Studienarbeit

# Reverse Engineering: Rechtliche Rahmenbedingungen und praktische Möglichkeiten

André Janz
E-Mail: 5janz@informatik.uni-hamburg.de

Oktober 2000

Universität Hamburg
Fachbereich Informatik
Betreuer: Prof. Dr. K. Brunnstein

# Inhaltsverzeichnis

# 1. Einführung

In der Vergangenheit wurde Reverse Engineering für die verschiedensten Zwecke eingesetzt. Dabei wurde anfangs meist ad hoc vorgegangen, und rechtliche Beschränkungen wurden entweder nicht berücksichtigt oder waren noch nicht vorhanden. Es gab zwar schon seit den 60er Jahren einige wissenschaftliche Ansätze, diese bezogen sich aber meist auf spezielle Maschinenarchitekturen und wurden durch die fehlende Portabilität schnell obsolet. Zu den Anwendungen gehörte hauptsächlich die Übertragung von bestehendem Code auf neue Maschinen [4, 11], was z. B. wegen fehlendem Quellcode oder nicht mehr verfügbaren Programmiersprachen nötig wurde. Es wurde aber auch versucht, durch eine derartige Analyse optimierende Compiler besser zu verstehen [18].

Heute existieren in den meisten Ländern gesetzliche Regelungen über die Zulässigkeit von Reverse Engineering, und auch die Einsatzgebiete haben sich zum Teil gewandelt. Bei der Übertragung auf neue Maschinenarchitekturen neigt man inzwischen eher dazu, die Software neu zu entwickeln, wenn kein Quellcode mehr vorhanden ist, da der Aufwand dabei kaum größer oder sogar geringer ist; außerdem ist die Qualität und Performanz der Software bei einer Neuentwicklung im allgemeinen deutlich besser.

Der Schwerpunkt der Anwendung liegt inzwischen eher bei der nachträglichen Dokumentation und der punktuellen Fehlerbeseitigung, wenn der Quellcode nicht mehr vorhanden ist (Y2K). Aber auch wenn letzterer vorliegt, ist Reverse Engineering manchmal unverzichtbar, z. B. muß bei der Compilerentwicklung der bei der Übersetzung von Testprogrammen erzeugte Maschinencode mit einem Disassembler überprüft werden, um die (zumindest annähernde) Korrektheit des Compilers zu gewährleisten. Eine Disassemblierung findet auch bei der Entwicklung einer Optimierungsstufe für einen Compiler statt, um die Effizienz des erzeugten Codes zu überprüfen, wobei hier letztendlich aber Benchmarkergebnisse ausschlaggebend sind.

Für einige Zwecke wird auch das Reverse Engineering von Fremd-Software betrieben, d. h. Software, an der man keine Urheberrechte besitzt und über deren Quellcode man nicht verfügt, der aber an anderer Stelle existiert. Hierbei geht es meistens darum, die der Software zugrundeliegenden Ideen, Grundsätze und Schnittstellen herauszufinden, um sie in eigener Software einzusetzen. Aufgrund der fehlenden Rechte treten hier naturgemäß die meisten rechtlichen Probleme auf; dieser Bereich wird in Kapitel 3 behandelt werden. Dieser Einsatzzweck ist aber trotzdem sehr wichtig, im Bereich der Schnittstellen für die Herstellung interoperabler Software (wenn diese nicht möglich ist, kann es sehr leicht zu einem Monopol kommen) und im Bereich der

sonstigen Grundsätze für die Übernahme von Algorithmen und Programmiertricks oder zumindest zur Anregung und Weiterbildung.

Ein weiterer wichtiger Punkt ist der Einsatz zu Sicherheitszwecken, also die Analyse von bösartiger Software („Malware“) oder die Überprüfung von sicherheitsrelevantem Code. Dies kann auch bei vorhandenem Quellcode eine Rolle spielen, wenn derart hohe Sicherheitsanforderungen gestellt werden, daß man selbst dem Compiler nicht vertrauen kann, sondern die Äquivalenz des erzeugten Objektcodes überprüfen muß [33].

Im folgenden soll die momentane Situation des Reverse Engineerings dargestellt werden. Dabei wird der praktische Bereich hauptsächlich die Analyse von binär vorliegenden Programmen behandeln. Ausgeschlossen wird hingegen der Softwaretechnik-bezogene Aspekt des Reengineerings, d. h. die Analyse von Quellcode zum Zwecke einer Neuimplementation. Der rechtliche Teil beschränkt sich größtenteils auf die aktuelle Rechtslage; im Falle Deutschlands und der Europäischen Union auf die Neuregelung 1992/1993.

# 2. Grundlagen

## 2.1. Was ist Reverse Engineering?

Generell ist Reverse Engineering der Analysevorgang an einem beliebigen technischen Gegenstand mit dem Ziel, seine Funktionsweise zu verstehen und gegebenenfalls seine Konstruktion nachempfinden zu können. Im hier behandelten Bereich der Software versteht man darunter die Analyse eines Programms oder eines Programmfragments, um seine Funktionsweise zu verstehen und eine Verhaltensbeschreibung auf einer höheren Ebene zu erhalten, sei diese nun umgangssprachlich, in einer Programmiersprache oder in einer Entwurfsdarstellung.

Hier soll der Aspekt des Reverse Engineerings behandelt werden, der von einem binär vorliegendem Programm ausgeht und später eine umgangssprachliche Verhaltensbeschreibung oder eine Darstellung als Assemblerprogramm oder in einer höheren Programmiersprache liefert. Die Verwendung des Begriffes in der Softwaretechnik, wo unter Reverse Engineering die Analyse eines existierenden Quellcodes mit dem Ziel einer Entwurfsmodellierung und einer Neuimplementation („Reengineering“) verstanden wird, bleibt hier außen vor.

Es gibt verschiedene Möglichkeiten, ein Reverse Engineering durchzuführen; diese unterscheiden sich grundsätzlich darin, ob das Programm während seiner Laufzeit (Beobachtung, Tracing, Debugging) oder statisch (Disassemblierung, Dekompilierung) analysiert wird. Ein weiterer, vor allem aus Urheberrechtsgründen wichtiger Unterschied ist, ob der Objektcode des Programms betrachtet wird (Debugging, Disassemblierung, Dekompilierung) oder lediglich die Wirkungen beobachtet werden; letzteres kann natürlich nur zur Laufzeit geschehen. Beim Debugging ist es des weiteren möglich, den Programmablauf zu beeinflussen, indem Variablen oder der Kontrollfluß direkt geändert werden. Der eigentliche Sinn dieser Maßnahmen besteht in der Fehlersuche und -korrektur, sie lassen sich aber z. B. auch für die Analyse einer einzelnen Routine einsetzen, indem diese mehrmals mit den interessierenden Parametern ausgeführt wird.

## 2.2. Quellcode

Unter einem Quellcode wird die in einer Programmiersprache abgefaßte Ausgangsdarstellung eines Programms oder Programmfragmentes verstanden. Als Programmiersprachen können höhere Programmiersprachen oder auch maschinennahe wie Assembler dienen.

## 2.3. Assembler

Mit Assembler wird eine sehr maschinennahe Programmiersprache, die exakt auf einen Prozessortyp (oder eine Prozessorfamilie) zugeschnitten ist, sowie der zugehörige Übersetzer bezeichnet. Ein Assemblerprogramm besteht aus Pseudobefehlen, die Hinweise an den Übersetzer darstellen (z. B. Reservierung von Datenbereichen) und sogenannten Mnemonics oder Assemblerbefehlen, die im Idealfall 1:1 in Maschinenbefehle übersetzt werden; dabei hängt die Kodierung auch von den Operanden ab. Es kann vorkommen, daß es mehrere Befehle gibt, die auf den gleichen Maschinenbefehl abgebildet werden, bzw. mehrere zulässige Maschinenkodierungen für einen Befehl; diese beiden Fälle treten etwa in der Intel 80x86-Reihe auf. Ein Assemblerbefehl kann ggf. in mehrere Maschinenbefehle übersetzt werden, d. h. es existiert auf Maschinenebene keine direkte Entsprechung, so daß der Befehl durch mehrere Befehle nachgebildet werden muß; dies ist in der Sun SPARC-Architektur oft der Fall.

Eine der wichtigsten Aufgaben eines Assemblers ist die Verwaltung der Speicherbereiche und Adressen, so daß man, wie von Hochsprachen her gewohnt, symbolisch arbeiten kann und nicht bei jeder Verschiebung von Code oder Daten das ganze Programm ändern muß.

## 2.4. Höhere Programmiersprachen

Hierunter versteht man Programmiersprachen, die eine gewisse Abstraktion von der Maschinenebene erlauben und die somit nicht mehr hauptsächlich maschinen-, sondern problemorientiert sind. Sie sind daher nicht mehr direkt vom Prozessor ausführbar, sondern müssen erst durch einen Compiler oder Interpreter übersetzt werden.

## 2.5. Objektcode

Mit Objektcode wird der Maschinencode eines Programms bezeichnet, der in binärer Form vorliegt. Der Objektcode ist unabhängig von spezifischen Dateiformaten, insbesondere darf der Begriff nicht mit dem einer Objektdatei verwechselt werden. Zum Objektcode im engeren Sinne gehören daher auch keine weiteren Debug- oder Struktur- oder Bindungsinformationen, die keine Befehle an den Prozessor darstellen, sondern für das Betriebssystem bestimmt sind.

