Hashwert: %s \n",md5);
```

[1]cmd-Befehl: fsutil file createnew [Pfad+Dateiname] [Dateigröße in Bytes]

```
15
16 fclose(data);
17 return 0;
18 }
```

Dieser Beispielcode hat folgende Ausgabe:



3.3 libgcrypt

Zum Thema Verschlüsselung gibt es unter glib keine Funktionen. Während der Recherchen wurde allerdings eine weitere Open-Source-Bibliothek, welche verschieden Verschlüsselungsfunktionen anbietet, gefunden. Darunter befinden sich nicht nur kryptografische Hashfunktionen, sondern auch weitere symmetrische und asymmetrische Kryptofunktionen, wie z.B. RSA, AES und co.
Aus Zeitgründen wurde dieser Teil des Themas Hashing allerdings weggelassen, da dieser Bereich schon eine eigene Bearbeitung, d.h. eine eigene Präsentation und Ausarbeitung einnehmen würde und sich diese Ausarbeitung auf das Allgemeine und die Anfänge des Hashing konzentriert. Es sollte trotzdem bekannt sein, dass der Bereich der kryptografischen Hashfunktionen einerseits sehr komplex ist, aber in unserer heutigen Zeit nahezu zum Standard gehört.
libgcrypt ist nun eine vordefinierte Bibliothek, welche verschiedene Kryptosysteme zur Verfügung stellt. Hierbei muss aber auch die Bibliothek error.h mit eingebunden werden, welche Error-Handler bereitstellt, die verschiedenen Fehlerszenarien vorbeugt. Links zu der Downloadseite der Bibliotheken und auch zur Dokumentation liegen im Anhang.

4 Effizienz

4.1 Test

Im Folgenden wird das Hauptaugenmerk dieser Ausarbeitung betrachtet: die Effizienz. Hierbei wurden einerseits theoretisch die Unterschiede zwischen Hashtabelle und den Standard-Datenstrukturen betrachtet, anderseits wurde auch praktische Unterschiede durch Testdurchführungen betrachtet.
Im Praxisteil wurden drei Szenarien durchleuchtet. Es wurden Test konstruiert zum Vergleich der beiden Datenstrukturen Hashtabelle und Array, zum Vergleich zwischen einer Hashtabelle mit offenere Adressierung und einer mit verketteten Listen sowie zum Vergleich zwischen selbst implementierten Funktionen und den Funktionen der glib-Bibliothek.

4.1.1 Theorie

Im folgenden Theorieteil wurde die Hashtabelle mit anderen Datenstrukturen[1], die zur Verwaltung von Daten benutzt werden, mit Hilfe der Big-O-Notation(auch Landau-Notation) verglichen. Hierbei wurden die Operationen Einfügen, Suchen und Löschen als Vergleichsparameter verwendet. Daraus ergab sich folgende Tabelle, die das Average-Case-Szenario[2] beschreibt:

	search()	insert()	delete()
Hashtabelle	O(1)	O(1)	O(1)
Array	O(1)/O(n)	O(n)/O(log n)	O(n)
BST[1]	O(log n)	O(log n)	O(log n)
Linked List	O(n)	O(1)	O(1)

Wie sehr gut zu erkennen ist, hat die Hashtabelle gegenüber allen anderen Datenstruktur jeweils einen großen Vorteil. Die einzige Datenstruktur, die der Hashtabelle „das Wasser reichen kann“ist die Linked List. Anhand dieser Tabelle sind die Ergebnisse leicht zu verstehen. Wird die Spalte search() betrachtet, befindet sich im Feld der Reihe Hashtabelle O(1). Dies ist leicht zu erklären, da die Hashtabelle nur mit einen Zugriff das gesuchte Element finden kann. Bei einem Array kann es im Worst-Case vorkommen, dass das ganze Array von oben nach unten durchlaufen werden muss, da sich das gesuchte Element an letzter Stelle befindet. Bei der Hashtabelle kennt man durch die Bildung des Hashwertes die Adresse des gesuchten Elements. Dies trifft auch bei den Operationen insert() und

[1] BST = **B**inary **S**earch **T**ree
[2] Durchschnittsdauer

delete() zu, hier wird jeweils immer vorab die Hashadresse errechnet, sodass das Element immer mit einem Zugriff gefunden und verwaltet wird. Im Worst-Case-Szenario hat die Hashtabelle aber auch die jeweilige Laufzeit von O(n), da bei Kollisionen die gesamte Tabelle durchsucht werden muss (offener Adressierung) bzw. die verkettete Liste.

