Universität Hamburg
Fakultät für Mathematik,
Informatik und Naturwissenschaften

Seminararbeit

System monitoring mit STRACE

Lars Grote

lagrote@gmail.com
Bachelor of Science, Informatik
Matr.-Nr. 5811957
Fachsemester 6

Betreuende Personen:
Prof. Dr. Thomas Ludwig
Julian Kunkel
Michael Kuhn

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair.
– *Douglas Adams*

Inhaltsverzeichnis

1 Einleitung

System monitoring mit STRACE lautet das Thema dieser Arbeit. Das Werkzeug das Untersucht wird ist das Programm und Titelgeber, STRACE. STRACE ist ein Kommandozeilenprogramm, das in fast allen LINUX Distributionen mitgeliefert wird. Andere Betriebssysteme haben vergleichbare Programme. Diese heißen nicht STRACE sondern DTRACE (SOLARIS), KTRACE (MACOS, ab 10.5 DTRACE) oder STRACENT (WINDOWS) [Wikf].

Es ist beeindruckend was mit STRACE über ein Programm herausgefunden werden kann, das nur in seiner kompilierten Form vorliegt: welche Dateien werden geöffnet, mit welchen Prozessen wird kommuniziert, welche neuen Prozesse werden erzeugt, wohin wird geschrieben usw.
Um zu verstehen wieso es möglich ist so viel mit STRACE herauszufinden, reicht es nicht aus sich lediglich STRACE anzusehen. Es ist nötig sich mit Systemcalls zu beschäftigen, und um die Notwendigkeit und Funktionsweise von Systemcalls zu begreifen, ist es nötig sich mit Kernelarchitektur auseinanderzusetzen.

Aus diesen, dem Verständnis geschuldeten Notwendigkeiten, ergeben sich die Themen dieser Arbeit: Kernelarchitektur, Systemcalls, STRACE.

2 Einführung Kernelarchitektur

Das Programm STRACE verfolgt die Systemcalls von Prozessen. Systemcalls sind das Kommunikationsmittel zwischen Benutzer-Applikationen und dem Kernel. Um dieses Zusammenspiel und diese Abhängigkeit zu verstehen ist ein grundlegendes Verständnis des Kernels nötig. Für einen Anwendungsprogrammierer ist der Kernel eine transparente Schicht. Die Existenz ist zwar immer klar, die Abgrenzung des Kernels jedoch oftmals diffus [Mau04, S.557].

Der Kernel ist das Herz des Betriebssystems. Wollte man sich ihm in angemessenem Umfang nähern, so sollte die Anzahl, der dafür verwendeten Seiten, in die Hunderte gehen. Deshalb wird dieser Versuch hier nicht unternommen. Da das Thema dieser Arbeit STRACE ist, wird im Folgenden auf die, hierfür relevanten Aufgaben des Kernels und anschließend auf die Schnittstelle zu den Anwendungen eingegangen. Erst genanntes dient dem allgemeinen Verständnis, Letzteres ist die Grundlage auf der STRACE arbeitet.

2.1 Aufgaben des Kernels

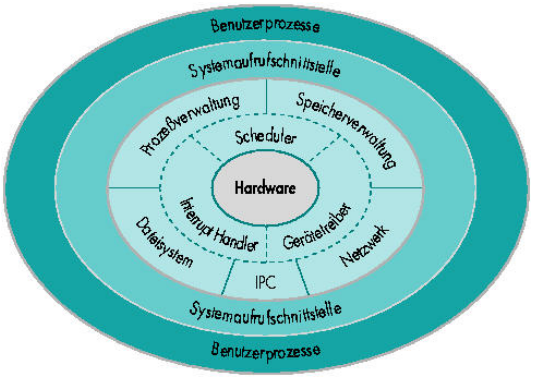
Die zentrale Aufgabe des Kernels ist die Vermittlung zwischen Hard– und Software. Der Kernel abstrahiert von der Hardware und stellt ein Interface bereit. Dieses Interface ist im Wesentlichen plattformunabhängig und ermöglicht so die Portabilität von Anwendungssoftware.[1] Im Umkehrschluss bedeutet das aber auch, dass der Kernel in hohem Maße plattformabhängig ist.[2] Nach außen stellt der Kernel Funktionen zur Verfügung, die oftmals sehr deklarativ sind, was den hohen Abstraktionsgrad deutlich werden lässt. Ein gutes Beispiel ist das Laden einer Datei. Die Anwendungssoftware teilt dem Kernel mit, dass sie eine bestimmte Datei öffnen will, dazu wird lediglich der Pfad angegeben. Der Kernel muss sich nun darum kümmern, auf welchem Weg die Datei von der Festplatte in den Speicher gelangt, welche Kommandos, in welcher Reihenfolge an die Festplatte geschickt werden müssen usw.[Mau04, S.2f.]

Ein weiterer wichtiger Aspekt des Kernels ist das Ressourcen Management. In dem Moment, in dem CPU, Speicher, Festplatte etc. von mehreren Prozessen verwendet werden, muss der Kernel die Ressourcen unter den Prozessen aufteilen und für einen möglichst reibungslosen Ablauf sorgen. Diese Aufgaben gehören zu dem Innersten des Kernels. Sie Stellen die serielle Ordnung her, die für die konkreten Aufrufe an die Hardware

[1] Für das angesprochene Interface besteht ein Standard: Der POSIX Standard. Auf diesen wird in Kapitel 3 weiter eingegangen.

[2] Die Plattform von der hier die Rede ist, ist die jeweilige Prozessor Architektur.

Abbildung 2.1: Schichten des Kernels [Han]



nötig ist [Mau04, S.3f.]. Im Falle der CPU ist das Management aufwändig, da es unterschiedliche Möglichkeiten gibt diese Ressourcenzuteilung zu realisieren. Die Einheit die dieses tut wird *Scheduler* genannt (siehe auch Abbildung 2.1).

Das das Memory Management für STRACE von besonderer Bedeutung ist, wird es an dieser Stelle erwähnt. Der Kernel abstrahiert an dieser Stelle sehr stark von dem physikalischen Speicher. Es wird ein komplett unabhängiger Adressraum gegeben, bei 32bit Systemen lassen sich 2^{32} Speicheradressen darstellen, bei 64bit Systemen 2^{64}. Dieser Adressraum ist ansprechbar, unabhängig davon wie viel Speicher tatsächlich in dem System vorhanden ist.[3] Auf LINUX (UNIX) Systemen wird dieser Arbeitsspeicher in zwei voneinander getrennte Adressräume geteilt. Den Kernelspace und den Userspace [Mau04, S.6f.].