## 2.6. Compiler

Als Compiler wird ein Programm bezeichnet, das ein im Quellcode vorliegendes Programm in eine maschinennähere Darstellungsform überführt; dieser Prozeß wird Kompilierung genannt. Als Ziel einer solchen Übersetzung gibt es z. B. folgende Möglichkeiten:

**Assemblerprogramm:** Das Quellprogramm ist zwar für einen spezifischen Prozessor übersetzt worden, liegt aber noch nicht in binärer Form vor. Auch die Bezeichner und die Trennung in Code und Daten sind noch vorhanden, Typen dagegen häufig nicht mehr bzw. nur so weit, wie sie direkt vom Prozessor unterstützt werden.

**Objektdatei:** Das Quellprogramm wurde in eine binäre Form für einen spezifischen Prozessor übersetzt, ist aber nicht eigenständig lauffähig. Eine Objektdatei muß erst noch mit anderen Objektdateien zusammengebunden („Linking“) werden, um ein lauffähiges Programm zu erhalten. In Objektdateien sind üblicherweise noch viele Strukturinformationen über das Programm enthalten, die eine Dekompilierung erleichtern können.

**Binärprogramm:** Eine fertig gebundene binäre Datei, die direkt ausführbar ist. Im weiteren Sinne auch alle Systemkomponenten wie Treiber, DLLs (Dynamic Link Libraries) etc., bei denen der Begriff „Ausführung“ keinen Sinn macht, die aber trotzdem in ihrer endgültigen Form vorliegen.[1] Je nach Betriebssystem wird das endgültige Binärprogramm aber beim Aufruf noch mit den notwendigen Bibliotheken gebunden.

Wieviel Strukturinformation noch vorhanden ist, hängt sehr von der Systemarchitektur ab; bei klassischen Singletasking-Systemen wie MS-DOS bestehen die ausführbaren Programme nur aus ausführbarem Code und Daten, bei neueren Multitasking-Systemen (z. B. Windows NE/PE[2], UNIX ELF[3]) können auch noch Bezeichner und insbesondere Verknüpfungen mit Bibliotheken enthalten sein.

**P-Code:** Kurz für Pseudo-Code; das Programm wird für eine virtuelle Maschine übersetzt, so daß der erzeugte Code relativ effizient interpretiert werden kann. Vorteile von P-Code sind (üblicherweise) Platzeffizienz und zumindest die Möglichkeit der Plattformunabhängigkeit. Heute wird P-Code etwa bei Java oder in älteren Versionen von Visual Basic verwendet.

## 2.7. Interpreter

Ein Interpreter ist ein Programm, mit dessen Hilfe man im Quellcode vorliegende Programme direkt ausführen kann. Das bedeutet, daß der Interpreter während der Laufzeit des Programms ebenfalls läuft und die Befehle des Programms Zeile für Zeile ausführt. Es findet normalerweise keine echte Übersetzung statt, d. h. der Interpreter erzeugt nicht dynamisch Code, um ihn direkt auszuführen, sondern wertet die Befehle

[1] Es muß nicht zwangsweise ein Unterschied zwischen Objektdateien und Binärprogrammen bestehen. Beispielsweise sind diese im Oberon-System identisch, auch zwischen Programmen und Bibliotheken wird dort kein Unterschied gemacht.

[2] new executable/portable executable

[3] enhanced link format

nur aus. Eine Übersetzung findet nur insofern gelegentlich statt, als daß der Quellcode tokenisiert[4] abgespeichert oder im Speicher gehalten wird.

Vom Standpunkt des Reverse Engineerings aus sind Interpreter relativ uninteressant, da der Quellcode (inklusive der Bezeichner und Typdeklarationen) ja bereits vorliegt. Allenfalls eine Tokenisierung stellt eine gewisse Erschwernis dar, die aber bei ausreichender Dokumentation des Dateiformats kein großes Hindernis darstellt. Dies ist höchstens von rechtlicher Relevanz; hierzu siehe Kapitel 3.5.4.

## 2.8. Disassemblierung

Disassemblierung in der trivialsten Form bedeutet zunächst nur die Umsetzung von Objektcode in einzelne Assemblerbefehle; dies kann etwa durch eine Übersetzungstabelle erfolgen. Zu einer Disassemblierung in diesem Sinn ist bereits ein Debugger in der Lage, oft kann dieser dabei auch noch symbolische Informationen anzeigen, die zur Erleichterung der Fehlersuche vom Compiler mit abgelegt wurden.

Wenn das Ziel ein komplettes Assemblerlisting ist, ist natürlich ein erheblicher Aufwand erforderlich, der bereits eine Vorstufe zu einer Dekompilierung darstellt. Hierzu gehört u. a. eine Trennung von Code- und Datenbereichen im Binärprogramm, die Identifizierung von Unterroutinen und eine Zuordnung von symbolischen Namen zu Adressen im Code- und Datenbereich.

## 2.9. Dekompilierung

Eine Dekompilierung ist der einer Kompilierung entgegengesetzte Prozeß, in dem aus einem Assemblerprogramm, einer Objektdatei oder einem Binärprogramm wieder ein Quellcode hergestellt werden soll. Dies muß nicht unbedingt automatisch, d. h. durch einen Decompiler, geschehen, sondern kann auch manuell durchgeführt werden; hierzu ist dann lediglich ein Disassembler oder Debugger nötig, da eine manuelle Disassemblierung zu aufwendig und fehlerträchtig wäre.

Idealerweise ist der resultierende Programmtext mit dem ursprünglichen Quellcode (abzüglich der Bezeichner, Kommentare, Formatierung, Aufteilung auf mehrere Dateien etc.) identisch. Dies kann natürlich allein deshalb schon nicht erreicht werden, da es für viele Operationen mehrere semantisch gleichwertige Konstrukte gibt, mit denen sie ausgedrückt werden können, die vom Compiler aber identisch umgesetzt werden.

Außerdem besteht das Problem, daß oft nicht bekannt ist, in welcher Quellsprache ein Programm ursprünglich geschrieben wurde oder daß diese Sprache nicht ausreichend beherrscht wird, um eine Dekompilierung durchzuführen. Im Falle einer automatischen Dekompilierung ist es wohl zu aufwendig, einen Decompiler zu schreiben, der die Quellsprache nicht nur identifizieren, sondern auch Code in ihr generieren

[4] Eine Art der Kompression, die auch die Ausführung beschleunigen kann. Tokenisierung war bei früheren BASIC-Dialekten weit verbreitet, wird aber auch heute noch bei in Office-Dokumenten abgelegten Skripten verwendet.

kann. Es sollte aber zumindest für die gängigen prozeduralen Programmiersprachen möglich sein, unabhängig von der Quellsprache als Ziel stets die gleiche, ausreichend universelle Sprache zu verwenden. Dies stellt normalerweise keine Beschränkung dar, wenn man die üblichen Einsatzzwecke betrachtet:

- Für eine Portierung in eine andere Programmiersprache oder auf eine andere Maschinenarchitektur ist es lediglich nötig, daß der Decompiler die entsprechende Zielsprache beherrscht; hier ist es sogar ein Vorteil, wenn die ausgegebene Sprache nicht von der ursprünglichen abhängt. Wenn allerdings der Quellcode vorhanden ist, ist es sicher sinnvoller, diesen zu übersetzen.

- Für eine Sicherheitsanalyse im Sinne einer Untersuchung auf absichtlich verborgene Dysfunktionalitäten wie etwa eine Trojanisierung ist die gewählte Sprache relativ unwichtig, solange der Untersuchende sie beherrscht. Für eine weitergehende Sicherheitsanalyse in Hinblick auf unbeabsichtigte Safety- oder Security-Implementationsmängel gilt dies aufgrund sprachspezifischer sicherheitsrelevanter Eigenschaften natürlich nur eingeschränkt[5]; allerdings könnte hier auch eine Disassemblierung angebracht sein, da derartige Sicherheitsmängel auch abhängig vom Compiler (etwa bei bestimmten Optimierungsstufen) oder den Compiler-Bibliotheken auftreten können. Bei Malware, insbesondere Viren, muß man sich ggf. sowieso auf eine Disassemblierung beschränken, da diese oft in Assembler geschrieben werden.

- Beim Entwurf von Compilern ist eine Disassemblierung sicherlich vorzuziehen, da nur so Eigentümlichkeiten im generierten Code auffallen können. Dies gilt insbesondere für die Untersuchung von optimiertem Code, da von einem Decompiler erwartet wird, daß er unabhängig von einer Optimierung immer denselben Quellcode erzeugt.

- Lediglich wenn ein fehlender Quellcode – etwa zur Fehlerkorrektur – wiedergewonnen werden soll, besteht ein Interesse, daß dies in einer bestimmten Programmiersprache geschieht.

[5] Beispiel: Die in C-Programmen häufig auftretenden Buffer-Overflows sind in anderen Sprachen wie LISP oder Java prinzipiell nicht möglich.

