Fehler in der dynamischen Speicherverwaltung finden

Gedächtnis-Training

Fehler in der Speicherverwaltung sind mit einem normalen Debugger kaum zu entdecken. Hilfe versprechen Mpatrol und Valgrind: Die Malloc-Debugger überwachen jede Memory-Aktion und warnen den Entwickler, wenn sein Programm Speicherlücken zurücklässt oder neben dem erlaubten Bereich schreibt. Herwart Kiram

Eine verzwickte Situation: Der Anwender beschwert sich, dass sein Programm etwa einmal pro Woche abstürzt. Leider lässt sich das Problem bei den Entwicklern nicht nachstellen und der User ist ein paar hundert Kilometer entfernt. Der hat natürlich keine Lust, sich selbst an einen Debugger zu setzen.

Kleine Ursache, späte Wirkung

Schuld sind häufig Fehler in der Speicherverwaltung. Sie zu entdecken ist besonders schwer, da die Symptome nicht eindeutig auf die Ursache hindeuten. Wenn eine Routine Daten an der falschen Adresse ablegt, wirkt sich das oft erst sehr viel später aus – es kann den Ablauf verändern, das Ergebnis verfälschen oder das Programm abstürzen lassen. Wie es dazu kam, lässt sich dann meist nicht mehr ermitteln.

Zum Glück gibt es mittlerweile etliche Tools, die dem Entwickler aus der Patsche helfen. Zwei sehr leistungsfähige Vertreter sind Mpatrol [1] und Valgrind [2]. Beide Libraries werden vor dem Programmstart mit Hilfe der »LD_PRELOAD«-Umgebungsvariablen gestartet und übernehmen zur Laufzeit die Kontrolle über die Applikation. Valgrind und Mpatrol prüfen und protokollieren die Speicheranforderung und Memory-Zugriffe. Außerdem können sie das Programm anhalten, wenn es fehlerhafte Speicherzugriffe ausführt.

Diese Tools erzeugen detaillierte Logfiles, die ungewöhnliche Vorgänge in der Speicherverwaltung aufzeichnen. Sie helfen damit dem Entwickler, jedes Problem nachzuvollziehen. Beide Libraries erledigen ihre Aufgabe allerdings auf höchst unterschiedliche Weise.

Ein kleines Testprogramm namens »error.c« ist in Listing 1 zu sehen, es demonstriert die Leistungsfähigkeit der Malloc-Debugger. Das Programm wertet den ersten Aufrufparameter aus – eine Ziffer zwischen 1 und 8 – und simuliert verschiedene Arten von Fehlern. In Tabelle 1 ist dargestellt, welches Argument zu welchem Bug führt. Wichtig ist, das Programm mit »gcc -g« zu kompilieren: Nur so enthält das Binary Debuginformationen. In den Logfiles steht dann statt der Speicheradresse die Zeilennummer des Quellcodes. Das erleichtert die Fehlersuche erheblich.

Mpatrol, der Memory-Wächter

Zunächst soll Mpatrol [1] zeigen, welche Fehler es findet. Die Installation dieses Tools gestaltet sich leider nicht ganz problemlos, die automatischen Installationsroutinen sind fehlerhaft und brechen mit einem Versionskonflikt im mitgelieferten Libtool ab. Aber es geht auch per Hand – der kleine Zusatzaufwand lohnt die Mühe allemal.

Auch für die manuelle Installation sind noch zwei Korrekturen nötig. Mpatrol geht davon aus, dass die Liberty-Bibliothek als Shared Library vorliegt. Unter Linux ist das in der Regel aber nicht der Fall. Die Lösung: »libiberty.a« fest zur Mpatrol-Bibliothek linken. Dazu ist eine Änderung am Makefile im Verzeichnis »mpatrol/build/unix« nötig. In dessen Zeile 127 ist als Ergänzung zusätzlich »-liberty« einzutragen:

```
$(LD) $(LDFLAGS) -o $@ $(SHARED_MPTOBJS) ➥
  -liberty
```

Auch ein Headerfile ist noch anzupassen: In »mpatrol/src/config.h« muss aus der Define-Anweisung in Zeile 686 der

Eintrag »MP_LIBNAME(iberty)« verschwinden. Danach sollte das Make-Kommando problemlos durchlaufen:

```
cd mpatrol/build/unix/
make all
```

Je nach Linux-System bereitet auch die BFD-Library Probleme. Sie greift ebenfalls auf Funktionen der Liberty-Bibliothek zurück, womit das »LD_PROLOAD« scheitert. Auch hier ist die Lösung: Im Makefile »-lbfd« hinzufügen und aus dem Config-Header »MP_LIBNAME (bfd)« entfernen. Wer den GUI-Support im »mptrace«-Kommando von Mpatrol nutzen will, muss in Zeile 39 noch das Makro »GUISUP« von »false« auf »true« ändern. Das setzt aber voraus, dass die X11-Entwicklerpakete installiert sind.