4.1.2 Test: Hashtabelle vs. Array

Beim Vergleich zwischen der Hashtabelle und einem Array wurde eine Hashtabelle mit offener Adressierung ausgewählt, da sie beinah auf dem gleichen Prinzip eines Arrays aufgebaut ist. Als Hashfunktion wurde wieder folgende Formel verwendet:

$$h(k) = 33 * k \bmod m;$$

Als Kollisionsauflösungsstrategie wurde die lineare Sondierung gewählt, d.h. bei einer Kollision wird die Hashadresse immer um eins addiert bis ein leerer Indexwert gefunden wird.
Beim Array werden die Elemente nacheinander in das Array eingefügt, bei der Operation Suchen wird eine While-Schleife verwendet, die das Array von Anfang bis Ende durchgeht, um das Element zu finden. Die Operation Löschen sieht ähnlich aus. Der komplette Quellcode der Tests befindet sich auf der Seite: http://wr.informatik.uni-hamburg.de/teaching/wintersemester_2013_2014/effiziente_programmierung_in_c bereit zum Herunterladen.
Um Ergebnisse zu erhalten, mit denen man arbeiten kann, werden die Operationen mit einer for-Schleife wiederholt. Bei diesen Test kamen folgende Parameter zum Einsatz:
Die Größe des Hashtabelle und des Arrays wurde auf 3902 gesetzt, da für die Test eine Liste aus 3902 Männervornamen gewählt wurde. Da hierbei mit einer offene Adressierung gearbeitet wird, muss die Hashtabelle bzw. das Array so viele freie Plätze haben, wie es Vornamen gibt. Zudem werden 100 Iterationen durchgeführt.

Folgende Ergebnisse kamen bei diesem Test heraus:

	Hashtabelle	**Array**
insert():	0,45s	0,06s
search():	0,47s	6,15s

Hierbei fällt zuerst auf, dass bei der Hashtabelle die Einfüge- und Suchoperation beinahe die gleiche Zeit benötigen, während bei dem Array eine große Lücke zwischen den beiden Operationen steht. Allerdings fällt auch auf, dass das Array beim Einfügen deutlich schneller ist. Dies kann man insofern deuten, dass bei der Hashtabelle vorher immer erst die Hashadresse ausgerechnet werden muss, die auch eine gewisse Zeit in Anspruch nimmt, während das Array jedes Element stumpf Schritt für Schritt einfügt.
Die Operation Löschen wurde hierbei außer Acht gelassen, da das Ergebnis der Suchoperation schon deutlich zeigt, dass die Hashtabelle schneller ist. Bei der Löschoperation muss man vorab eine Suchoperation durchführen, um das Element in der Hashtabelle/Array zu finden.

Die Operation Suchen verdeutlicht den großen Vorteil einer Hashtabelle. Die Hashtabelle ist um ein Sechsfaches schneller als das Array. Dies ist dadurch zu erklären, dass die Hashtabelle vorab zwar die Hashadresse ausrechnen muss, danach mit nur einem Zugriff sofort das Element findet, währenddessen das Array im Worst-Case-Szenario sich selbst komplett von Anfang bis Ende durchlaufen muss.

4.1.3 Test: Offene Adressierung vs. verkettete Listen

Bei dem Vergleich zwischen diesen zwei Arten der Adressierung wird die gleiche Hashfunktion verwendet, allerdings sind hierbei die Größen der beiden Tabellen unterschiedlich. Da in diesem Beispiel auch wieder die Liste mit den männlichen Vornamen verwendet wird, muss die Hashtabelle mit der offenen Adressierung auch dementsprechend 3902 freie Plätze besitzen. Die Hashtabelle mit verketten Listen hat dagegen die Größe 100, d.h. grob gesagt: eine Tabelle mit 100 verketten Listen. Zudem werden hierbei 1000 Iterationen genommen, um ein geeignetes Ergebnis zu bekommen:

	offene Ad.	**Verkettete Listen**
insert():	4,46s	1,36s
search():	5,44s	1,56s

Bei diesem Test wird offensichtlich, wie effizient eine Hashtabelle mit verketten Listen ist. Eine Hashtabelle brauch beinahe das Fünffache an Zeit gegenüber einer Hashtabelle mit verketten Listen. Dies ist auch leicht zu nachvollziehen. Bei der Tabelle mit offener Adressierung muss bei einer Kollision eine extra While-Schleife durchgegangen werden, solange bis ein freier Platz gefunden wird. Dies kann viel Zeit in Anspruch nehmen. Bei einer Hashtabelle mit verketteten Listen wird das Element an den Anfang oder an das Ende der Liste gehangen. Dies ist beim Einfügen und Suchen deutlich effizienter.