# 3. Rechtliche Rahmenbedingungen

Daß die bisher genannten Methoden ein Reverse Engineering ermöglichen, bedeutet natürlich nicht, daß dies auch in jedem Fall gestattet ist. Über die rechtliche (Un-)Zulässigkeit des Kopierens von Software, das im Vergleich zu Literatur oder Musik wesentlich zurückhaltender geregelt ist, sind heute die meisten informiert. Daß man eine Software legal erworben hat, bedeutet aber nun nicht, daß man damit alles machen kann, was man möchte, solange man sie nicht weitergibt oder mehrfach nutzt. Selbst für kostenlose Software können gewisse Nutzungseinschränkungen bestehen.

Andererseits kann der Urheber in seine Lizenzvereinbarungen auch keine beliebigen Nutzungsbeschränkungen aufnehmen, da dem Nutzer bestimmte Rechte gesetzlich garantiert sind. Auf die Frage, ob die Lizenzvereinbarungen überhaupt gültig sind, soll hier nicht weiter eingegangen werden.

Nachfolgend sollen die aktuelle Situation in Deutschland dargestellt und daneben noch interessante Aspekte aus anderen Staaten erwähnt werden.

## 3.1. Relevante Gesetze und Richtlinien

### 3.1.1. Die EU-Softwarerichtlinie 91/250/EEC

Die EU-Softwarerichtlinie vom 14.5.1991 (siehe Anhang A) stellt den Rahmen dar, in dem die Mitgliedstaaten den Softwareschutz in ihren jeweiligen Landesgesetzen umzusetzen hatten. Im Falle Deutschlands sind alle relevanten Artikel in den Paragraphen 69a–g Urheberrechtsgesetz umgesetzt worden. Da diese größtenteils wörtlich der Softwarerichtlinie entsprechen, sollen im folgenden die Paragraphen §§ 69a ff systematisch behandelt werden.

Es sei aber darauf hingewiesen, daß schon in der Softwarerichtlinie und den dazugehörigen Erwägungsgründen Widersprüche zu finden sind. So ist beispielsweise nach dem Wortlaut der Artikel 5 Abs. 1 und Artikel 9 eigentlich klar ausgedrückt, daß das dem Nutzungsberechtigten zustehende Recht der Fehlerberichtigung vertraglich verboten werden kann, während der 17. Erwägungsgrund dies verneint. Dies wird auch in [15] bemerkt, wo sich die folgende zusätzliche Erläuterung findet:

> „Auf diesen Widerspruch hingewiesen, erklärte ein maßgebliches Mitglied der zuständigen Generaldirektion bei der EG, daß „selbstverständlich“ nicht der eigentliche Richtlinientext sondern die Erwägungsgründe ausschlaggebend seien – eine Behauptung, für die deutsche Richter wohl wenig Verständnis aufbringen werden.“

Auch zum Punkte der Zulässigkeit einer Dekompilierung besteht eine solche Diskrepanz. So ist dem Artikel 5 Abs. 1 zu entnehmen, daß für eine Fehlerberichtigung alle Handlungen von Artikel 4 Buchstaben a) und b) zulässig sind, also auch eine Dekompilierung. Dagegen findet sich im 21. Erwägungsgrund die folgende gegenteilige Aussage, die sich auf die Zulässigkeit der Dekompilierung zur Herstellung von Interoperabilität nach Artikel 6 bezieht:

> „Folglich ist davon auszugehen, daß *nur* in diesen begrenzten Fällen eine Vervielfältigung und *Übersetzung* [...] rechtmäßig ist [...]“

Eine weitere Ungereimtheit ergibt sich darin, daß der Artikel 6 eindeutig von einer Interoperabilität zwischen eigener und fremder *Software* spricht (so auch [12]), während in den Erwägungsgründen 10–11 und 22 teilweise auch von *Hardware* die Rede ist oder keine Einschränkung auf Software gemacht wird:

> „Zu diesem Zweck ist eine logische und, wenn zweckmäßig, physische Verbindung und Interaktion notwendig, um zu gewährleisten, daß Software und *Hardware* mit anderer Software und *Hardware* und Benutzern wie beabsichtigt funktionieren können. Die Teile des Programms, die eine solche Verbindung und Interaktion zwischen den Elementen von Software und *Hardware* ermöglichen sollen, sind allgemein als „Schnittstellen“ bekannt.“
>
> [...]
>
> „Ein Ziel dieser Ausnahme ist es, die Verbindung *aller* Elemente eines *Computersystems*, auch solcher verschiedener Hersteller, zu ermöglichen, so daß sie zusammenwirken können.“

### 3.1.2. Systematik der §§ 69a ff Urheberrechtsgesetz

Im folgenden werden die in Bezug auf Reverse Engineering entscheidenden Teile der §§ 69a ff UrhG (siehe Anhang B.1) besprochen. Keine Berücksichtigung finden die Abschnitte, die für das Thema nicht relevant sind, etwa wer der Urheber und wer berechtigter Benutzer des Programms ist. Diese Punkte werden als geklärt vorausgesetzt, d. h. der an einer Programmanalyse interessierte Benutzer sei zur Verwendung des Programmes berechtigt, habe aber – bis auf den üblichen Lizenzvertrag – keine Sondervereinbarungen mit dem Urheber.

#### § 69a Gegenstand des Schutzes

Hier ist der Absatz 2 interessant, der besagt, daß zwar die konkrete Ausdrucksform eines Programms geschützt ist, jedoch nicht die zugrundeliegenden Ideen und Grundsätze, auch nicht die der Schnittstellen. Das heißt, daß z. B. Algorithmen, Benutzeroberfläche und Dateiformate nicht durch das Urheberrecht geschützt werden.

Es kann jedoch ein Schutz durch das Patentrecht oder andere Gesetze bestehen. Geschützt werden aber alle Entwicklungsstufen des Programms wie etwa Quellcode und Entwurfsmaterial sowie das sogenannte „Gewebe von Algorithmen[1]“ [37].

### § 69c Zustimmungsbedürftige Handlungen

Der § 69c gibt dem Urheber des Programmes sehr weitreichende Rechte; diese werden aber in den folgenden Paragraphen teilweise eingeschränkt. Für uns wichtig ist hier die Nr. 2 (Umarbeitung), unter die Disassemblierung und Dekompilierung fallen, aber auch andere Änderungen wie eine Fehlerkorrektur („Patchen“) oder eine Übersetzung von einer Programmiersprache in eine andere. Dabei ist auch die Nr. 1 (Vervielfältigung) u. U. relevant, da eine Vervielfältigung eines Programmes ja auch vorliegt, wenn ein Assembler- oder Quellcodelisting ausgedruckt oder gespeichert wird.

### § 69d Ausnahmen von den zustimmungsbedürftigen Handlungen

Hier werden die weitgehenden Rechte des Urhebers aus § 69c eingeschränkt. Nach Abs. 1 darf der Benutzer alle Handlungen aus § 69c Nr. 1 und 2 vornehmen, sofern sie für eine bestimmungsgemäße Benutzung des Programms notwendig sind (z. B. zur Fehlerberichtigung) und keine besonderen vertraglichen Bestimmungen[2] vorliegen. Wie bereits in Kapitel 3.1.1 erläutert, dürfen laut den Erwägungsgründen der Softwarerichtlinie das Laden und Ablaufen und die Fehlerberichtigung nicht untersagt werden. In [15] wird dies dahingehend eingeschränkt, daß eine Fehlerkorrektur durch den Benutzer dann verboten werden darf, wenn sich der Hersteller zu einer Berichtigung auftretender Fehler zu vernünftigen wirtschaftlichen Konditionen verpflichtet. Dies dürfte in der Praxis aber kaum der Fall sein, da in Softwarelizenzverträgen meist keinerlei Garantien gegeben werden. Selbst die Eignung für irgendeinen bestimmten Zweck wird oft explizit ausgeschlossen, so daß sich die Frage stellt, was denn die oben genannte „bestimmungsgemäße Benutzung“ sein soll. Unter die Fehlerberichtigung fallen auch Fehler, die beim Erwerb des Programms noch nicht vorhanden waren, wie etwa eine Vireninfektion; in diesem Fall wäre die Verwendung eines Antivirusprogramms gedeckt [38].

Die Frage, ob eine Dekompilierung nach § 69d Abs. 1 zulässig sein kann, wird unter § 69e diskutiert; technisch dürfte aber für eine Fehlerbehebung („Patch“) eine zumindest partielle Disassemblierung nötig sein.

Ein weiteres Beispiel für eine zulässige Maßnahme soll nach [12, 38] eine Portierung sein, da diese nötig sei, um ein Programm und die daran hängenden Datenbestände nach einem Wechsel des Betriebssystems oder der Hardware weiter nutzen zu können. Eine echte Portierung erfordert allerdings mindestens eine Dekompilierung und eine darauf aufsetzende Rekompilierung für das Zielsystem (ggf. auch noch

[1] Mit „Gewebe von Algorithmen“ ([37]) ist offenbar gemeint, daß mehrere Algorithmen so zusammengekoppelt werden, daß sie zusammenarbeiten können.

[2] Auf die prinzipielle Gültigkeit von Softwarelizenzverträgen („EULA“), die ja häufig auf das US-Rechtssystem zugeschnitten sind, soll hier nicht eingegangen werden.

manuelle Anpassungen am Code), so daß diese Forderung recht weitreichend ist. Abgesehen davon wird in [12, 38] die Frage der Dekompilierung sehr kritisch betrachtet; wahrscheinlich sind den Verfassern die technischen Implikationen nicht bewußt.