Installation mit Hürden

Zuletzt müssen die erzeugten Files noch an die richtigen Stellen gelangen. Das Skript aus Listing 2 vereinfacht diesen Vorgang – im Originalpaket ist leider nichts Vergleichbares zu finden. Beim Kompilieren entstehen insgesamt fünf Programme.

Es gibt zwei Techniken, um ein Programm unter Mpatrol ablaufen zu lassen. Zum einen lässt sich die Bibliothek per »LD_PRELOAD« vor dem Programmstart laden. Ein erneutes Kompilieren und Linken des Programms ist in dieser Variante nicht nötig. Alternativ kann der Entwickler die Mpatrol-Library auch beim Linken mit einbinden. Das Programm muss dazu die Mpatrol-Headerfiles verwenden; Änderungen im Sourcecode sind nicht erforderlich, der folgende GCC-Parameter genügt:

```
gcc -include /usr/local/include/mpatrol.h
    -c -g programm.c
```

Beim Linken ist zusätzlich »-lmpatrol -lbfd -liberty« anzugeben. Der Vorteil dieser Technik: Das Programm kann selbst Mpatrol-Routinen aufrufen. Die Funktion »int __mp_logaddr(const void *ptr)« zum Beispiel gibt eine Allozierungsstatistik der Speicherstelle »*ptr« aus. Das funktioniert natürlich nur, wenn »ptr« auf einen mit »malloc()« reservierten Bereich zeigt. Die Ausgabe dieser Funktion landet in der Datei »mpatrol.log«. Mpatrol enthält noch viele weitere nützliche Funktionen dieser Art. Die sehr ausführliche Dokumentation gibt darüber Auskunft.

Der Praxistest

Um ein Programm unter Mpatrol-Kontrolle zu starten, benutzt der Entwickler das gleichnamige Kommando: »mpatrol

Listing 1: Fehlersimulation

```
01 #include <stdio.h>
02 #include <stdlib.h>
03
04 char* up () {
05   char a='5';
06   return &a;
07 }
08
09 void access(void* ptr) {
10   *(char*)ptr=0;
11 }
12
13 int main(int argc, char **argv) {
14   char* c1_ptr;
15   char* c2_ptr;
16   int c;
17   c1_ptr = (char *)malloc(8);
18
19   if (argc!=2) return 0;
20   c = *argv[1];
21   switch (c) {
22     case '1':
23       c1_ptr=0;
24       break;
25     case '2':
26       free (c1_ptr);
27       free (c1_ptr);
28       break;
29     case '3':
30       free (c1_ptr);
31       access(c1_ptr);
32       break;
33     case '4':
34       free((void *)c2_ptr);
35       break;
36     case '5':
37       access(c2_ptr);
38       break;
39     case '6':
40       free (c1_ptr);
41       c2_ptr = up();
42       printf("%c\n",*c2_ptr);
43       break;
44     case '7':
45       access(c1_ptr+9);
46       break;
47     case '8':
48       free (c1_ptr);
49       break;
50   }
51   printf("Ende\n");
52   return 0;
53 }
```

Tabelle 1: Fehler im Beispielprogramm (Listing 1)

Parameter	Fehler
1	Gibt Speicher nicht frei (Speicherleck)
2	Gibt Speicher mehrfach frei
3	Schreibt auf bereits freigegebenen Speicher
4	Gibt Speicher frei, der gar nicht alloziert wurde
5	Schreibt auf beliebigen Speicher
6	Benutzt den Stack eines bereits verlassenen Unterprogramms
7	Schreibt über den allozierten Speicher hinaus
8	Kein Fehler