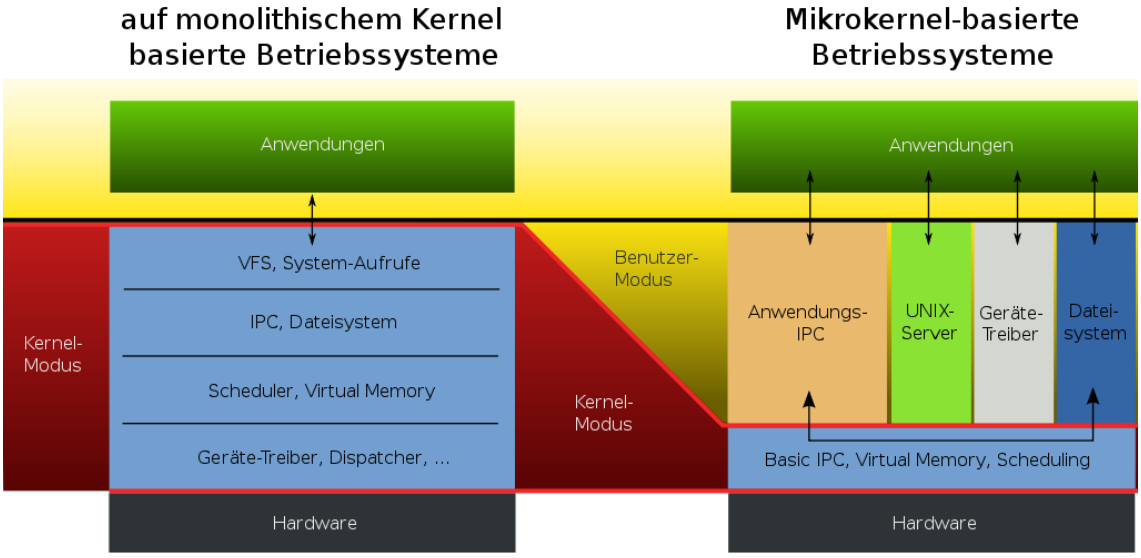
Hinter all den genannten Aufgaben verbirgt sich der Gedanke, den Anwendungsprogrammen eine Umgebung und Basis zu geben, auf der sie zum einen leichter programmiert und zum anderen zuverlässig ausgeführt werden können. Die Programmierung wird durch die zu Verfügung gestellten, einheitlichen Funktionen – die Abstraktion – vereinfacht. Das Ressourcenmanagement sorgt für eine Umgebung, in der die Anwendungsprogramme sicher, ohne Systemabstürze und ohne sich gegenseitig zu stören, laufen können.

2.2 Kernel Implementationen

Ein Kernel lässt sich auf unterschiedliche Art und Weise implementieren. Es gibt sicher viele Modelle wie man dieser Aufgabe begegnen kann, jedoch gibt es zwei Konzepte die besonders zu erwähnen sind: Monolithische Kernel und Micro Kernel. Mittlerweile dienen diese beiden Ansätze zur wesentlichen Klassifizierung von Kernel Implementationen. Besonders im erweiterten Umfeld der UNIX Welt gibt es viele unterschiedliche

[3] Wird mehr Speicher angesprochen als in dem System wirklich vorhanden ist, so wird ein Teil mittels Swapping auf die Festplatte ausgelagert (auf LINUX-Systemen auf die SWAP Partition) [Mau04, S690ff.].

Abbildung 2.2: Monolithischer und Micro Kernel [Wikd]



Kernelvarianten, die entweder eines der beiden Dogmen vertreten oder eine hybride Variante darstellen.

2.2.1 Monolithische Kernel

Kernel die alle Kernelaufgaben, das heißt nicht nur Speicher- und Prozessverwaltung, sondern auch andere wie Treiber, Interprozesskommunikation (IPC) und Netzwerk, direkt eingebaut haben, nennt man Monolithische Kernel. Diese Kernel haben den Vorteil, dass für die einzelnen Aufgaben keine zusätzlichen Programme benötigt werden, was einen Geschwindigkeitsvorteil mit sich bringt. Damit hängt zusammen, dass alle Kernelaufgaben innerhalb des Kernelspaces (dem Speicheradressraum, der für Kernelaufgaben reserviert ist) bearbeitet werden. Somit muss weniger oft zwischen Kernelspace und Userspace hin und her geschaltet werden. Monolithische Kernel haben aber auch einige Nachteile. Aus der Sicht der Software Technik oder der Software Modellierung wäre es übersichtlicher den Kernel entsprechend der Aufgabenbereiche in *Services* (Prozesse) zu teilen [Wikd]. Darüber hinaus sind Monolithische Kernel anfälliger gegenüber Systemabstürzen, etwa durch einen fehlerhaft programmierten Treiber, da auch die Treiber im Kernelspace laufen. LINUX begegnet vielen der Probleme durch ein trickreiches Modul System, auf das an dieser Stelle nicht weiter eingegangen wird [Mau04, S.15]. Betriebssysteme, die einen Monolithischen Kernel verwenden [Wikd]:

- UNIX
- LINUX
- BSD

- DOS / WINDOWS bis zur Version ME

2.2.2 Micro Kernel

Ein Kernel, der nur die notwendigsten Funktionen, wie Speicher- und Prozessverwaltung direkt bereitstellt und alle anderen Funktionen durch extra Prozesse realisiert, nennt man Micro Kernel. Die Idee des Micro Kernels kam auf, weil die Monolithischen Kernel größer und damit immer schwieriger zu pflegen wurden. Das Konzept soll den wesentlichen Teil des Kernels klein und beherrschbar halten und die übrigen Aufgaben in extra Prozessen realisieren, die unabhängig entwickelt werden können. Diese extra Prozesse sind für Aufgaben wie: Unix-Server, Gerätetreiber, Dateisystem etc. zuständig. Abgesehen von der modulareren Struktur hat dieses Konzept den Vorteil, dass die extra Prozesse im Userspace laufen können. Das macht den Kernel selber unanfälliger gegenüber Systemabstürzen, da z.B. beim Absturz des Dateisystems einfach der Prozess des Dateisystems neugestartet werden kann. In der Praxis hat sich jedoch gezeigt, dass Betriebssysteme, die auf einem Micro Kernel basieren, nicht nennenswert robuster sind als solche mit Monolithischem Kernel [Wike]. Micro Kerneln wird oft ein Geschwindigkeitsnachteil nachgesagt, da sie oft zwischen Kernelspace und Userspace wechseln müssen und viel Interprozesskommunikation (IPC) nötig ist. Jedoch gibt es Beispiele für gut designte Micro Kernel, die mit der Performance von Monolithischen Kerneln mithalten können, ein Beispiel ist der L4[4] Kernel von Jochen Liedtke [HHL+97]. Betriebssysteme, die einen Micro Kernel verwenden [Wike]:

- Mach
- L4
- Nucleus
- Minix

2.2.3 Hybrid Kernel

Als letztes sei noch die Mischform der beiden zuvor genannten Kernelarchitekturen genannt: der Hybrid Kernel. Dieser versucht die Vorteile des Monolithischen und des Micro Ansatzes zu verbinden. Generell lässt sich sagen, dass versucht wird weitest möglich den Micro Kernel Ansatz zu verfolgen, und nur dort wo es signifikante Performancegewinne bringt zum Monolithischen Modell zu wechseln. So sind beim Hybriden Ansatz oft bestimmte Treiber mit in den Kernel gebunden. Ein Beispiel dafür ist die DARWIN Distribution, welche eigentlich auf einem MACH Micro Kernel basiert, jedoch aus Leistungsgründen einige Treiber in den Kernel bindet [Wikb]. Betriebssysteme, die einen Hybrid Kernel verwenden [Wikc]:

[4] Siehe: `http://l4linux.org/` (18.07.2010)

- Mac OS X
- BeOS
- Windows NT – Windows 7
- ReactOS

2.2.4 Kernelspace vs. Userspace

In den vorangegangenen Punkten wurden mehrfach die Begriffe Kernelspace und Userspace verwendet und teilweise kurz erläutert. Da diese Begriffe für das Verständnis von Systemcalls in Kapitel 3 zentral sind, sollen sie hier kurz erklärt werden.

Der Kernel und die darin arbeitenden Prozesse bedürfen eines besonderen Schutzes, um nicht von Anwendungsprogrammen in ihrer Arbeit gestört zu werden. Geschähe eine solche Störung, so ist nicht unwahrscheinlich, dass das gesamte Systemabstürzt. Um also eine Trennung zwischen den Programmteilen, die den Kernel ausmachen (somit für den generellen Betrieb des Systems zuständig sind) und den Anwendungsprogrammen zu erreichen, wurden Kernelspace und Userspace eingeführt (siehe auch Abbildung 2.2). Diese Trennung bezieht sich auf den Arbeitsspeicher und ist hierarchisch organisiert. Ein Anwendungsprogramm läuft im Userspace und es ist ihm nicht möglich Speicher im Adressraum des Kernelspace zu adressieren. Umgekehrt ist es jedoch möglich, dass der Kernel oder Systemteile die im Kernelspace laufen, Speicher im Userspace adressieren [Mau04, S.7f.].

In der Regel wird in Betriebssystemen nur zwischen Kernelspace und Userspace unterschieden, diese Unterscheidung wird durch Privilegienstufen der CPU realisiert. Moderne CPU unterstützen unterschiedliche Modi mit unterschiedlichen Berechtigungen in denen sich ein Prozess befinden kann [Mau04, S.7f.]. Die IA-32-Architektur, die unteranderem vom Intel®Pentium, Celeron und Core implementiert wird, unterstützt vier hierarchisch organisierte Modi [Int10, Kapitel 3]. Wie diese im Detail auf CPU-Ebene realisiert werden und vom Kernel aufgerufen werden, wird in dieser Arbeit nicht weiter besprochen. [5]

[5] Für weitere Informationen zu diesem Thema mit dem Fokus auf die IA-32-Architektur sei das *Intel®64 and IA-32 Architectures Software Developer's Manual, Volume 3A, Chapter 3* genannt

3 Systemcalls

Der letzte Abschnitt beschäftigt sich mit dem Thema Kernelspace und Userspace. Aus dem dort gesagten ergibt sich die Frage, wie Anwendungsprogramme, die natürlicher Weise im Userspace laufen, mit dem Kernel kommunizieren können. Die Methoden des Kernels liegen im Kernelspace und lassen sich somit vom Userspace aus nicht adressieren. Wie schon besprochen stellt der Kernel jedoch viele Dienste bereit, die für Anwendungsprogramme nutzbar sind (Datei öffnen, Speicher allokieren, in den Standard Output schreiben, usw.). Diese Dienste sind Systemcalls. Systemcalls sind die Brücke zwischen Kernelspace und Userspace [Mau04, S.557][Lov07, S.3].

Als normaler Anwendungsentwickler wird man in seltensten Fällen einen Systemcall direkt ansprechen. Dieses ist Aufgabe der Systemprogrammierer. Jeder Entwickler verwendet jedoch Systemcalls, ohne dass er sich dessen notwendiger Weise bewusst ist, weil die Systemcalls durch die Standardbibliotheken der jeweiligen Programmiersprache gekapselt werden[Lov07, S.4]. Das einfachste Programm, das *Hello World* Programm, ist ein Beispiel für diese Kapselung. Im Folgenden ist das *Hello World* Programm in unterschiedlichen Sprachen dargestellt:

Listing 3.1: Sprache: JAVA

```
1   System.out.print("Hello World");
```

Listing 3.2: Sprache: C

```
1   printf("Hello World");
```

Listing 3.3: C (über das Syscall-Interface)

```
1   char buf[] = "Hello World";
2   syscall(4, 1, buf, 11);
```

Listing 3.4: ASM Assembler (32bit, x86 Architektur)

```
1   section .data
2   s: db 'Hello world',10
3   sLen: equ $-s
4
5   _start:
6   mov eax, 4  ; Nr. des syscall
7   mov ebx, 1  ; Nr. des Outputs (1 = stdout)
8   mov ecx, s  ; der String (der Pointer zu dem String)
```

```
9     mov edx, sLen  ; Laenge des Strings
10    int 80h  ; loest den Interrupt aus
```