Um das Programm weiter nutzen zu können, würde jedenfalls auch eine Emulation ausreichen, die entweder nur eine passende Betriebssystemumgebung (VMWare, Plex86 [34], DOSEMU [7], WINE [45]) oder auch einen passenden Prozessor (Bochs [2], FX!32, PowerMac) zur Verfügung stellt. Dabei gibt es fließende Übergänge zur De- und folgenden Rekompilierung, die teilweise automatisch zur Performanzoptimierung durchgeführt wird (FX!32, Transmeta Crusoe [40]). Eine echte Portierung ist zumindest prinzipiell auch möglich, ohne daß der dekompilierte Quellcode sichtbar wird [6], was die Rechte des Urhebers sicherlich ausreichend schützt.

Nach Abs. 3 darf der Benutzer das Funktionieren des Programms untersuchen, um die zugrundeliegenden Ideen und Grundsätze (die nach § 69a Abs. 2 ja nicht geschützt sind) zu ermitteln. Das darf dabei aber nur durch Handlungen geschehen, zu denen er ohnehin berechtigt ist. Der Sinn dieser Regelung liegt offenbar darin, daß der Benutzer ansonsten die Handlungen, zu denen er im Prinzip berechtigt wäre, nur dann durchführen dürfte, wenn er damit das Programm *bestimmungsgemäß* benutzen wollte, worunter eine Analyse wohl nicht fällt. Außerdem wird betont, daß dieses Recht nicht vertraglich untersagt werden kann [38].

### § 69e Dekompilierung

Die Vervielfältigung des Codes oder die Übersetzung der Codeform im Sinne des § 69c Nr. 1 und 2 (also z. B. Disassemblierung oder Dekompilierung, jedoch keine Veränderungen wie etwa durch Patchen) ist nach Absatz 1 zulässig, wenn dies unerläßlich ist, um die erforderlichen Informationen zur Herstellung der Interoperabilität eines unabhängig geschaffenen Computerprogramms mit anderen Programmen zu erhalten, sofern folgende Bedingungen erfüllt sind:

1. Durchführung nur durch nutzungsberechtigte Personen oder von ihnen Bevollmächtigte.

2. Die notwendigen Informationen sind für die in Nummer 1 genannten Personen noch nicht ohne weiteres zugänglich gemacht. Laut [25] bedeutet „ohne weiteres“ wohl leicht und schnell; heute dürfte man z. B. eine – ggf. auf Programmbesitzer beschränkte – Verfügbarkeit im Internet fordern können. Nach [12] darf insbesondere für die Informationen kein Entgelt verlangt werden, sofern dies nicht eine reine Aufwandsentschädigung darstellt. Fraglich ist natürlich die Situation, in der die öffentliche Spezifikation unvollständig oder inkorrekt ist; dies dürfte u. U. nur schwer nachweisbar sein.

3. Die Handlungen beschränken sich auf die Teile des ursprünglichen Programms, die zur Herstellung der Interoperabilität notwendig sind. Dabei kann es durchaus erforderlich sein, nicht nur die Schnittstellen des Ursprungsprogramms zu

dekompilieren (sofern sich diese überhaupt abgrenzen lassen), sondern auch weitere Teile oder sogar das ganze Programm [12].

Dabei dürfen nach Absatz 2 die gewonnenen Informationen nicht

1. zu anderen Zwecken als zur Herstellung der Interoperabilität des unabhängig geschaffenen Programms verwendet werden. D. h. auch wenn die Algorithmen des Programms nicht durch das Urheberrecht geschützt werden, so darf man sie trotzdem nicht weiter verwenden, wenn man sie als „Abfallprodukt“ der Dekompilierung erhält.

2. an Dritte weitergegeben werden, wenn dies nicht für die Interoperabilität nötig ist. Die Notwendigkeit der Weitergabe kann etwa bestehen, wenn wie in [15] gefordert, Analyse und Programmierung personell getrennt werden, um den Punkt 1 sicherzustellen („clean room procedures“).

   Ein weiterer betrachtenswerter Fall scheint der zu sein, in dem das unabhängig geschaffene Programm als Open Source, also im Quellcode vertrieben wird. In diesem Fall würde die Schnittstelleninformation implizit mit dem Code veröffentlicht. Dies kann durchaus erforderlich sein; wenn ein Programm beispielsweise der GNU Public License unterliegt, muß der vollständige Quellcode offengelegt werden. Hier bieten sich mehrere Möglichkeiten an:

   - Man betrachtet die Weitergabe der Informationen als notwendig für die Interoperabilität.
   - Man betrachtet dies nicht als Weitergabe der Informationen, da auch bei vorliegendem Quellcode erst durch Reverse Engineering auf die Spezifikationen geschlossen werden muß.
   - Man entwickelt ein Plugin-Konzept, das es erlaubt, den für die Interoperabilität zuständigen Programmteil nur in Binärform zu verteilen. Dies widerspricht leider der Open Source-Philosophie.

3. für die Entwicklung oder Vermarktung eines Programms mit im wesentlichen ähnlicher Ausdrucksform oder für irgendwelche anderen das Urheberrecht verletzenden Handlungen verwendet werden. Laut [25] ist die Dekompilierung aber durchaus zulässig, um ein unabhängig geschaffenes Konkurrenzprodukt interoperabel zu machen. Dies ist ein wichtiger Punkt, schließlich könnte sonst kaum ein Konkurrenzprodukt zu einem Programm geschaffen werden, das dessen Dateiformat verarbeiten kann, wenn der Hersteller des ursprünglichen Programms nicht kooperiert; die einzige Möglichkeit bestünde in einer direkten Analyse des Dateiformats, bei der aber Unklarheiten entstehen könnten.

### § 69g Anwendung sonstiger Rechtsvorschriften, Vertragsrecht

Nach Abs. 1 bleiben andere Rechtsvorschriften, die Computerprogramme betreffen, unberührt. Dies kann z. B. Patentrecht, Kennzeichenschutz, § 1 UWG (unlauterer

Wettbewerb) und § 17 UWG (Schutz von Geschäfts- und Betriebsgeheimnissen) betreffen.

Nach Abs. 2 können die Ausnahmen von § 69d Abs. 2 (Sicherungskopie) und 3 (allgemeine Analyse) und § 69e (Dekompilierung) nicht vertraglich verboten werden. Wie schon in Kapitel 3.1.1 erwähnt, erstreckt sich dies laut den Erwägungsgründen offenbar auch auf große Teile von § 69d Abs. 1.

## 3.2. Bisherige Rechtsprechung in Deutschland

Leider hat es bisher nur wenige Urteile gegeben, die sich auf die Reverse Engineering-spezifischen Paragraphen (§ 69c Nr. 2, § 69d Abs 1 & 3, § 69e) beziehen; fast alle davon befassen sich zudem mit der Zulässigkeit der Entfernung einer Dongleabfrage [23, 24, 28, 29, 30].

### 3.2.1. Dekompilierung zur Erzielung von Interoperabilität

Der Fall „Kostenlose Kassenzahnarzt-Software“ [1] hat im Prinzip nichts mit Reverse Engineering zu tun; es wird aber bei der Urteilsbegründung in folgender Weise auf § 69e eingegangen:

> „Da es technisch erfahrungsgemäß keine nicht behebbaren Schwierigkeiten bereitet, verschiedene Software interoperabel zu gestalten und das Dekompilieren eines Programms zu diesem Zweck rechtlich nicht zu beanstanden ist (vgl. § 69e UrhG), [...] “

Da dieser Bezug für das Urteil keine Relevanz besitzt, soll hier nicht weiter auf den Sachverhalt eingegangen werden. Man sieht aber, daß der BGH hier die Neuregelung nach der Softwarerichtlinie bestätigt; allerdings wird die Formulierung bereits in [1] mit dem Satz „Die These, das Dekompilieren eines Programms sei zum Zweck der Herstellung der Interoperabilität nicht zu beanstanden, lädt in dieser Kürze zu Mißverständnissen förmlich ein.“ kritisiert, worauf noch weitere Erläuterungen folgen.

### 3.2.2. Entfernung einer Dongle-Abfrage

Vor der Änderung des Urheberrechts durch die Softwarerichtlinie wurde die Umgehung von Dongles durch die Gerichte eindeutig als wettbewerbswidrig bewertet [22]. Das erste nach der Änderung gefällte Urteil [24] ließ diese im Gegensatz zu, was hauptsächlich mit § 69d Abs. 1 begründet wurde; außerdem wird für die Fehleranalyse § 69 Abs. 3 genannt. Es scheint aber, als ob das Gericht mit der Neuregelung noch leichte Probleme hat, da auch der § 69e herangezogen wird, obwohl es sich hierbei weder um eine Dekompilierung noch um die Herstellung einer Interoperabilität handelt. In [22] wird zudem kritisiert, daß das Gericht § 69f Abs. 2 (Mittel zur unerlaubten Beseitigung technischer Programmschutzmechanismen) nicht berücksichtigt, obwohl der Autor dem Urteil zustimmt, da durch Dongles vielfältige Probleme auftreten können.