Listing 2: Installationsskript

```
01 # Source- und Target-Directory:
02 MPATROL=~/mpatrol
03 PREFIX=/usr/local
04
05 # Libs kopieren und Ldconfig aufrufen
06 cd $MPATROL/build/unix/
07 cp -v *.so.1.4 $PREFIX/lib/
08 cd $PREFIX/lib/
09 ln -vs libmpalloc.so.1.4 libmpalloc.so
10 ln -vs libmpatrol.so.1.4 libmpatrol.so
11 ln -vs libmpatrolmt.so.1.4 libmpatrolmt.so
12 ldconfig $PREFIX/lib/
13
14 # Programme und Tools kopieren
15 cd $MPATROL/build/unix/
16 cp -v mpatrol mprof mptrace mleak $PREFIX/bin/
17 cd $MPATROL/bin/
18 cp -v mpsym mpedit hexwords $PREFIX/bin/
19
20 # Headerfiles kopieren
21 cd $MPATROL/src/
22 cp -v mpatrol.h mpalloc.h mpdebug.h
   $PREFIX/include/
23 mkdir $PREFIX/include/mpatrol
24 cp -v $MPATROL/tools/*.h
   $PREFIX/include/mpatrol/
25
26 # Manpages kopieren
27 cp -v $MPATROL/man/man1/* $PREFIX/man/man1/
28 cp -v $MPATROL/man/man3/* $PREFIX/man/man3/
```

Abbildung 1: Das Mpatrol-Logfile zeigt die Ursache des Absturzes, »mpsym« ergänzt die Namen der betroffenen Funktionen und die Zeile im Quellcode.

Mpatrol-Optionen Programm Programm-Argumente«. Der Aufruf »mpatrol -h« zeigt alle Optionen. Für das Error-Programm aus Listing 1 passt folgender Aufruf:

```
mpatrol -d -B --leak-table error 7
```

Die Option »-d« bindet die Mpatrol-Bibliothek per »LD_PRELOAD« ein, »--leak-table« listet am Ende des Programmlaufs alle Speicherlecks und »-B« legt geschützte Speicherseiten an.

Wird das Error-Programm mit der Option »7« aufgerufen, schreibt es über den allozierten Speicher hinaus. Am Ende des Programms produziert Mpatrol ein Logfile namens »mpatrol.*PID*.log«. In dieser Datei stehen alle Informationen zum Programmablauf sowie die Speicherstatistik und vieles mehr.

Vor der Analyse des Logfiles ersetzt »mpsym *Optionen Programmdatei Logfile*« die Speicheradressen im Logfile durch die Zeilennummern der Quelldatei, für obiges Beispiel lautet der Aufruf:

```
mpsym error mpatrol.1234.log
```

Im Abbildung 1 ist das resultierende Logfile inklusive Mpsym-Ergänzungen zu sehen. In »error.c:45« (Zeile 45, siehe Listing 1) schreibt das Programm über die Grenzen des Speicherbereichs hinaus, den es in Zeile 17 (»error.c:17«) alloziert hat. Leider zeigt Mpatrol aber nicht die unterste Ebene der Aufrufhierarchie, sondern nur die zweite: Die »access()«-Funktion (Zeile 10) ist die tatsächliche Quelle.

Nun sind die weiteren Fehlermöglichkeiten des Error-Programms an der Reihe: Mpatrol entdeckt die Bugs Nummer 1 bis 5. Lediglich Fehler 6 bemerkt es nicht, das Programm benutzt hier über einen Zeiger den Stackframe des bereits verlassenen Unterprogramms »up()«.

Funktionsweise

Mpatrol kennt zwei Techniken, um Verletzungen von Speicherbereichsgrenzen zu erkennen. Es leitet die »malloc()«- und »free()«-Aufrufe auf eine eigene Routine um. Diese fordert mehr Memory an als der Malloc-Aufruf. Den zusätzlichen Speicher füllt Mpatrol mit Schutz-Bytes, die eine magische Zahl enthalten. Wenn das Programm den Speicher wieder an das Betriebssystem zurückgibt oder sich beendet, prüft Mpatrol, ob die Schutzbytes unversehrt sind. Wenn nicht, trat offensichtlich eine Speicherbereichsverletzung auf.

Der Nachteil dieser Strategie: Mpatrol kann die Programmzeile, die den fehlerhaften Speicherzugriff verursacht hat, nicht exakt ermitteln. Deshalb verwendet es noch eine andere Strategie, um Speicherbereichsverletzungen zu erkennen. Mpatrol sorgt dafür, dass jeder Malloc immer mindestens zwei Speicherseiten vom Betriebssystem anfordert. Innerhalb der ersten Seite liegt der vom Programm gewünschte Speicher, er schließt mit dem Seitenende ab (Abbildung 2, rechts). Die dahinter liegende Speicherseite markiert Mpatrol als lese- und schreibgeschützt.

Wenn das Programm auf Speicherplatz zugreift, der hinter dem allozierten Bereich liegt, landet dieser Zugriff auf der geschützten Speicherseite. Die Memory Management Unit (Kasten „MMU“) löst daraufhin einen SIGSEGV-Interrupt aus (Segmentation Fault). Dieses Signal wird von einem Handler in Mpatrol aufgefangen. Er analysiert den Stack des Programms und stellt die Aufrufhierarchie der Unterprogramme zum Zeitpunkt des Zugriffsfehlers fest. So erfährt er die Zeile, die den Fehler ausgelöst hat.