4.1.4 Test: Selbst implementiert vs. glib

Der letzte Test befasst sich mit dem Vergleich zwischen der glib-Bibliothek und dem selbst entwickelten Code, welcher während des Semesters gefertigt wurde. Es wurde auch hier wieder die Liste mit den Männer-Vornamen verwendet. Die Hashtabelle arbeitet mit verketteten Listen und hat die Größe 100. Die glib-Hashtabelle hat eine maximal-Größe von 13845163. Die Hashfunktion von glib lautet:

$$h(k) = 33 * k \bmod m$$

und beginnt mit dem Startwert bei 5381. Die selbst implementierte Hashfunktion benutzt die gleiche mathematische Funktion und startet auch bei 5381. Der einzige Unterschied ist das m. Bei glib ist m entweder 11 (minimale Größe der Hashtabelle) oder, wie in diesem Fall, 13845163 (maximale Größe). Bei der selbst implementierten Funktion wird die Hashtabellengröße 100 gewählt, somit wäre m 100. Bei dem direkten Vergleich zwischen den Hashfunktionen wurde das berücksichtigt, d.h. m wurde auch bei der selbst

implementierten Funktion auf 138451163 gesetzt. Bei einem kleinen Zwischentest kamen dadurch immer die selben Hashwerte heraus. Danach wurden sie auch zeitlich bemessen und für 100.000.000 Iterationen ergab sich Folgendes:

	eigener Code	**glib**
Hashfunktion:	3,66s	2,63s

Es folgt daraus, dass die Hashfunktion von glib besser ist als die, die während des Seminars entwickelt wurde. Dies bedeutet aber nicht, dass eine selbst implementierte Funktion nicht besser sein kann als glib. Durch Weiterentwicklung des selbst implementierten Codes könnte man eine bessere Leistung hervorbringen, dies hängt immer von der Qualität des Codes, zeitlichen Aufwandes, Können des Entwicklers und weitere Eigenschaften ab.

Nun folgt der Vergleich der Hashtabellen. Hierbei wurden 1000 Iterationen verwendet. Zudem ist zu beachten, dass die Größe der Hashtabelle des selbst implementierten Codes aus 100 festgelegt wurde und dadurch des Hashfunktion als m auch 100 hat.:

	eigener Code	**glib**
insert():	1,33s	1,36s
search():	1,56s	0,81s

Beim Einfügen ergab sich annähernd eine Übereinstimmung und das bei mehrfachem Testen. Somit kann man sagen, dass die Operationen gleich gut funktionieren. Beim Suchen sieht das anderes aus. Hierbei ist die Funktion von glib beinahe doppelt so schnell wie der selbst entwickelte Code. Eine Vermutung liegt darin, das glib eine größere Tabelle benutzt und somit weniger Kollisionen hat. Genauer gesagt, muss sie weniger die Vergleichsfunktion aufrufen als bei der selbst entwickelten Tabelle. Dies wird durch den folgenden Test überprüft, indem diesmal die selbst entwickelte Hashtabelle auf 10000 erhöht wird:

	eigener Code	**glib**
insert()	1,46s	1,35s
search()	0,88s	0,81s

Dieser Test zeigt eindeutig, dass bei einer größeren Hashtabelle die Zeiten ungefähr gleich sind, was die Vermutung, dass bei einer kleineren Tabelle viele Kollisionen auftauchen, zum Teil bestätigt. Das zeigt weiterhin, dass die selbst entwickelte Hashtabelle und deren Funktionen gleichwertig mit der glib-Bibliothek ist.