Durch die Berufungsinstanz wurde das Urteil allerdings aufgehoben [29], und auch alle weiteren diesbezüglichen Urteile [23, 28, 30] stufen die „Entdonglierung“ eines Programms als unzulässig ein. Dabei wird kaum auf technische Details eingegangen, bzw. es wird sogar explizit eine funktionale Betrachtungsweise gefordert [23], weshalb diese Urteile leider nicht viel über Reverse Engineering *per se* aussagen. Der Grundgedanke scheint jedenfalls zu sein, daß § 69d Abs. 1 deshalb nicht anwendbar ist, weil ein Betrieb der Software ohne Dongle nicht bestimmungsgemäß, sondern im Gegenteil bestimmungswidrig ist, da die Dongleabfrage vom Programmschöpfer vorgesehen war.

## 3.3. Anwendbarkeit auf verschiedene Analysemethoden

Im folgenden soll untersucht werden, für welche Zwecke und mit welchen Mitteln eine Analyse gerechtfertigt sein kann.

### 3.3.1. Beobachtende Analyse

Da diese Art der Analyse lediglich auf Handlungen beruht, die auch bei der normalen Verwendung des Programms zulässig sind, ist sie im Rahmen der sogenannten „Experimentierklausel“ [25] § 69d Abs. 3 uneingeschränkt zulässig, ohne daß eine besondere Begründung erforderlich wäre.

### 3.3.2. Tracing

Ein Tracing stellt keinen Eingriff [12] in das untersuchte Programm dar, sondern setzt lediglich eine angepaßte Umgebung ein, um die Beobachtungsmöglichkeiten zu verbessern. Laut [38] sind alle sogenannten *Black-box-Techniken* nach § 69d Abs. 3 zulässig, wozu auch Speicherabzüge und die Protokollierung der Signalkommunikation zählen. Somit dürfte auch der Einsatz von Hardware-Tracern wie etwa Logikanalysatoren oder Debugging-Hilfen wie dem Periscope Model IV erlaubt sein.

### 3.3.3. Debugging

Laut [38] dürfen nach § 69d Abs. 3 auch „ [...] Tools eingesetzt werden, die den Ablauf des Programms in einzelne Befehlsschritte aufteilen und damit ein Verfolgen des Programmablaufs ermöglichen, *ohne den Programmcode zu kennen*, beispielsweise Debugger oder Linetracer, mit denen man die Zusammenarbeit von Programmen an Hand von deren Signalkommunikation beobachten kann.“ Unklar ist leider, was der Autor exakt unter einem Debugger versteht, da derartige Tools üblicherweise sowohl Eingriffe in das untersuchte Programmen vornehmen als es auch partiell disassemblieren können. Denkbar wäre es natürlich, sich im Gebrauch des Debuggers zu beschränken und diese Möglichkeiten nicht zu nutzen, was den Nutzen aber beträchtlich reduzieren dürfte.

Aussichtsreicher sieht ein Debugging nach § 69d Abs. 1 aus, da die Fehlerbehebung der ureigenste Zweck eines Debuggers ist und die Regelung zu diesem Zweck alle in § 69c Nr. 1 genannten Handlungen gestattet. Hierbei dürfte auch die u. U. strittige partielle Disassemblierung gerechtfertigt sein.

### 3.3.4. Disassemblierung und Dekompilierung

Zwischen den beiden technischen Begriffen wird rechtlich nicht unterschieden, es handelt sich in beiden Fällen, wie auch bei Assemblierung und Kompilierung, um eine Übersetzung der Codeform nach § 69c Nr. 2. Im folgenden sollen sie daher austauschbar verwendet werden.

Relativ eindeutig ist die Zulässigkeit einer Dekompilierung nach § 69e. Zu bemerken wäre hier lediglich, daß Schnittstellen nicht vom Hersteller als solche vorgesehen sein müssen, sondern auch unabhängig entwickelte Möglichkeiten des Datenaustausches dazugehören, d. h. die Schnittstelle eines Programms wird „ [...] nicht mehr alleine vom Ausgangsprogrammierer definiert, sondern vor allem von später darauf folgenden Programmherstellern.“ [15] Hierzu gehören auf jeden Fall undokumentierte Funktionen[3], die eigentlich nur für interne Zwecke gedacht sind. Dies dürfte aber auch für die Entwicklung beliebiger Zusatzmodule (auch „Plugins“ oder „Add-ons“ genannt) gelten, die die Funktionalität eines existierenden Programms erweitern sollen, solange dabei nicht der ursprüngliche Code verändert wird.

Dem Wortlaut von § 69d Abs. 1 nach sind aber alle Maßnahmen von § 69c Nr. 1 und 2, also auch eine Dekompilierung, gerechtfertigt, wenn sie für eine bestimmungsgemäße Benutzung einschließlich der Fehlerberichtigung notwendig sind. Wie schon in Kapitel 3.1.1 erläutert wird dies bereits in den Erwägungsgründen zur Softwarerichtlinie eingeschränkt; dieser Punkt wird daher in der Literatur kontrovers diskutiert.

Die Meinung, daß eine Dekompilierung ausschließlich nach § 69e UrhG bzw. nach Artikel 6 Softwarerichtlinie zulässig ist, wird in [36, 37, 25] vertreten. Dabei wird in [25] der Grundsatz „lex specialis derogat legi generali“ angeführt, d. h. eine Spezialvorschrift hat Vorrang vor einer allgemeineren. In [17, 38] wird dieser Punkt als fraglich bezeichnet, während in [12] eine Dekompilierung „ [...] in ganz besonderen Ausnahmefällen, z. B. bei Vorliegen eines besonders schweren Programmfehlers [...] “ für zulässig gehalten wird. Einzig in [15] wird die Zulässigkeit einer Dekompilierung nach Art. 5 Abs. 1 Softwarerichtlinie bejaht.

## 3.4. Betrachtung zweier prominenter Fälle

### 3.4.1. Entwicklung des Linux-Treibers für Parallelport-ZIP-Laufwerke

Zur Zeit der Entwicklung des ersten Linux-Treibers für das externe Parallelport-ZIP-Laufwerk (1995) hatte der Hersteller IOMEGA noch keine Spezifikationen über das

[3] Eine lange Tradition undokumentierter Funktionen findet sich z. B. bei den Microsoft-Betriebssystemen.

Protokoll zur Verfügung gestellt, mit dem das Laufwerk angesprochen wurde. Wie in [14] beschrieben griff der Entwickler daher zu Methoden des Reverse Engineerings:

> „The driver was developed without the benefit of any technical specifications for the interface. Instead, a modified version of DOSemu was used to monitor the protocol used by the DOS driver, 'guest.exe', for this adapter. I have no idea how my programming model relates to IOMEGA's design."

Es wurde also keine direkte Analyse des vorhandenen Treibers vorgenommen, sondern dieser wurde in einer überwachten Umgebung eingesetzt, um die Signalkommunikation zu beobachten. Dies ist, wie schon in Kapitel 3.3.2 beschrieben, nach § 69d Abs. 3 zulässig, um die zugrundeliegenden Ideen und Grundsätze – in diesem Fall das Protokoll – zu ermitteln.

Nicht zulässig wäre es hingegen gewesen, zu diesem Zweck den Treiber zu dekompilieren, da dies nur zur Herstellung der Interoperabilität zwischen Software und Software, nicht aber zwischen Software und Hardware zulässig ist. Diese Problematik wurde bereits in Kapitel 3.1.1 erläutert. Es kann auch nicht damit argumentiert werden, daß eine Interoperabilität zwischen dem Linux-Kernel und dem IOMEGA-Treiber hergestellt wird, da der DOS-Treiber nicht eingebunden, sondern nachempfunden werden soll. Zur Frage der Zulässigkeit einer Portierung siehe Kapitel 3.1.2, § 69d; in diesem Fall wird die Lauffähigkeit des Treibers bereits durch DOSEMU [7] erreicht.

### 3.4.2. Angriff auf CSS (Content Scrambling System)

Das Kopierschutzverfahren CSS [3], welches das Duplizieren von DVD-Videos verhindern soll, wurde im Laufe des Jahres 1999 in mehreren Schritten geknackt. Bekannt geworden ist hierbei die Entwicklung des Tools DeCSS durch eine norwegische Hackergruppe, das die kompletten DVD-Datenströme entschlüsselt und auf die Festplatte kopiert. Dabei war ein wesentlicher Teil der Arbeit bereits durch Linux-Programmierer geleistet worden, die die Authentifizierungs- und Verschlüsselungsalgorithmen – ebenfalls durch Reverse Engineering – herausgefunden hatten.

Der Anteil der norwegischen Hackergruppe bestand darin, durch das Reverse Engineering des Software-Decoders Xing-DVD sogenannte Player Keys [4] zu erlangen und diese mit einer Oberfläche und den bereits entwickelten Algorithmen zu kombinieren.