In allen Varianten passiert im Wesentlichen das Gleiche. Der Systemcall `write` wird aufgerufen und *"Hello World"* wird in den Standard Output geschrieben. Die ersten beiden Varianten, JAVA und C sind die gewohnten, hierbei werden Methoden der Standardbibliotheken aufgerufen. Die dritte Variante greift auf ein Systemcall-Inteface der Standardbibliothek für Systemcalls mit drei Parametern zu, und liegt *auf halbem Weg* der Abstraktion. Als letztes die vierte Variante: Der in Assembler geschriebene Aufruf des Systemcalls. Dieser geht direkt an den Kernel ohne eine Schicht dazwischen. Es sei erwähnt, dass der Assemblercode, der in der Standardbibliothek gclib für die x86 Architektur verwendet wird, syntaktisch anders aussieht. Der Grund dafür ist, dass dort der Assemblercode in C Quellcodedateien geschrieben wird, was etwas umständlich anmutet, da die ASM Umgebung von C verwendet werden muss (die ASM Umgebung wird mit: `__asm__` eingeleitet) [Mau04, S.585f.]. Inhaltlich sind beide Varianten identisch, die Variante ohne die ASM Umgebung ist deutlich leichter zu lesen und wurde nur deshalb ausgewählt.

An dem oben gegebenen Beispiel soll die Funktionsweise des Aufrufs genauer erläutert werden. Zunächst wird die mittlere Abstraktionsschicht, die Systemcall-Interface Variante, erläutert. Die gclib Standardbibliothek beinhaltet für Systemcalls mit 0 bis 6 Parametern jeweils eine Funktion `syscall` mit der entsprechenden Anzahl + 1 an Parametern. Betrachtet man nun den obigen Aufruf

```
syscall(4, 1, buf, 11);
```

so ist der erste Parameter die Nummer des Systemcalls der aufgerufen wird. Diese Nummer wird in der Datei `unistd.h` bzw. `unistd_32.h` der Konstanten `__NR_write` zugewiesen und ist die Nummer des Systemcalls `write`.[1] Die vergebenen Nummern sind plattformabhängig.[2]

Der zweite Parameter ist der erste Parameter des Systemcalls `write` und bestimmt den Output in den geschrieben werden soll. [3]

Der dritte Parameter `buf` ist der String der ausgegeben werden soll (genauer gesagt der Pointer auf den String).

Der vierte Parameter ist die Anzahl der Bytes, die ausgegeben werden soll, in diesem Fall 11. Käme in dem String ein Zeichen vor, das mehr als ein Byte benötigt wie in *"Hello*

[1] Zu finden, auf Ubuntu Linux unter: /usr/src/linux-headers-2.6.xx-xx/arch/x86/include/asm

[2] Alle Angaben die in dieser Arbeit gemacht werden, soweit nicht anders vermerkt beziehen auf die x86 Architektur (32 bit). Distributionsabhängige Angaben beziehen sich auf Ubuntu Linux 10.04

[3] Es gibt drei mögliche Ausgaben:

0 = Standard Input

1 = Standard Output

2 = Standard Error

wörld", so wäre die Anzahl höher, in diesem Fall 12. Der Rückgabewert dieses Aufrufes ist die Anzahl der geschriebenen Bytes. Ist die Zahl negativ so kam es bei der Ausführungzu einem Fehler [Wika].

Betrachtet man im Vergleich die ASM Implementation, werden die Parallelen deutlich [Swa02, Kapitel 4.3]. Die Parameter, die eben genannt wurden gibt es auch hier, jedoch wird zusätzlich angegeben was mit ihnen geschieht. `mov eax, 4 ;` (Listing 3.4, Zeile 6) bedeutet, dass in den Register `eax` der Wert 4 eingetragen wird. Die Register `eax, ebx, ecx, edx` sind Register der x86 Architektur.

In den drei auf `mov eax, 4 ;` (Listing 3.4, Zeile 7-9)folgenden Zeilen werden die Parameter in jeweils ein Register geschrieben. Entscheidend ist die Zeile `int 80h` (Listing 3.4, Zeile 10). Mit diesem Aufruf wird ein Interrupt ausgelöst. Der Kernel übernimmt ab jetzt die Kontrolle und führt den angegebenen Systemcall aus. Das Ergebnis wird entsprechend der Angabe in den Standard Input, Standard Output oder Standard Error geschrieben.

Bei Aufrufen mit bis zu sechs Parametern wird immer verfahren wie oben beschrieben. Einen Unterschied kann lediglich der Rückgabewert machen. Es kann sein dass es erforderlich ist, dass das Ergebnis des Aufrufes zurück in den Speicher des aufrufenden Prozesses geschrieben wird. Dieser Vorgang ist etwas aufwändiger als er zunächst erscheint. Man könnte vermuten, dass es ausreicht, dass der Kernel das Ergebnis berechnet, in den Speicher schreibt und dem aufrufenden Prozess den Pointer übergibt. Das ist jedoch nicht der Fall. Der Kernel arbeitet im Kernelspace und der aufrufende Prozess im Userspace. Bekäme der aufrufende Prozess lediglich den Pointer, so müsste er die Daten aus dem Kernelspace laden können, was er nicht kann. Deshalb muss der Kernel die Daten aus dem Kernelspace in den Userspace kopieren oder direkt in den Userspace schreiben [Mau04, S.576f.].

Sollte ein Systemcall mehr als sechs Parameter haben ändert sich das Verfahren, die Parameter werden nicht mehr direkt in die Register geschrieben, sondern zunächst im Speicher gehalten. Es wird eine C-Struktur verwendet und der Pointer auf diese übergeben [Mau04, S.576f.]. Auf dieses Verfahren wird in dieser Arbeit nicht weiter eingegangen.

Wie oben beschrieben stellen die Systemcalls die Schnittstelle zum Kernel dar. Im Umfeld der Betriebssysteme gibt es viele Varianten von Kernelimplementationen. Um die Anwendungssoftware zwischen diesen Systemen portierbar zu machen, gibt es einen definierten Standard für diese Schnittstelle: Den POSIX Standard (Portable Operating System Interface [for Unix]) [Lov07, S.6f.][Jos, Q0]. [4] Dieser Standard wir von wenigen Betriebsystemen komplett implementiert, jedoch wird er von vielen Betriebsystemen zu sehr großen Teilen eingehalten. So dass, das Ziel, die Portierbarkeit, in weiten Teilen gegeben ist.