5 Fazit

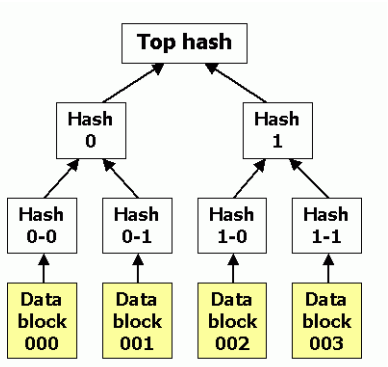
Die Tests haben sehr schöne Ergebnisse geliefert, welche meine Thesen zum größten Teil bestätigten. Schon der theoretische Teil des Punktes Test, wobei die Landau-Notation betrachtet wurde, zeigte, dass die Hashtabelle deutlich effizienter ist als andere Datenstrukturen. Beim praktischen Vergleich mit einem Array zeigte sich dieses Bild genauer. Beim Vergleich zwischen offenere Adressierung und Hashtabellen mit verketten Listen sah man auch einen deutlichen Gewinner. Die Hashtabelle mit verketten Listen war zum Teil um das vierfache schneller. Man sollte allerdings beachten, dass bei Verwendung einer besseren Hashfunktion, d.h. einer mit wenigeren Kollisionen, das Ergebnis der offenen Adressierung besser hätte ausfallen können. Allerdings trifft das auch für die Hashtabelle mit den verketten Listen zu. So kann festgestellt werden, dass bei Datenbankverwaltung mit dynamischer Anzahl an Elementen eine Hashtabelle mit verketten Listen zu wählen ist. Doch sollte die Anzahl an Elementen fest sein und zudem überschaubar, so ist der Gebrauch der einfachen Implementierung der Hashtabelle mit offener Adressierung auch eine gute Wahl.
Der Einsatz von externen Bibliotheken wie glib kann deutlich befürwortet werden. Sie erleichtern die Arbeit mit Hashing um ein Vielfaches und haben in den Tests entsprechend abgeschnitten. Man sollte aber auch beachten, dass mit mehr Arbeit und bei höhere Erfahrung in der Programmiersprache ein eigens entwickelter Code auch besser abschneiden kann und es häufiger zutreffen wird, dass die Hashtabelle oder Hashfunktionen andere Attribute ansprechen soll, sodass man auch vorsichtig mit diesen Thema umgehen sollte. Trotzdem kann man sagen, dass die Verwendung von externen Bibliotheken, sofern es erlaubt ist, auch genutzt werden soll.
Zusammenfassend kann gesagt werden, dass diese Ausarbeitung sehr interessant und aufschlussreich war. Allerdings zeigt sie nur die Anfänge des Themas und beweist, dass Hashing ein weit gefächertes Themengebiet ist. So wurden z.B. der kryptographische Teil durch seine Größe und Komplexität nur sehr kurz erklärt. Zudem wurde auch nur eine externe Bibliothek durchleuchtet und auch nur einfache Beispiele verwendet. Dennoch haben sie die Funktionalität und Einsetzbarkeit von Hashing gut verdeutlicht und beweisen bei ihrer Verwendung das effizientere Programmieren in C.

6 Weiterführendes

Im Folgenden und letzten Kapitel werden die Fragestellungen angesprochen, die das Thema Hashing weiterführen und aus Zeit- und Platzgründen nicht in dieser Ausarbeitung ausführlich besprochen wurden. Z.B. wurde in den vorherigen Punkten das Themengebiet Kryptographie kurz angeschnitten. In den folgenden Punkten Hashtrees/Hashbäume und Hardware werden die Schwerpunkte kurz und bündig besprochen, um ihre Wichtigkeit und Komplexität zu zeigen.

6.1 Hashtree

Hashtrees oder Hashbäume werden in der Praxis zur Kontrolle der Integrität von Daten eingesetzt. Es handelt sich hierbei um Binär-Bäume, deren Kinder aus Hashblöcken bestehen, außer in der letzten Ebene, in dieser befinden sich die Datenblöcke, in denen die Datei aufgeteilt ist. Die Wurzel des Baumes bildet der sogenannte Top-Hash. An diesen kann jeder Zeit die Integrität der Datei kontrolliert werden, auch wenn diese noch nicht vollständig geladen ist. Der Sinn dahinter ist die Effektivität beim Herunterladen von kleineren Datenblöcken. Sollte einer der Datenblöcke einen Schreibfehler aufweisen, so kann dieser anhand seines Hashwertes leicht ermittelt und ausgetauscht werde, ohne die gesamte Datei neu zu laden.



http://upload.wikimedia.org/wikipedia/commons/6/6d/Hash_tree.png, 06.03.2014

6.2 Hardware

In diesem Punkt wird auf das Thema Hashing in der Hardwareebene eingegangen. Hierbei werden zwei Punkte beachtet. Der erste Punkt ist das weitverbreitete Trusted-Platform-Modul und der zweite Punkt ist die voraussichtlich 2015 erscheinende SHA-Erweiterung.