Ob dieses Vorgehen durch das deutsche Urheberrecht [5] gedeckt ist, ist nicht ganz klar. Eine Rechtfertigung könnte durch § 69e erfolgen, und zwar insofern, als daß für eine Player-Software die Interoperabilität mit den entsprechenden Video-DVDs

[4] Jede lizensierte DVD-Decoder-Soft- oder -Hardware enthält zwei Player-Keys, mit denen sie sich über das DVD-Laufwerk bei der DVD authentifizieren kann und dann die zur Entschlüsselung der DVD nötigen Disk- und Title Keys erhält. Zu diesem Zweck enthält jede CSS-verschlüsselte DVD die Schlüssel aller erlaubten Player – insgesamt 408 Schlüssel. [16]

[5] Offenbar stammten die Player Keys von einem deutschen Mitglied der Hackergruppe. [16]

hergestellt werden soll. Daß die hergestellte Interoperabilität sich nicht auf die analysierte Software (Xing-DVD) bezieht, steht dem Wortlaut von § 69e Abs. 1 nicht entgegen; dort wird lediglich gefordert, daß eine Dekompilierung „unerläßlich ist, um die erforderlichen Informationen zur Herstellung der Interoperabilität eines unabhängig geschaffenen Computerprogramms mit anderen Programmen zu erhalten“. Die wichtigste Frage ist allerdings, wieweit die DVD hier als Hard- oder Software einzustufen ist [39] bzw. ein Programm darstellt – verschlüsselt werden nur die Video- und Audiodaten, die weiteren Informationen zur Menüsteuerung etc. werden von DeCSS nicht berücksichtigt [16].

Klar ist jedenfalls, daß die Algorithmen und Schlüssel selbst nicht durch das Urheberrecht geschützt werden; allerdings sind sie Betriebsgeheimnisse der Industrie und werden durch das Gesetz gegen den unlauteren Wettbewerb (UWG) geschützt, wonach auch die Verbreitung dieser Informationen unzulässig ist und zu Strafe oder Schadensersatz führen kann [21].

Auf jeden Fall problematisch ist die Veröffentlichung der erlangten Informationen. Die Verbreitung als binäres, interoperables Programm ist nach § 69e natürlich zulässig, die Offenlegung des Quellcodes, in dem die Schlüssel sichtbar sind, evtl. schon nicht mehr [39]. Zur Open Source-Problematik siehe auch die Erläuterung in Kapitel 3.1.2, § 69e.

## 3.5. Anwendbarkeit bei Malware

Im Falle von Malware stellt sich die rechtliche Situation etwas komplizierter dar, jedoch ist bei einer vorhandenen automatischen Replikation anzunehmen, daß der Autor kein Interesse daran hat, die Vervielfältigung zu kontrollieren, sondern daß ihm im Gegenteil daran gelegen ist, daß das Programm eine möglichst große Verbreitung findet. Man darf also davon ausgehen, daß man zur Benutzung und Vervielfältigung jeglicher selbstreplizierender Malware berechtigt ist, die man erhält. Die anderen Rechte stehen dem Urheber aber prinzipiell zu, auch wenn dieser anonym bleibt. Ein Lizenzvertrag liegt hier natürlich nicht vor, so daß der Vorbehalt der vertraglichen Bestimmungen entfällt.

### 3.5.1. Beobachtende Analyse & Tracing: § 69d Abs. 3

Definitiv zulässig ist eine Analyse nach § 69d Abs. 3, also durch den Einsatz von Tracing und evtl. auch Debugging (siehe Kapitel 3.3.3). Da gerade Viren im Codeumfang und damit auch im Verhaltensrepertoire recht klein sind, kann man dadurch bereits viele ihrer Eigenschaften aufklären. Weil man aber keinen umfassenden Test machen kann, können nur gelegentlich auftretende Funktionen, sog. “logische Bomben“, nur zufällig entdeckt werden. Zudem setzt man sich durch das Ausführen des bösartigen Codes unnötigerweise Gefahren aus.

### 3.5.2. Fehlerberichtigung: § 69d Abs. 1

Eine Analyse nach § 69d Abs. 1 erschließt sich dagegen nicht so leicht. Es ist zwar erlaubt, ein Programm von einer Vireninfektion zu befreien, wenn diese als Fehler betrachtet wird (siehe Kapitel 3.1.2, § 69d); wenn für diese Aufgabe kein Antivirusprogramm zur Verfügung steht, wird man dies auch von Hand tun dürfen. Dabei besteht aber das Problem, daß hierdurch nur die Modifikation am Wirtsprogramm gerechtfertigt wird, nicht aber die Untersuchung des Virus, da dieser lediglich technisch, nicht aber urheberrechtlich ein Teil des infizierten Programms ist[6]. Man kann also höchstens den Schluß ziehen, daß das Analysieren einer Malware nach § 69d Abs. 1 dann zulässig ist, wenn dies notwendig ist, um die Malware bestimmungsgemäß benutzen zu können (Also etwa, wenn ein Virus sich nicht korrekt repliziert).

Eine Möglichkeit bieten allenfalls die Trojanischen Pferde, bei denen die bösartigen Funktionen in der vorliegenden Dokumentation nicht erwähnt werden; diese könnte man nämlich – wie auch undokumentierte Dysfunktionalitäten in anderen Programmen – als Fehlfunktionen ansehen und als solche ohne weiteres entfernen, was eine partielle Analyse erfordert.

### 3.5.3. Dekompilierung & Disassemblierung: § 69e

Eine Analyse nach § 69e wäre denkbar, wenn es darum geht, eine Malware mit einem Antivirusprogramm bezüglich der Erkennung und Reinigung interoperabel zu machen. Für die Erkennung muß lediglich eine Signatur erstellt werden. Hierzu dürfte es bei den meisten Viren ausreichen, den Code von dem des Wirtsprogramms abzugrenzen[7], bei verschlüsselten Viren wird zusätzlich eine Analyse der Verschlüsselung notwendig. Hinsichtlich der Reinigung dürfte man eine Analyse des Replikations- bzw. Integrationsmechanismus vornehmen.

Die größte Einschränkung besteht hierbei durch § 69e Abs. 2 Nr. 2, wo die Weitergabe der gewonnenen Informationen verboten wird. Dabei ist dies ein sehr wichtiger Punkt der Malware-Analyse, da das Interesse ja gerade darin liegt, die Öffentlichkeit über das spezifische Verhalten aufzuklären.

### 3.5.4. Skript-Malware

Da bei der immer häufiger auftretenden Skript-Malware der Quellcode direkt gelesen werden kann, ist keine Übersetzung der Codeform nach § 69c Abs. 2 für die Analyse nötig. Mit Einschränkungen gilt dies auch für die Makrosprachen wie VBA (Visual Basic for Applications); diese liegen zwar prinzipiell im Quellcode vor, werden aber meist tokenisiert, d. h. komprimiert abgespeichert, was eine Übersetzung der Codeform (auch wenn der Übersetzungsvorgang trivial ist) nötig macht. Hier könnte evtl. argumentiert werden, daß das Laden der Makros mit der dazugehörigen Anwendung sicherlich bestimmungsgemäß ist, und daß demzufolge auch alle Operationen, die

[6] Die Entfernung des Virus kann hingegen keine Urheberrechtsverletzung an diesem sein, da das Löschen nicht unter die dem Rechtsinhaber vorbehaltenen Handlungen aus § 69c fällt.

[7] Andere Arten von Malware existieren bereits als eigenständige Programme.

man in der Anwendung vornimmt, zulässig sind, einschließlich der Betrachtung des Malware-Codes[8].

### 3.5.5. Mögliche rechtliche Konsequenzen einer unzulässigen Analyse

Da es sehr wichtig ist, die genaue Funktionsweise von Malware zu untersuchen, damit man eine Abschätzung des Risikos vornehmen, sie wieder von einem infizierten System entfernen und einen eventuellen Schaden soweit wie möglich beheben kann, und da eine Analyse nach § 69d Abs. 3 keine Vollständigkeit der erhaltenen Information gewährleistet, wird es sich in der Praxis nicht vermeiden lassen, eine Disassemblierung oder Dekompilierung durchzuführen oder zumindest einen Debugger zu verwenden.

Ähnliches gilt allgemein für Software in Bereichen mit hohen Sicherheitsanforderungen. Dort darf Software weder Eigenschaften besitzen, die die Security verletzen (etwa unzureichenden Zugriffsschutz oder sogar selbständige Übertragung von Daten nach außen) noch solche, die die Safety verletzen (etwa die absichtliche oder unabsichtliche Zerstörung von Daten oder unvorhersehbare Abstürze). Dabei kommen sowohl absichtlich eingebaute Funktionen („Backdoors“, trojanische Funktionen) als auch unbeabsichtigte Fehler („Bugs“) in Frage. Aus diesen Gründen wird häufig Open Source-Software eingesetzt, deren Quellcode man zumindest in der Theorie auf unerwünschte Eigenschaften hin untersuchen kann; dies ist aber nicht immer möglich oder gewollt. Da zum einen bestimmte Fehlfunktionen nur in selten auftretenden Situationen vorkommen und zum anderen absichtliche Backdoors oft getarnt werden, wird eine Analyse nach § 69d Abs. 3 nicht in allen Fällen die Sicherheitsanforderungen erfüllen. Aus diesen Gründen soll im folgenden ein Überblick gegeben werden, mit welchen Folgen der Analysierende ggf. zu rechnen hat.

#### Zivilrechtlich: § 97 UrhG. Anspruch auf Unterlassung und Schadenersatz

Nach § 97 UrhG (siehe Anhang B.1) kann man wegen einer Verletzung des Urheberrechts *vom Verletzten* auf Beseitigung der Beeinträchtigung, bei Wiederholungsgefahr auf Unterlassung und, bei Vorsatz oder Fahrlässigkeit, auch auf Schadenersatz in Anspruch genommen werden. Da der Verletzte in diesem Fall der Autor der Malware ist, wird er kaum auf seinem Anspruch bestehen, da ihm das kaum einen Vorteil, durch das Eingeständnis der Urheberschaft aber deutliche Nachteile einbringen dürfte.