MMU (Memory Management Unit)

Bei früheren Computerarchitekturen entsprachen die Adressen der CPU direkt den RAM-Adressen (i286). Heute gibt es das Konzept des virtuellen Speichers: Der Adressbereich, den ein Prozess verwendetet, stimmt nicht mit den physikalischen Adressen im Speicher überein. Eine Hardwarekomponente, die MMU, übersetzt automatisch die virtuellen Adressen der CPU in die physikalischen Adressen der RAM-Bausteine.

Dieser aufwändige Umweg hat einige Vorteile. Zum Beispiel kann dadurch ein Prozess mehr Speicher benutzen als tatsächlich im Computer eingebaut ist. Das System lädt nicht das ganze Programm auf einmal in den Speicher, sondern nur den gerade ausgeführten Teil. Wenn das Programm auf Speicheradressen zugreift, die sich nicht im RAM befinden, löst die MMU eine Exception aus und ein Prozesswechsel findet statt. Während ein anderer Prozess abläuft, lädt das Betriebssystem die fehlenden Programm- oder Datenteile des ersten Prozesses in den Speicher. Zum Ausgleich lagert es gerade nicht benötigte Programmteile auf die Festplatte aus (Paging). Danach kann der erste Prozess weiterlaufen.

Der segmentierte Adressraum

Der Adressraum sieht also nur aus der Sicht der CPU linear und sehr groß aus. In Wirklichkeit ist er viel kleiner und stark segmentiert. Die kleinste Einheit, die eine MMU verwaltet, ist die so genannte Seite oder Page. Sie ist häufig 4 KByte groß. Ein weiterer Vorteil der MMU: Die Seiten lassen sich einzelnen Prozessen zuordnen. Wenn ein Prozess auf eine Seite außerhalb seines eigenen Adressraums zugreifen will, löst dieser Fehlgriff einen Segmentation Fault aus (SIGSEGV).

Viele Malloc-Debugger nutzen dies aus und markieren Speicherseiten als geschützt. Wenn ein fehlerhaftes Programm darauf zugreift, löst die MMU einen Interrupt aus. Jetzt kann ein Interrupthandler den Stack analysieren.

Mpatrol nutzt dieses Verfahren, wenn es mit der Option »-B« (oberer Speicherbereich) oder »-b« (unterer Bereich) aufgerufen wird. Diese Strategie hat allerdings den Nachteil, dass sie wesentlich mehr Speicher beansprucht.

Valgrind

Ein sehr interessanter, von Julian Seward neu entwickelter Malloc-Debugger ist Valgrind [2]. Dieses Tool lässt sich problemlos installieren: entpacken und »./configure && make && make install« aufrufen genügt. Valgrind wird wie das Mpatrol-Kommando aufgerufen: »valgrind *Valgrind-Optionen Programm Programm-Argumente*«. Ein Testlauf mit dem Fehlerprogramm aus Listing 1 soll den falschen Heap-Zugriff bemerken:

```
valgrind --leak-check=yes --logfile=elog ➥
  ./error 7
```

Die Option »--leak-check = yes« sorgt dafür, dass Valgrind Speicherlecks findet, und »--logfile = elog« schreibt Ausgaben in das Logfile »elog.pid*22521*« (die Zahlen entsprechen der Prozess-ID). Alle Optionen, die Valgrind unterstützt, listet der Aufruf »valgrind --help«.

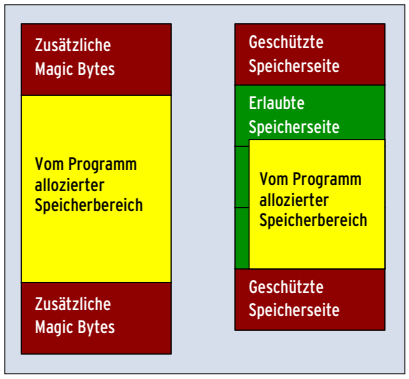

Abbildung 2: Malloc-Debugger nutzen zwei Methoden, um Speicherbereichsverletzungen zu erkennen: Sie fügen so genannte magische Bytes ein (links) oder schützen ganze Memory Pages (rechts).

Das sehr ausführliche Logfile (Listing 3) informiert – nach ein paar einleitenden Worten zum Copyright, den Autor und die vermutete CPU-Taktung – über die Fehler, die Valgrind gefunden zu haben glaubt. Den plumpen Zugriff über den allozierten Heap hinaus erkennt es sofort, ebenso alle anderen Fehler des Error-Programms. Die ausführlichen Kommentare helfen besonders bei den ersten Versuchen mit diesem Tool.