6.2.1 TPM

Das Trusted Platform Modul ist ein Chipsatz, der auf dem Main-/Motherboard integriert ist und als Hardware-Schutz vor Manipulationen dient. Er gehört in die Sparte Trusted Computing. In der aktuellen Zeit werden beinahe sämtliche Motherboards mit diesem TPM ausgestattet und ausgeliefert. Dieser Chip hat verschiedene Funktionen: Speicherung von Schlüsseln, Zufallszahlengenerator, RSA-Schlüsselgenerator und weitere verschiedene Ver-/Entschlüsselungs- und Signatureinheiten. Hierbei ist auch ein Hashalgortihmus(SHA-1) integriert, welcher die Systemkonfiguration hasht, um damit die Daten des Systemes an das TPM zu binden. Dabei werden die Daten von dem Hashwert verschlüsselt. D.h. die Daten können nur noch mit den Hashwert des TPM entschlüsselt werden. Diese Funktion nennt man auch Versiegelung(engl.: sealing).

6.2.2 Intel SHA-Extension

Die SHA-Extension von Intel wird ab der neuen Chipgeneration Skylake von Intel verwendet. Dieser kommt voraussichtlich 2015 auf dem Markt. Die SHA-Erweiterung beinhaltet dann die SHA-1- und SHA-256-Familien. Sie wurden mit eingebunden, um die Rechenzeit dieser Algorithmen auf Intel-Chips deutlich zu verbessern, da in der aktuellen Zeit immer mehr Wert auf Sicherheit, Verschlüsselung und Integrität gelegt wird.
Momentan kann man sich über Intel eine SDE[1] herunterladen, welche die SHA-Extension beinhaltet.

[1]Software Development Emulator

7 Vokabeln

Chaos	Zwei ähnliche Elemente haben komplett verschiedene Abbildungen
hashen	Verb, auf einen Schlüssel wird eine Hashfunktion angewendet
Hashadresse	Adresse der Hashtabelle, in der ein Element verwaltet wird
Hashwert	Mathematisches Ergebnis, berechnet aus einem gegebenen Schlüssel
Indexwert	Inhalt eines Array an dem gegebenen Index
Injektivität	Jeder Wert wird höchstens auf einen anderen abgebildet
Kollision	$k \neq k'$, $h(k) = h(k')$ Gleichheit der Hashwerte von zwei verschieden Schlüsseln
Kryptographie	Verschlüsselung
Linked Lists	Verkettete Listen
Modulo	Dividieren mit Rest
Quersumme	Addition aller Ziffernwerte einer Zahl
Subjektivität	Jeder Werte wird auf mind. einen anderen abgebildet

8 Quellenverzeichnis

Hashing, http://openbook.galileocomputing.de/c_von_a_bis_z/022_c_algorithmen_005.htm, 10.01.2014

C-Programmierung, http://de.wikibooks.org/wiki/C-Programmierung/* , 10.01.2014

glib-Hashtabellen, https://developer.gnome.org/glib/unstable/glib-Hash-Tables.html, 10.01.2014

glib-Prüfsummen, https://developer.gnome.org/glib/2.37/glib-Data-Checksums.html, 06.03.2014

Verkettete Listen, http://perlgeek.de/de/artikel/einfach-verkettete-listen, 06.03.2014

Hashfunktion, http://de.wikipedia.org/wiki/Hashfunktion, 10.01.2014

Hashtabelle, http://de.wikipedia.org/wiki/Hashtabelle, 10.01.2014

Adler32, http://de.wikipedia.org/wiki/Adler-32, 06.03.2014

SHA, http://de.wikipedia.org/wiki/Secure_Hash_Algorithm, 06.03.2014

GLib, http://de.wikipedia.org/wiki/GLib,06.03.2014

Injektivität, http://de.wikipedia.org/wiki/Injektivit, 06.03.2014

TPM, http://de.wikipedia.org/wiki/Trusted_Platform_Module, 06.03.2014

libgcrypt, https://www.gnu.org/software/libgcrypt/, 06.03.2014

ghash.c Sourcecode, https://www.opensource.apple.com/source/X11misc/X11misc-20/pkg-config/pkg-config-0.25/glib-1.2.10/ghash.c,06.03.2014

Big O-Notation, http://bigocheatsheet.com/, 19.01.2014

Alexander Koglin, Hashing, http://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2012_2013/epc-1213-koglin-hashing-praesentation.pdf, 10.01.2014

Intel SHA-Extension, http://software.intel.com/de-de/intel-isa-extensions,09.03.2014

libgcrypt, Internetseite mit Download- und Dokumentation-Links: https://www.gnu.org/software/libgcrypt/, 06.03.2014