Abgesehen davon dürfte selbst die Durchsetzung der Ansprüche kein großes Risiko darstellen, da ein zu ersetzender Schaden schwerlich nachzuweisen ist. Es könnte also höchstens untersagt werden, weitere Malware desselben Autors zu analysieren, aber da der Autor bei Malware üblicherweise nicht sofort zu erkennen ist, läuft dieses Verbot ins Leere.

Bei einer Sicherheitsanalyse sieht es bezüglich der Beseitigung der Beeinträchtigung ähnlich aus; nachdem die Analyse vorgenommen wurde, läßt sie sich schwerlich

[8] Bei Viren besteht hier leider das Risiko einer Infektion; außerdem treffen einige Viren bereits Gegenmaßnahmen gegen eine derartige Beobachtung.

ungeschehen machen, das Wissen um das Ergebnis bleibt vorhanden. Eine Unterlassung ist dagegen ernst zu nehmen, da bei neuen Programmversionen ggf. eine erneute Analyse notwendig wäre. Ein Anspruch auf Schadensersatz dürfte allenfalls entstehen, wenn man die Ergebnisse einer ungünstig ausgefallenen Analyse verbreitet und so den Ruf des Softwareherstellers schädigt; bei einer ausschließlich betriebsinternen Verwendung entsteht dem Hersteller kein Schaden.

### Strafrechtlich: § 106 UrhG. Unerlaubte Verwertung urheberrechtlich geschützter Werke

In Bezug auf Computerprogramme ist es nach dem Wortlaut des § 106 UrhG (siehe Anhang B.1) strafbar, wenn unzulässigerweise und ohne Einwilligung des Berechtigten ein Programm oder eine Bearbeitung oder Umgestaltung eines Programms vervielfältigt[9] wird. Im Umkehrschluß muß man daraus folgern, daß die bloße *Herstellung* einer Umgestaltung (also etwa eine Dekompilierung) *strafrechtlich* nicht relevant sein kann. Zu diskutieren ist allenfalls, inwieweit technisch gesehen bei einer Dekompilierung bereits eine Vervielfältigung[10] des Programmes stattfindet; dies hängt vom konkreten Vorgehen ab und dürfte sich bei entsprechender Umsicht vermeiden lassen.

### Kein öffentliches Interesse an einer Verfolgung

Laut § 109 UrhG wird eine Tat nach § 106 UrhG nur auf Antrag verfolgt, es sei denn, daß die Strafverfolgungsbehörde ein besonderes öffentliches Interesse sieht. Im allgemeinen kann bei Reverse Engineering wohl davon ausgegangen werden, daß dieses Interesse nicht besteht, da im Normalfall[11] nur der Urheber des untersuchten Programms geschädigt werden kann. Bei normaler Piraterie (d. h. Raubkopieren) sieht dies anders aus, da hierdurch auch Hersteller von Konkurrenzprodukten durch einen geringeren Umsatz geschädigt werden können.

Unterstützt wird dies auch von § 153 StPO, wonach die Staatsanwaltschaft unter bestimmten Umständen von der Verfolgung absehen kann, „wenn die Schuld des Täters als gering anzusehen wäre und kein öffentliches Interesse an der Verfolgung besteht."

### Rechtfertigung durch Notwehr und Notstand

Wenn eine Bedrohung durch Malware besteht, die durch die verfügbaren Antivirusprogramme nicht bekämpft werden kann, ist es natürlich nicht zumutbar, sich auf die zulässigen Verfahren zu beschränken, wenn dies eine Analyse verzögert oder sogar verhindert; aus diesem Grund sollen nun die zur Rechtfertigung in Frage kommenden Paragraphen betrachtet werden. Zu beachten ist allerdings, daß die mögliche

[9] Für Computerprogramme wird eine Vervielfältigung durch eine Verbreitung impliziert; die öffentliche Wiedergabe kommt technisch nicht in Betracht.

[10] In § 69c und § 69e UrhG wird allerdings eine Vervielfältigung des Codes von der Herstellung einer Umgestaltung deutlich unterschieden.

[11] siehe allerdings Kapitel 3.4.2.

Rechtfertigung nur gelten kann, solange noch keine Abhilfe allgemein verfügbar ist bzw. kein ausreichendes Analyseergebnis veröffentlicht wurde; eine Analyse zu reinen *Ausbildungs-* oder *Forschungszwecken* läßt sich hierdurch *nicht* rechtfertigen.

Die Anwendbarkeit des § 227 BGB „Notwehr“ bzw. des sinngleichen § 32 StGB „Notwehr“ scheitert vermutlich daran, daß der Angriff, gegen den sich die Notwehr richtet, nach den gängigen Rechtsprinzipien aktiv von einem Menschen ausgehen muß (so z. B. [9]). Das Ziel der Verteidigung ist zwar nicht auf den Verursacher beschränkt, da die Handlung lediglich durch Notwehr geboten sein muß, so daß bei einem indirekten Angriff durch ein Werkzeug auch gegen dieses Maßnahmen ergriffen werden können. Die Anwendbarkeit wird aber nur in dem Fall gegeben sein, in dem der Malware-Angriff gezielt erfolgt und nicht wie üblich ein Virus freigesetzt wird, der dann sein Unwesen treibt.

Für diesen Fall ist offenbar der § 228 BGB „Notstand“ gedacht, der lediglich eine durch eine fremde Sache drohende Gefahr fordert. Allerdings werden hier die zulässigen Verteidigungshandlungen auf die Beschädigung oder Zerstörung der betreffenden Sache eingeschränkt, was die möglicherweise erforderliche Verletzung des Urheberrechts natürlich nicht einschließt – allenfalls das ohnehin zulässige Löschen der Malware (siehe Fußnote in Kapitel 3.5.2) ließe sich hierdurch rechtfertigen.

Anlaß zur Hoffnung bietet aber der § 34 StGB „Rechtfertigender Notstand“, nach dem eine Tat nicht rechtswidrig ist, wenn sie zur Abwehr einer Gefahr notwendig ist und „das geschützte Interesse das beeinträchtigte wesentlich überwiegt“, was hier der Fall sein dürfte[12]. Auch wenn dieser Paragraph im Strafrecht angesiedelt ist, so läßt sich doch die Grundaussage auf das Zivilrecht übertragen, so daß auch Schutz vor sonstigen Ansprüchen bestehen dürfte.

## 3.6. Gesetzeslage und Rechtsprechung in anderen Staaten

### 3.6.1. USA

Bis zum Anfang der 90er Jahre blieb die Frage, ob das Dekompilieren eines Programms eine Urheberrechtsverletzung darstellt, in den USA ungeklärt. Da in den USA ein sog. fallbasiertes Recht („Case Law“) gilt, wurde dieser Punkt erst durch die Urteile in zwei Fällen [41, 43] festgelegt. Beide Gerichte stellten fest, daß Reverse Engineering in bestimmten Fällen zulässig sein kann („may constitute fair use“). Im folgenden sollen daher unter anderem diese beiden Fälle diskutiert werden. Das größte Problem ist hierbei, daß nicht allgemein festgelegt wurde, wann die Zulässigkeit besteht; nach dem neuen europäischen Recht wäre ein Reverse Engineering aber wahrscheinlich in beiden Fällen unzulässig[13] gewesen, so daß die US-Rechtsprechung hier offenbar weiter geht.

[12] Integrität der Daten bzw. Verfügbarkeit der Rechenanlage sind abzuwägen gegen das wirtschaftlich nicht relevante Urheberrecht an der Malware.

[13] Siehe hierzu die Erörterung der Problematik der Interoperabilität zwischen Software und Hardware in Kapitel 3.1.1.

### Sega Enterprises v. Accolade, Inc.

Der Videospielehersteller Accolade hatte, um Spiele für die Genesis-Spielekonsole von Sega herstellen zu können, den Objektcode der System-ROMs dieser Konsole disassembliert. Dies war nötig, da die Konsole einen bestimmten Sicherheitscode auf jedem Spielemodul erwartete und dieses sonst nicht akzeptierte. Dieser Sicherheitscode wurde dann von Accolade in eigene Spiele eingebaut, hatte aber ansonsten keine echte Funktionalität.

Nachdem Sega eine Klage wegen Urheberrechtsverletzung eingereicht hatte, entschied das Gericht [43], daß eine Disassemblierung zulässig sei („fair use“), wenn dies die einzige Möglichkeit ist, an die nicht urheberrechtsgeschützten Programmelemente heranzukommen und ein legitimes Interesse an diesen besteht[14]. Das Interesse, Zugang zu der Spielekonsole zu bekommen, sei legitim, und eine spezielle Lizenz von Sega sei unzumutbar gewesen.

### Atari Games Corp. v. Nintendo of America, Inc.

Dieser Fall [41] ist sehr ähnlich gelagert; diesmal wollte Atari Spielemodule für die Konsolen von Nintendo herstellen. Es bestand allerdings ein Unterschied dahingehend, daß zuerst Atari eine Klage unter anderem wegen unfairem Wettbewerbs einreichte, worauf dann Nintendo unter anderem wegen Urheberrechtsverletzung klagte.