[4] Die Definition des POSIX Standard IEEE Std 1003.1 ist unter: `http://www.unix.org/version3/` zu finden.

3.1 `ptrace` – Ein Systemcall im Speziellen

Das Thema dieser Arbeit ist STRACE. Deshalb wurden viele Grundlagen zum Verständnis erläutert. Nun wird die letzte Grundlage geliefertnämlich `ptrace`, der Systemcall, der die Basis von STRACE darstellt. Dieser Systemcall ist sehr mächtig und ist nicht nur die Basis von STRACE, sondern auch die Basis von ganz anderen Programmen, wie z.B. dem Debugger GDB. `ptrace` wird hauptsächlich aus der Perspektive von STRACE beschrieben werden [Mau04, S.577ff.].

`ptrace` hat vier Parameter. Stellt man ihn via Pseodocode dar, ist dieses eine verständliche Variante:

```
ptrace(opt, pid, addr, data)
```

Dominiert wird ptrace von dem ersten Parameter `opt`. Dieser gibt an welche Operation ausgeführt wird. Abhängig davon werden die anderen Parameter gefüllt oder leer gelassen. Im Folgenden werden einige der Operation vorgestellt. Eine vollständige Darstellung findet sich auf den meisten LINUX Systemen mit dem Shell-Aufruf `man ptrace`.

3.1.1 Tracing Operation

`PTRACE_ATTACH`: Wird dieser Befehl und eine ProzessID übergeben, so ist dies die Aufforderung zum Anhängen an den Prozess, was die Verfolgung aktiviert. Bei diesem Vorgang wird der Prozess, der `ptrace` aufgerufen hat, bei dem zu verfolgenden Prozess als Elternprozess eingetragen und ein `STOP` Signal gesendet. Der verfolgte Prozess wir nun immer vor dem Senden eines Signals an ihn gestoppt. Der Verfolgende, also Elternprozess, wird mittels des Signals `SIGCHLD` informiert und kann eingreifen. Das `SIGCHLD`-Signal kann mittels der `wait` Funktion erwartet werden [Mau04, S.581].

`PTRACE_DETACH`: Mit dieser Operation wird die Verfolgung wieder beendet.

`PTRACE_CONT`: Setzt die Verfolgung bis zum nächsten Signal fort.

`PTRACE_SYSCALL`: Diese Operation funktioniert im Prinzip genauso wie `PTRACE_CONT`, jedoch wird der verfolgte Prozess nicht vor dem Senden eines Signals an ihn unterbrochen, sondern vor und nach dem Ausführen eines Systemcalls. Auf dieser Funktion basiert STRACE wesentlich.

`PTRACE_SINGLESTEP`: Wie mit `PTRACE_CONT` und `PTRACE_SYSCALL` wird ein Prozess verfolgt, unterbrochen wird bei jeder Assembleranweisung. Diese Funktion ist die Basis von dem Debugger GDB.

3.1.2 Examinierende Operationen

`PEEKUSER, PEEKTEXT, PEEKDATA`: Mit dieser Operation lassen sich die Werte der Register der CPU auslesen. Die Register werden mittels einer Kennzahl, z.B. der

Konstanten ORIG_EAX, selektiert. Mit PEEKTEXT und PEEKDATA lassen sich Daten aus dem Adressraum des verfolgten Prozesses lesen, sowohl aus dem Text- als auch aus dem Datenbereich.[5]

3.1.3 Injektive Operationen

POKEUSER, POKETEXT, POKEDATA: Mit diesen Operationen lassen sich Daten in die Bereiche: Register, Textbereich, Datenbereich schreiben. Diese Funktion ist zentral beim interaktiven Debugging, da sich gezielt der Adressraum eines Prozesses manipulieren lässt. Allerdings lässt sich mit dieser Operation noch noch vieles mehr machen. Jegliche Art von Codeinjektion ist möglich.[6]

Wie schon oben erwähnt sind die hier genannten, nicht alle mit ptrace möglichen Operationen, jedoch geben sie eine Vorstellung, welche Art von Operation möglich sind. Daraus ergibt sich auch in welchen Gebieten ptrace eingesetzt wird.

1. Tracingtools wie STRACE: Diese Tools werden jeweils eine der Tracing Operationen verwenden und anschließend eine oder mehrere der examinierenden Funktionen, um Informationen über den Prozess zur Verfügung zu stellen.

2. Debugger in unterschiedlichen Variationen: Breakpoint, Singlestep, Interaktiv. Für Breakpoint- und Singlestep-Debugging sind in erster Linie die Tracingfunktionen (besonders PTRACE_SINGLESTEP) und die examinierenden Operationen relevant. Während für das interaktive Debugging auch die injizierenden Operationen interessant sind.

3. Unterschiedliche Formen der Codeinjektion: ptrace eignet sich mit den POKE Operationen gut, um sich in Programme einzuhaken. Vorstellbar ist z.B. dass man ein Programm welches nur kompiliert vorliegt mit dieser Methode erweitert. Angemerkt sei dazu, dass diese Variante extrem aufwändig und plattformabhängig ist.

3.2 SSTRACE

Nun sind alle wesentlichen Bestandteile von STRACE einzeln erklärt. Was zum Verständnis der Arbeitsweise von STRACE noch fehlt, ist die richtige Zusammensetzung der Einzelteile. Dieses soll mit Hilfe der exemplarischen Implementierung eines Tracing Pro-

[5] Zum Zeitpunkt der Unterbrechung ist selbstverständlich ptrace selber der aktive Prozess. Das bedeutet, dass seine Werte in den Registern stehen. Da diese Werte nicht interessant sind, werden die Werte verwendet, die beim nächsten Taskswitch in die Register kopiert werden. Somit kommt das Auslesen, sowie die Manipulation dieser Daten, der Manipulation der realen Register - der Erwartung entsprechend - gleich [Mau04, S.578].