Valgrind ist eigentlich ein Code-Instrumentierungstool. Der Befehl »valgrind« (ein Shellskript) setzt die Umgebungsvariable »LD_PRELOAD« auf die Bibliothek »valgrind.so« und linkt diese Library somit zu jedem dynamisch gelinkten ELF-Binary. Im vorliegenden Fall also zu dem Programm, das debuggt werden soll.

Künstliche CPU

Der dynamische Linker erlaubt es jeder Bibliothek, eine Initialisierungsfunktion anzugeben. Diese Funktion kommt zum Zuge, noch bevor »main()« ausgeführt wird. Nach dem Ende der »main()«-Funktion darf jede Library noch eine Finalisierungsfunktion einsetzen.

Die Initialisierungsfunktion der Valgrind-Library ruft eine Art synthetische CPU ins Leben, die ab jetzt den Code des Programms abarbeitet. Sie führt den Maschinencode allerdings nicht direkt aus, sondern wandelt ihn zuerst in einen Zwischencode um, den so genannten UCode. Die einzelnen Schritte sind:

- Valgrind liest einen Block von x86-Anweisungen und wandelt ihn in UCode um.
- Das Werkzeug instrumentiert den UCode mit zusätzlichen Anweisungen, die Werte und Adressen prüfen.

Electric Fence und Libcwd

Neben Mpatrol und Valgrind gibt es noch viele andere Memory-Debugger. Zuerst ist der Klassiker Electric Fence [3] von Bruce Perens zu nennen. Efence ist in den meisten Distributionen enthalten und funktioniert ähnlich wie Mpatrol. Auch hier linkt der Entwickler sein Programm statisch mit »libefence.a« oder er startet es mit dem Befehl »ef *Programm Programm-Optionen*« (Efence-Version 2.2.2).

Electric Fence – Klassiker von Bruce Perens

Ähnlich wie »mpatrol« nutzt auch »ef« das »LD_PRELOAD«-Verfahren und lädt die Bibliothek »libefence.so.0.0«. Sie fängt alle »malloc()«-Aufrufe ab und leitet sie auf die eigenen Routinen um. Vor oder hinter dem allozierten Speicher legt Efence geschützte Speicherseiten an, die bei einem Zugriff einen Segmentation Fault auslösen (siehe Kasten „MMU"). Läuft das Programm in einem Debugger, kann der Entwickler somit bequem nach dem Problem suchen.

Mehrere Umgebungsvariablen beeinflussen das Verhalten von Efence. Bei »EF_PROTECT_BELOW=1« schützt das Tool den Bereich unterhalb des allozierten Speichers. Per Default schützt es den Bereich oberhalb. Wer den Verdacht hegt, dass sein Programm auf bereits freigegebene Speicherblöcke zugreift, setzt die Umgebungsvariable »EF_PROTECT_FREE =1«. Damit gibt Efence bei »free()« die Speicherbereiche nicht mehr an das Betriebssystem zurück, sondern markiert sie als lese- und schreibgeschützt. Zugriffe darauf führen wieder zu einem Segmentation Fault. Je nach Programm kann dieses Verfahren aber zu enormem Speicherverbrauch führen.

Die Umgebungsvariable »EF_FILL« sorgt dafür, dass Efence jeden angeforderten Speicher mit dem Wert der Variablen (zwischen 0 und 255) füllt. Lesezugriffe auf nicht initialisierten Speicher sind so leichter zu finden. Efence funktioniert übrigens auch mit den »new«- und »delete«-Operatoren von C++.

Libcwd – Debugging-Kanäle inklusive

Libcwd [4], eine C++-Debugging-Bibliothek von Carlo Wood, implementiert eine eigene Technik, um Speicherlecks und Dangling Pointer zu entdecken. Diese Library stellt im Wesentlichen zwei Grundfunktionalitäten zur Verfügung: Stream-basierte Kanäle für Debugausgaben sowie einige mächtige Funktionen für das Überprüfen und Loggen der Speicherallozierungen.

Beim Debugging ist es oft erforderlich, an verschiedenen Stellen im Programm Ausgaben zu machen. Einerseits sind möglichst viele Informationen zur Laufzeit wünschenswert, andererseits wächst das Logfile dann sehr schnell und wird unübersichtlich. Am besten ist es, verschiedene Ausgabekanäle zu definieren, die man nach Bedarf ein- und ausschaltet. Die Kanäle lassen sich bereits zu Beginn des Programms, aber auch während des Programmablaufs aktivieren.