[6] Alle Angaben zu Parametern von ptrace stammen aus: LINUX Programmers Manual 30.3.2009 (http://www.kernel.org/doc/man-pages)

grammes, des Programmes SSTRACE (simple strace)[7], geschehen.[8] Das Programm SSTRACE besteht im Wesentlichen aus drei Teilen:

1. Dem Kernel muss mitgeteilt werden, welcher Prozess verfolgt werden soll.
2. Das Programm muss auf die Unterbrechung (das Senden von `SIGCHLD`) reagieren. Anschließend muss dem Kernel mitgeteilt werden, wenn der Prozess weiterverfolgt werden soll und bei welchem Ereignis als nächstes unterbrochen werden soll.
3. Die gewünschten Informationen müssen ausgelesen werden.

Dieses sind die Kernaufgaben. STRACE macht natürlich noch um einiges mehr, insbesondere was die Bereiche Aggregation, Formatierung und Filterung der Informationen anbelangt.

[7] Ein besserer Name wäre VVSSTRACE (very very simple strace)
[8] Die größten Teile dieses Programmes stammen aus: [Mau04, S.580f.]

Listing 3.5: simple strace: sstrace

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<sys/ptrace.h>
4  #include<sys/wait.h>
5  #include<sys/reg.h>
6  #include<string.h>
7
8  static long pid;
9
10 // sucht syscall id aus dem Userspace des child prozesses
11 int peekuser(int pid, long off, long *res){
12   long val;
13   val = ptrace(PTRACE_PEEKUSER ,pid ,off ,0);
14
15   if(val == -1){
16     return -1;
17   }
18
19   *res = val;
20   return 0;
21  }
22
23 // traced den naechsten syscall
24 void trace_syscall(){
25   long res;
26   res = ptrace(PTRACE_SYSCALL, pid, (char*) 1, 0);
27
28   if(res < 0){
29     printf("Failed to execute trace for next syscall: %ld\n", res);
30   }
31 }
32
33 // Handler methode die mit den Signalen vom Child-Prozess aufgerufen
     wird (Listener)
34 void sigchld_handler(int signum){
35   long scno;
36   int res;
37 // der offset fr amd64 waere sind ORIG_RAX * 8
38   if(peekuser(pid, 4 * ORIG_EAX, &scno) < 0 ){
39     printf("No peekuser\n");
40     return;
41   }
42   if(scno != 0){
43     printf("System call nr: %ld\n", scno);
```

```
44   }
45
46 // den naechsten syscall tracen
47   trace_syscall();
48 }
49
50 int main(int argc, char** argv){
51   int res;
52   if (argc != 2){
53     printf("Usage: ptrace <pid>\n");
54     exit(-1);
55   }
56
57   pid = strtol(argv[1], NULL, 10);
58
59   if(pid <= 0){
60     printf("No valid pid specified\n");
61     exit(-1);
62   } else {
63     printf("Tracing request for pid %ld\n", pid);
64
65   }
66
67 // Die Signale vom CHILD Prozess abfangen. (Einen Listener einhaengen)
68   struct sigaction sigact;
69   sigact.sa_handler = sigchld_handler;
70   sigaction(SIGCHLD, &sigact, NULL);
71 // den Prozess <PID> zum Child Prozess machen
72   res = ptrace(PTRACE_ATTACH, pid, 0, 0);
73   if(res < 0){
74     printf("Faild to attach: %d\n", res);
75     exit(-1);
76   } else {
77     printf("Attached to %ld\n", pid);
78   }
79
80 // wait loop
81   for(;;){
82     wait(&res);
83     if(res == 0){
84       exit(1);
85     }
86   }
87 }
```

Der erste oben erwähnte Teil ist leicht implementiert. Es ist nur ein `ptrace` Aufruf nötig. Der Prozess wird angegeben, und von `ptrace` zu einem Childprozess des verfolgenden Prozesses gemacht. Dabei wird ein `STOP` Signal gesendet.

`ptrace(PTRACE_ATTACH,pid,0,0);` (Listing 3.5 Zeile 72)

Der zweite Teil, das jeweilige erneuern des Verfolgens ist etwas aufwändiger. Dieser Teil des Programmes muss immer dann ausgeführt werden, wenn der verfolgte Prozess ein `SIGCHLD`-Signal sendet [Lov07, S.139f.]. Dieses geschieht mittels des Codes zwischen Zeile 68 und 71 aus Listing 3.5. Das eigentliche *Horchen* auf das Signal wird mit der Methode `sigaction gestartet`. Diese Methode bekommt den Pointer auf eine C-Struktur, welche wiederum eine Handlermethode enthält: `sigchld_handler` (Listing 3.5 Zeile 34ff.). Am Ende dieser Methode wird das Verfolgen bis zum nächsten Systemcall mittels der Methode `trace_syscall()` (Listing 3.5 Zeile 24ff.) initiiert. In der Methode `trace_syscall()` wird lediglich `ptrace` wieder aufgerufen, wiederum mit der ProzessID, diesmal jedoch mit dem Parameter `PTRACE_SYSCALL`.

Der dritte Teil, das Auslesen der Daten, findet in der Handlermethode `sigchld_handler` statt, wiederum mittels ptrace. Mit dem Parameter `PTRACE_PEEKUSER` und der Angabe des Registers `ORIG_EAX` wird die Nummer des aktuellen Systemcall ausgelesen.[9] Man könnte dieses Programm nun noch damit anreichern den Namen des Systemcalls auszugeben und mit anderen Varianten von `ptrace`, die Parameter auszulesen. Das zugrunde liegende Prinzip bliebe jedoch dasselbe.

[9] Die Nummer aus der Datei `unistd.h` bzw. `unistd_32.h`, vgl. auch Kapitel 3

4 STRACE verwenden

Bisher wurde die Funktionsweise und der interne Aufbau von STRACE sowie die dafür notwendigen Grundlagen behandelt. Im Folgenden wird die Anwendung von STRACE im Mittelpunkt stehen. Um noch einmal das Einsatzgebiete von STRACE deutlich zu machen, sei an dieser Stelle die Manual Seite von STRACE zitiert:

> STRACE is a useful diagnostic, instructional, and debugging tool. System administrators, diagnosticians and trouble-shooters will find it invaluable for solving problems with programs for which the source is not readily available since they do not need to be recompiled in order to trace them.[1]

Die Möglichkeiten der Verwendung zu diagnostischen Zwecken und als Hilfsmittel zur Fehlersuche, gehen in einander über, deshalb wird im Folgenden keine scharfe Trennung vorgenommen [Wikf]. Stattdessen werden zunächst einfache Beispiele dargestellt, die im Fortgang um einige Parameter ergänzt werden, um so die wesentlichen Funktionen zu erläutern. STRACE kann grundsätzlich in zwei Varianten verwendet werden. In der einen startet STRACE das Programm das verfolgt werden soll, in der anderen hängt sich STRACE an ein laufendes Programm an. Die erste Variante hat den Vorteil, dass Teile der Umgebung (Umgebungsvariablen etc.) leicht mitgegeben und dadurch leicht konfiguriert und variiert werden können. Die Vorteile der zweiten Variante sind deutlicher. Es können auf diese Weise sehr gut Dienste verfolgt werden ohne den Betrieb zu stören [Wei06].

4.1 Zwei Beispiele

Der Befehl