Die Speicherprüfung von Libcwd stellt folgende Funktionen bereit:

- Sie überprüft Zeiger auf freizugebende Speicherblöcke,
- informiert über den Ort der Allozierung eines Speicherblocks im Sourcecode sowie über seinen Beginn, die Größe und den Typ,
- führt eine Statistik über alle allozierten Speicherblöcke und
- überprüft, ob bei freigegebenen Memory-Blöcken eventuell Speicherbereichsverletzungen aufgetreten sind.

- Valgrind wandelt den UCode wieder in x86-Anweisungen zurück und führt diese aus (im Stil eines JIT-Compilers).

Der entscheidende zweite Schritt enthält die eigentliche Prüffunktionalität. Die Art der Instrumentierung ist interessanterweise nicht fest in Valgrind einprogrammiert, sie ist vielmehr durch so genannte Skins selbst definierbar. Dadurch ist es möglich, die unterschiedlichsten Prüfaufgaben zu realisieren.

Es gibt bereits eine Menge Skins, die man sofort benutzen kann, siehe Tabelle 2. Um eine bestimmte Skin auszuwählen, ist die Option »--skin« zu benutzen. Ohne diese Option verwendet Valgrind automatisch die »memcheck«-Skin.

Mit wechselnden Häuten zum Universalwerkzeug

Valgrind wird durch die Skins zum universellen, frei programmierbaren Programmprüfer. Skins kann jeder Programmierer auch selbst schreiben, eine genaue Anleitung dafür gibt die sehr ausführliche Dokumentation.

Beim Debuggen mit Valgrind tritt das Problem auf, dass das Werkzeug nicht nur die Fehler im gerade untersuchten Programm findet, sondern auch die Bugs in den Bibliotheken, die die Applikation verwendet. Da man sich aber nicht unbedingt mit den Problemen anderer Leute befassen will, kann Valgrind Ausschlusslisten verwenden. Sie steuern sehr präzise, welche Fehler das Tool anzeigt und welche es ignoriert.

Beim Installieren von Valgrind werden einige Ausschlusslisten automatisch erstellt. Sie liegen im selben Verzeichnis wie die Valgrind-Bibliothek, üblicherweise »/usr/lib/valgrind« oder »/usr/local/lib/valgrind«. Sie sind an der Extension ».supp« zu erkennen. Unter anderem unterdrücken sie Fehler, die aus der Glibc und aus XFree resultieren. Die Datei »default.supp« enthält eine leider etwas zu kurz geratene Anleitung, wie Ausschlusslisten aufgebaut sind.

Tabelle 2: Valgrind-Skins

Skin	Aufgabe
memcheck	Ein sehr leistungsfähiger Memory Checker
addrcheck	Ein einfacher Memory Checker
helgrind	Findet Race-Bedingungen
cachgrind	Ein Cache-Miss-Profiler

Bestimmte Fehler absichtlich ignorieren

Ein wenig eigenes Experimentieren ist also anzuraten, allerdings helfen dabei die Dokumentation sowie die Option »--gen-suppressions = yes«: Valgrind hält dann bei jedem Fehler an und fragt den Entwickler, ob er diesen Bug künftig ignorieren will. Wenn ja, gibt Valgrind eine passende Suppression aus, die der Entwickler dann in seine Ausschlusslisten kopieren kann.

Das Hauptproblem von Valgrind ist seine schwache Performance: Programme laufen unter Valgrind-Kontrolle sehr langsam. Mit der Memcheck-Skin lief ein Testprogramm etwa 27-mal langsamer als normal, mit Addrcheck etwa 23-mal. Diese Werte variieren zwar je nach Programm, sind bei rechenintensiver Software aber problematisch. Die Ursache ist, dass das Programm zur Laufzeit instrumentiert werden muss. Einen anderen Grund sieht Julian Seward darin, dass es aufgrund der Art der Program-

Listing 3: Ausschnitt der Valgrind-Ausgabe

```
==22521== Memcheck, a.k.a. Valgrind, a memory error detector for x86-linux.
==22521== Copyright (C) 2002-2003, and GNU GPL'd, by Julian Seward.
==22521== Using valgrind-2.0.0, a program supervision framework for x86-linux.
==22521== Copyright (C) 2000-2003, and GNU GPL'd, by Julian Seward.
==22521==
==22521== My PID = 22521, parent PID = 1606.  Prog and args are:
==22521==    ./error
==22521==    7
==22521== Estimated CPU clock rate is 451 MHz
==22521== For more details, rerun with: -v
==22521==
==22521== Invalid write of size 1
==22521==    at 0x8048506: access (error.c:10)
==22521==    by 0x402677ED: __libc_start_main (in /lib/libc.so.6)
==22521==    by 0x8048400: (within /home/fjl/lm/2004/02/mdebug/src/error)
==22521==    Address 0x4108E02D is 1 bytes after a block of size 8 alloc'd
==22521==    at 0x4002A39F: malloc (vg_replace_malloc.c:153)
==22521==    by 0x804851F: main (error.c:17)
==22521==    by 0x402677ED: __libc_start_main (in /lib/libc.so.6)
==22521==    by 0x8048400: (within /home/fjl/lm/2004/02/mdebug/src/error)
==22521==
==22521== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
==22521== malloc/free: in use at exit: 8 bytes in 1 blocks.
==22521== malloc/free: 1 allocs, 0 frees, 8 bytes allocated.
==22521== For counts of detected errors, rerun with: -v
==22521== searching for pointers to 1 not-freed blocks.
==22521== checked 3528428 bytes.
==22521==
==22521==
==22521== 8 bytes in 1 blocks are definitely lost in loss record 1 of 1
==22521==    at 0x4002A39F: malloc (vg_replace_malloc.c:153)
==22521==    by 0x804851F: main (error.c:17)
==22521==    by 0x402677ED: __libc_start_main (in /lib/libc.so.6)
==22521==    by 0x8048400: (within /home/fjl/lm/2004/02/mdebug/src/error)
==22521==
==22521== LEAK SUMMARY:
==22521==    definitely lost: 8 bytes in 1 blocks.
==22521==    possibly lost:   0 bytes in 0 blocks.
==22521==    still reachable: 0 bytes in 0 blocks.
==22521==         suppressed: 0 bytes in 0 blocks.
==22521== Reachable blocks (those to which a pointer was found) are not shown.
==22521== To see them, rerun with: --show-reachable=yes
==22521==
```

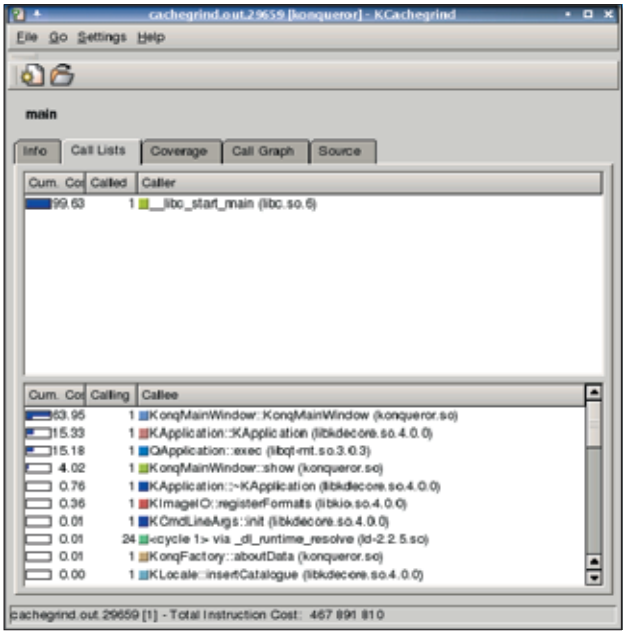

Abbildung 3: KCachegrind stellt in diesem Beispiel die Aufrufhierarchie von Konqueror dar. Es zählt für jede Funktion, wie oft sie aufgerufen wurde und welchen Anteil an der Gesamtlaufzeit des Programms sie hat.

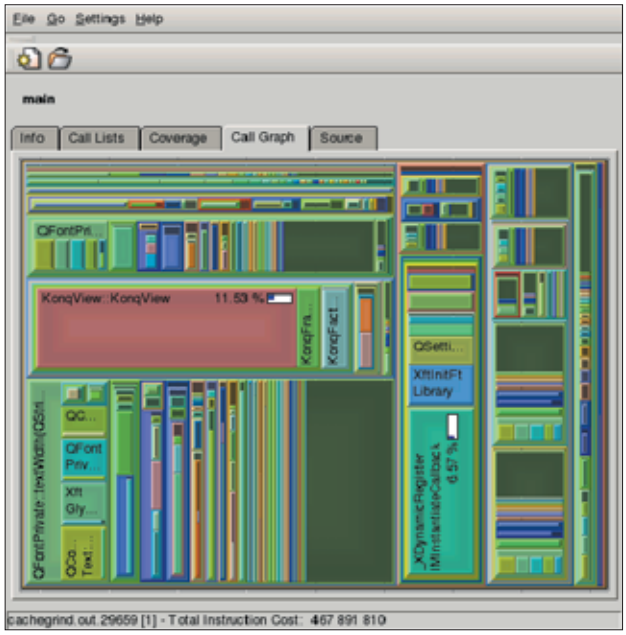

Abbildung 4: Die Aufrufhierarchie lässt sich auch grafisch mit ineinander verschachtelten Rechtecken darstellen. Die Größe der Flächen entspricht der Rechenzeit, die die jeweilige Funktion benötigt hat.

mierung zu vielen Cachefehlern kommt. Letzteres kann sich aber in künftigen Versionen verbessern.