```
strace cat  /.profile > /tmp/profile
```

verfolgt das Programm cat über seine ganze Ausführungszeit. Der Output von STRACE erscheint über die Standard Error Ausgabe auf der Konsole. Für das Öffnen der Datei erscheint diese Ausgabe:

```
open("/home/lars/.profile", O_RDONLY) = 3
```

Der ausgeführte Systemcall, in Klammern dahinter die Parameter als letztes, hinter dem Gleichheitszeichen der Rückgabewert. Würde man den Pfad einer nicht existenten Da-

[1] Manual Seite zu STRACE Stand: 21.1.2001

tei (z.B. `.profileXXX`) mit dem oben genannten Befehl aufrufen, ergibt sich folgende STRACE Ausgabe:

```
open("/home/lars/.profileXXX", O_RDONLY) = -1 ENOENT
                (No such file or directory)
```

Mit dem Parameter `-p` gefolgt von der ProzessID lässt sich STRACE in der zweiten Variante, der Verfolgung eines schon laufenden Programmes, starten. Ein gutes Beispiel ist die Verfolgung des Programmes top. Auf einer Konsole wird nun zunächst TOP gestartet. Auf einer zweiten Konsole wird dann STRACE gestartet:

`strace -p 5570` (unter der Annahme das 5570 die ProzessID von TOP ist)

Alle drei Sekunden (Taktung von TOP) ergießt sich nun ein Schwall von Ausgaben auf die Konsole, die der oben genannten Lesart folgen. Die bisher genannten Aufrufe von STRACE sind die einfachsten und am wenigsten gefilterten. Schon bei der Betrachtung der Ausgabe, der Verfolgung von TOP, wird deutlich, dass mit dieser Ausgabe ohne weitere Eingrenzungen nicht sehr hilfreich ist. Ein erste Schritt ist das Umleiten der Ausgabe in eine Datei. Das kann entweder mit

```
strace -p 5570 > /tmp/strace_out
```

erfolgen oder - und diese Variante ist die bessere - mit dem Parameter `-o` gefolgt von dem Dateipfad. Der Vorteil ist, dass sich beim Aufruf von

```
strace -o /tmp/strace_out top
```

die Ausgabe von TOP und von STRACE nicht vermischen.

Wird STRACE ernsthaft eingesetzt um Fehler zu suchen, so liegt in vielen Fällen eine Vermutung vor, z.B. dass eine Datei nicht verfügbar ist, Berechtigungen nicht gesetzt sind etc. Zu diesem Zweck lässt sich STRACE sehr genau konfigurieren. Es gibt einige vorgefertigte Sets an Systemcalls die man auswählen kann. So werden mit `strace -e file` (Kurzform von `strace -e set=file`) nur Systemcalls verfolgt, die mit Dateioperationen im Zusammenhang stehen (wie: `access, open, write, stat`).[2] Andere Sets die zur Verfügung stehen sind: `network, signal,process, ipc`. Es lassen sich aber auch einzelne Systemcalls festlegen und verfolgen. Z.B. würde

```
strace -e trace=read,open
```

nur die Systemcall `read` und `open` verfolgen. Mit diesen Filterungsmechanismen lässt sich, in den meisten Fällen die Ausgabe ausreichend einschränken. Für weitere Filterungen lassen sich anschließend, Tools zu Log-Analyse einsetzen. Auch grep kann mit

[2] Für einen Ausschnitt der Ausgabe des Befehls `strace -e file -o /tmp/strace_out top` siehe Appendix 1

seiner Unterstützung für Reguläre-Ausdrücke sehr Hilfreich sein. Ein Beispiel wäre dass mit STRACE nur der Systemcall `open` verfolgt wird (`strace -e trace=open -o /tmp/strace_out`)[3]. Anschließend kann man sich mit `cat /tmp/strace_out | grep /home` die Systemcalls herausfiltern, die auf das `/home/*` Verzeichnis zugegriffen haben.

STRACE kann aber noch anderes ausgeben als die Systemcalls, die aufgerufen wurden. Gerade bei Dateioperationen kann es interessant sein, was in den geöffneten Dateien enthalten ist. Mit dem eben schon erwähnten Parameter `-e` lässt sich in die geöffneten Dateien hinein schauen. Mit dem Befehl `strace -e trace=read -e read=all` wird durch den Teil `-e trace=read` nur der Systemcall `read` verfolgt, und durch den Teil `-e read=all` wird angegeben, dass der Inhalt der Datei mit ausgegeben werden soll. Diese kann extrem hilfreich sein, um zusehen mit welchen Eingaben das verfolgte Programm umgeht.

STRACE verfügt noch über viele weitere Parameter. Mit `-T` wird die Zeit, die ein Systemcall gebraucht hat mit ausgegeben, mit `-t` die Uhrzeit, mit `-f` werden die neu erzeugten Childprozesse des verfolgten Prozesses mit verfolgt etc. Viele von diesen Parametern sind nur für sehr spezielle Anwendungsfälle interessant und es wird nicht weiter auf sie eingegangen. Es gibt jedoch noch eine Variante, die besonders für diagnostische Zwecke bzw. die Selbstkontrolle des Softwareentwicklers sehr interessant ist. Mit dem Parameter `-c` legt STRACE eine Statistik über die ausgeführten Systemcalls an: welcher wie oft verwendet wurde, welcher in der Summe wie viel Zeit gebraucht hat. Diese Statistik kann sehr viel über ein Programm aussagen und eventuell einige Optimierungen nahe legen. Die Verwendung von `-c` und den Filterungsparametern kann, zum Beispiel, Sinn machen um zu überprüfen ob alle Dateien, die geöffnet wurden auch wieder geschlossen wurden:[4]