Laufzeitoptimierung mit KCachegrind

Eine wichtige Aufgabe beim Programmieren ist das Optimieren. Dafür benötigt der Entwickler Informationen, wie oft ein Unterprogramm eine andere Funktion aufgerufen hat. Subroutinen, die besonders häufig aufgerufen werden, sind Kandidaten für eine Laufzeitoptimierung. Hier ist die KCachegrind-Skin »calltree« ein praktischer Helfer, sie erzeugt ein Aufrufprofil des Programms. Die Ausgabe dieser Skin ist allerdings nicht leicht zu lesen.

Auch hier haben die Entwickler für Abhilfe gesorgt: KCachegrind visualisiert die Calltree-Ausgaben. Beide Programme stehen unter [8] bereit. Die Installation gestaltet sich weitgehend problemlos, eventuell benötigt Configure aber Root-Rechte. Voraussetzung sind QT 3 (Entwicklerversion, mit Thread-Support übersetzt), Valgrind 1.9.6 oder höher sowie die KDE-Bibliotheken, ebenfalls mit Entwicklerpaketen.

Wenn alles richtig installiert ist, kann man ein Programm unter der Calltree-Skin ablaufen lassen. Als Beispiel musste Konqueror sein Innerstes offenbaren. Valgrind produziert jetzt eine Ausgabedatei namens »cachegrind.out.*PID*«, PID ist die Prozessnummer. KCachegrind kann diese Datei laden:

```
kcachegrind cachegrind.out.PID
```

Unter der Karteikarte »CallLists« ist die Aufrufhierarchie als Liste dargestellt (siehe Abbildung 3). Man kann in ihr auf und ab navigieren. Ein etwas umfassenderes, aber manchmal auch verwirrendes Bild offeriert die Karteikarte »CallGraph«. Sie stellt die Aufrufhierarchie als Rechteckgrafik dar, das Ergebnis ist in Abbildung 4 zu sehen. Um den Überblick zu behalten, begrenzt ein Eintrag im Kontextmenü die Schachtelungstiefe auf ein erträgliches Maß.

Fazit

Im Rahmen eines modernen Software-Entwicklungsprozesses sind leistungsfähige Tools zum Auffinden von Fehlern unabdingbar. Trotz der einen oder anderen kleinen Schwäche sind Mpatrol und Valgrind uneingeschränkt praxistauglich. Sie ergänzen sich gegenseitig optimal. Valgrind ist zwar langsamer, findet dafür aber mehr Probleme und braucht weniger RAM, um fehlerhafte Speicherzugriffe zur Laufzeit zu entdecken. Zudem ist Valgrind durch die Skins wesentlich vielseitiger.

Mpatrol ist dagegen schneller, es verzögert den Programmablauf kaum spürbar. In Kombination mit dem Codeprüfer SPLint ([6], [7]) stehen auf allen Stufen des Entwicklungsprozesses leistungsfähige Tools zur Verfügung, die bei der Fehlersuche helfen. *(fjl)* ■

Infos

[1] Mpatrol: [http://www.cbmamiga.demon.co.uk/mpatrol/]
[2] Valgrind: [http://valgrind.kde.org]
[3] Electric Fence: [http://perens.com/FreeSoftware/]
[4] Libcwd: [http://libcwd.sourceforge.net/]
[5] Daniel P. Bovet und Marco Cesati, „Understanding the Linux Kernel“: O'Reilly
[6] Steven Goodwin und Dean Wilson, „Flusen-Sieb – Code-Qualität mit Splint überprüfen“: Linux-Magazin 5/03, S. 90
[7] Herwart Kiram, „Vollwaschmittel – Code-Qualität mit Splint überprüfen, Teil 2“: Linux-Magazin 6/03, S. 53
[8] KCachegrind: [http://kcachegrind.sf.net/]

Der Autor

Herwart Kiram arbeitet seit zehn Jahren als Software-Entwickler in der Telekommunikationsindustrie. Seine Spezialgebiete sind Linux und Datenkommunikationsprotokolle.