```
strace -e trace=open,close -c
```[5]

[3] Für einen Ausschnitt der Ausgabe des Befehls `strace -e trace=open -o /tmp/strace_out top` siehe Appendix 2

[4] Alle Angaben zu Parametern von STRACE sind der Manual Seite zu STRACE entnommen Manual (21.1.2001)

[5] Für die Ausgabe des Befehls `strace -c -e trace=open,close -o /tmp/strace_out top` siehe Appendix 3

5 Fazit

Mit STRACE liegt ein Programm vor, dass an der letzten Schicht vor dem Kernel monitoren kann. Viel tiefer braucht ein Softwareentwickler selten zu schauen. Die Ausgaben die STRACE liefert, zeigen eine andere Perspektive auf die Software, die geschrieben wird. Dieser Perspektivwechsel kann sehr aufschlussreich und vor allem hilfreich bei der Entwicklung von hochwertiger Software sein. Weiterhin kann STRACE ein hilfreiches Werkzeug sein, falls Programme, die nur in kompiliertet Form vorliegen kryptische Fehlermeldungen werfen.

Um dieses Programm sinnvoll einsetzen zu können, ist ein Verständnis für die Arbeitsweise von Systemcalls und damit auch einige Grundlagen der Kernelarchitektur nötig. Das Ziel dieser Arbeit war, diese Grundlagen in einem überschaubaren Rahmen aufzuzeigen und in die Möglichkeiten die STRACE bietet einzuführen.

Appendix

1 STRACE-Ausgaben

Listing 1: Teil der Ausgabe des Befehls:
`strace -e file -o /tmp/strace_out top`

```
1  execve("/usr/bin/top", ["top"], [/* 39 vars */]) = 0
2  access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or
       directory)
3  access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or
       directory)
4  open("/etc/ld.so.cache", O_RDONLY) = 3
5  access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or
       directory)
6  open("/lib/libproc-3.2.8.so", O_RDONLY) = 3
7  access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or
       directory)
8  open("/lib/libncurses.so.5", O_RDONLY) = 3
9  access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or
       directory)
10 open("/lib/libc.so.6", O_RDONLY) = 3
```

Listing 2: Teil der Ausgabe des Befehls:
`strace -e trace=open -o /tmp/strace_out top`

```
1  open("/etc/ld.so.cache", O_RDONLY) = 3
2  open("/lib/libproc-3.2.8.so", O_RDONLY) = 3
3  open("/lib/libncurses.so.5", O_RDONLY) = 3
4  open("/lib/libc.so.6", O_RDONLY) = 3
5  open("/lib/libdl.so.2", O_RDONLY) = 3
6  open("/proc/version", O_RDONLY) = 3
7  open("/proc/stat", O_RDONLY|O_CLOEXEC) = 3
8  open("/proc/sys/kernel/pid_max", O_RDONLY) = 3
9  open("/etc/toprc", O_RDONLY) = -1 ENOENT (No such file or directory)
10 open("/home/<username>/.toprc", O_RDONLY) = -1 ENOENT (No such file
      or directory)
11 open("/lib/terminfo/x/xterm", O_RDONLY) = 3
12 open("/proc", O_RDONLY|O_NONBLOCK|O_DIRECTORY|O_CLOEXEC) = 3
```

Listing 3: Ausgabe des Befehls:
`strace -c -e trace=open,close -o /tmp/strace_out top`

```
1  % time seconds usecs/call calls errors syscall
2  ------ ----------- ----------- --------- --------- ----------------
3   85.93 0.000391 0 809 close
4   14.07 0.000064 0 813 2 open
5  ------ ----------- ----------- --------- --------- ----------------
6  100.00 0.000455 1622 2 total
```

Literaturverzeichnis

[Han] Fred Hantelmann. Mechanismen und Aufbau des Linux-Kernels - Innenansichten. http://research.intelego.net/heise/ix/1999/03/120/art.htm. Abgerufen: 15.5.2010.

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of $\mu - Kernel - basedSystems$. $InOperatingSystemsPrinciples(SOSP - 97), pages66 - -77, NewYork$, 1997.$ACMPress$.

[Int10] Intel®. *Intel®64 and IA-32 Architectures Software Developer's Manual, Volume 3A*. Intel®, 2010.

[Jos] Andrew Josey. POSIX®1003.1 Frequently Asked Questions. http://www.opengroup.org/austin/papers/posix_faq.html. Abgerufen: 23.09.2010.

[Lov07] Robert Love. *Linux system programming: system and library calls every programmer needs to know*. O'Reilly & Associates, Inc., pub-ORA:adr, 2007.

[Mau04] Wolfgang Mauerer. *LINUX Kernelarchitektur*. Carl Hanser Verlag, 2004.

[Swa02] Derick Swanepoel. Roy Thomas Fielding. http://www.cin.ufpe.br/~if817/arquivos/asmtut/index.html, 2002. Abgerufen: 14.5.2010.

[Wei06] Hendrik Weimer. strace - Dissecting Programs. http://www.osreviews.net/reviews/admin/strace, 2006. Abgerufen: 20.9.2010.

[Wika] Code Wiki. write (C System Call). http://codewiki.wikidot.com/c:system-calls:write. Abgerufen: 25.09.2010.

[Wikb] Wikipedia. Darwin (operating system). http://en.wikipedia.org/wiki/Darwin_(operating_system). Abgerufen: 24.9.2010.

[Wikc] Wikipedia. Hybridkernel. http://de.wikipedia.org/wiki/Hybridkernel. Abgerufen: 17.6.2010.

[Wikd] Wikipedia. Monolithic kernel. http://en.wikipedia.org/wiki/Monolithic_kernel. Abgerufen: 15.5.2010.

[Wike] Wikipedia. Monolithic Kernel. `http://en.wikipedia.org/wiki/Microkernel`. Abgerufen: 15.5.2010.

[Wikf] Wikipedia. strace. `http://en.wikipedia.org/wiki/Strace`. Abgerufen: 21.9.2010.

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den _____ Unterschrift: _____