Das Gericht vertrat zwar prinzipiell die gleiche Ansicht wie im ersten Fall, das Verfahren ging aber aus den folgenden zwei Gründen zugunsten von Nintendo aus:

- Ein zulässiges Reverse Engineering kann nur an einer legitimen Kopie stattfinden. Atari hatte den Code aber unter Vorspiegelung falscher Tatsachen von der Copyright-Behörde (Copyright Office) erhalten.

- In diesem Fall wurde nicht der Sicherheitscode kopiert, sondern ein Programmteil, das diesen generierte. Der Code selbst war zwar nicht geschützt; da es aber sehr viele Möglichkeiten gab, diesen zu erzeugen, stellte die direkte Kopie der konkreten Implementation eine Urheberrechtsverletzung dar.

### Lotus Development Corporation v. Borland International, Inc.

Dieser Fall soll hier exemplarisch für eine Übernahme von Elementen der Benutzeroberfläche eines Programms stehen. Die Firma Borland hatte in ihren Produkten Quattro und Quattro Pro 1.0 zusätzlich zu der normalen Bedienoberfläche eine Nachbildung der Menüstruktur von Lotus 1-2-3 eingebaut, die Umsteigern die Bedienung erleichtern sollte. Borland wurde daraufhin von Lotus wegen Urheberrechtsverletzung verklagt, und nachdem zuerst einige Entscheidungen zugunsten von Lotus fielen, ging das Verfahren letztendlich zugunsten von Borland aus, da die Menüstruktur ein Teil

[14] „... if such disassembly provides the only means of access to those elements of the code not protected by copyright and the copier has a legitimate reason for seeking such access.“ [43]

der Funktionsweise („method of operation“) sei und diese nach US-Recht explizit vom Urheberrecht ausgenommen ist.

Im Unterschied zu den beiden anderen Fällen ging es hier nicht um die *Methode* des Reverse Engineerings, da eine reine Beobachtung wohl auch im US-Recht nicht untersagt ist und die Menüs offen sichtbar waren. Die Rechtsprechung bezüglich der Schutzfähigkeit von Bedienelementen scheint allerdings uneinheitlich zu sein, wie sich bereits daran zeigt, daß der Klage von Lotus zunächst nachgegeben wurde; auch in der Urteilsbegründung [42] finden sich Gegenbeispiele.

#### Digital Millennium Copyright Act

Im Jahr 1998 wurde der Digital Millennium Copyright Act (DMCA) [31] verabschiedet, in dem unter anderem die Frage des Reverse Engineerings von Software angesprochen wird. Allerdings gestattet es der betreffende Abschnitt lediglich, unter gewissen Umständen zum Zwecke einer Analyse technische Maßnahmen zu umgehen, die den Zugang zu einem bestimmten Teil des Programms verhindern. Vorausgesetzt wird aber unter anderem, daß die Analyse selbst keine Rechtsverletzung darstellt, so daß sich durch den DMCA an der prinzipiellen Frage der Zulässigkeit von Reverse Engineering nichts ändert.

### 3.6.2. Australien

Im australischen Recht hat es 1999 eine interessante Änderung[15] des Urheberrechts gegeben. Vorher konnte der Urheber eine Dekompilierung komplett untersagen; nun ist eine Dekompilierung von Software und *Hardware* zu folgenden Zwecken erlaubt:

- Herstellung interoperabler Software
- Fehlerbehebung, einschließlich Y2K-Fehler
- Testen und Korrigieren der Sicherheit von Informationssystemen

Dies ist aber nur dann zulässig, wenn die nötigen Informationen, die durch die Dekompilierung gefunden wurden, nicht ohne weiteres verfügbar sind bzw. wenn keine fehlerfreie Version des Programms verfügbar ist.

Im ganzen scheint die australische Gesetzgebung der europäischen also recht ähnlich zu sein (auch die sonstigen Rahmenbedingungen ähneln sich, z. B. in Bezug auf Weitergabe der Informationen, Sicherheitskopie oder beobachtende Analyse). Daß eine Dekompilierung für die Fehler*behebung* zulässig ist, ist nach der EU-Richtlinie zumindest strittig; sie ist aber auch in Australien für die Fehler*suche* nur dann zulässig, wenn der Fehler bekannt ist und dieser eine bestimmungsgemäße Benutzung des Programms verhindert.

Der Punkt des Testens der Sicherheit ist natürlich sehr wichtig. Interessant ist hier hauptsächlich die retrospektive Zulässigkeit der Analyse; diese ist – wie oben

[15] Copyright Amendment (Computer Programs) Bill 1999 [32]

bereits erwähnt – nur dann erlaubt, wenn durch sie Informationen erlangt werden, die noch nicht ohne weiteres verfügbar waren. Dabei stellt sich natürlich die Frage, ob eine Analyse zulässig ist, die entweder keine Sicherheitslücken findet oder nur die bereits vom Hersteller dokumentierten; im letzteren Fall könnte man sie rechtfertigen, indem man die Sicherheitsmängel behebt (dieser Punkt wird im Copyright Amendment explizit und getrennt von der normalen Fehlerbehebung als zulässig aufgeführt). Zumindest ist aber gesichert, daß eine Malware-Analyse keine Verletzung darstellt.

### 3.6.3. Japan

Im japanischen Recht gibt es offenbar keine direkte Erwähnung der Dekompilierung. In den Gesetzeskommentaren herrscht aber die Meinung vor, daß diese nach dem Urheberrecht zulässig ist.

Während der Debatte um die EU-Direktive wurde ein Fall[16] zitiert, in dem der Angeklagte einen BASIC-Interpreter von Microsoft dekompiliert, kommentiert und die Ergebnisse als Buch veröffentlicht hatte. Der Klage wurde verständlicherweise stattgegeben, was damals so ausgelegt wurde, daß Reverse Engineering in Japan unzulässig sei. Dabei hat das Gericht den Schwerpunkt auf die Veröffentlichung des Quellcodes gelegt, und selbst der Anwalt von Microsoft betonte, daß es nicht primär um die Legalität des Reverse Engineerings ging. [27]

[16] Microsoft Corp v Shuuwa System Trading K.K.

# 4. Praktische Methoden

Im folgenden sollen die verschiedenen praktischen Methoden des Reverse Engineerings eingehend behandelt werden. Dabei wird versucht, die Themen nach steigender Komplexität der theoretischen Aspekte sowie der Werkzeuge zu ordnen.

Die Methoden unterscheiden sich weiterhin im Grad an Gewißheit bezüglich der dynamischen (Beobachtung, Tracing & Spying, Debugging) oder strukturellen (Disassemblierung, Dekompilierung) Eigenschaften sowie im Umfang des an der Software vorgenommenen Eingriffs.

## 4.1. Beobachtende Analyse

Wie es der Begriff bereits ausdrückt, beschränkt man sich bei der beobachtenden Analyse darauf, ein ablaufendes Programm ohne besondere Hilfsmittel zu untersuchen. Dies kann z. B. durch die Betrachtung der Bildschirmausgabe geschehen; besonders interessant ist aber im allgemeinen die Untersuchung der vom Programm erzeugten bzw. veränderten Dateien.

In den Einsatzzwecken ist diese Methode natürlich eingeschränkt; dennoch verbleiben einige Anwendungen, die nun betrachtet werden sollen. Bei der Betrachtung der Nützlichkeit dieser Methode müssen außerdem folgende Vorteile berücksichtigt werden:

- Es ist nur ein relativ geringer Aufwand nötig; dies gilt auch für die nötigen Werkzeuge.
- Die rechtliche Situation ist völlig unproblematisch; siehe Kapitel 3.3.1.
- Es sind außer dem gewöhnlichen Anwenderwissen kaum besondere Kenntnisse über das verwendete System nötig. Eine Ausnahme hiervon ist allerdings das Endianess[1]-Problem bei der Analyse von Datendateien.

### 4.1.1. Dokumentation

Wenn für ein Programm keine Dokumentation vorliegt, kann versucht werden, diese durch systematisches Testen zu erarbeiten. Dabei können natürlich keine absichtlich

[1] Betrifft die Anordnung von im Speicher abgelegten Datenworten. Die beiden Hauptvarianten sind „Little Endian“ (z. B. Intel 80x86, DEC Alpha) und „Big Endian“ (z. B. Motorola 68k, SUN SPARC, PowerPC). Eine exakte Definition ist an dieser Stelle nicht relevant und würde zu weit führen.

verborgenen Funktionen entdeckt werden; für Standardsoftware oder für Anwendungssoftware, deren Anwendungsdomäne bekannt ist, läßt sich aber durchaus eine ausreichende Dokumentation erstellen. Dies gilt insbesondere für Programmfehler, da diese auch bei der normalen Programmentwicklung hauptsächlich bei Tests festgestellt werden.

### 4.1.2. Analyse von Dateiformaten

Wenn das Ziel darin besteht, mit einem bereits existierenden Programm Daten auszutauschen, so kann dies häufig dadurch erreicht werden, daß das verwendete Dateiformat gelesen oder geschrieben werden kann. Einfache Dateiformate können – etwa durch die Eingabe von Testdaten in das ursprüngliche Programm und Betrachtung der erzeugten Dateien – analysiert werden, ohne dabei das Programm selbst zu untersuchen; bei komplexeren Dateiformaten, die z. B. auf einer hierarchischen Struktur basieren, ist diese Methode allerdings nur gangbar, wenn bereits eine partielle Dokumentation vorliegt.

### 4.1.3. Übernahme von Ideen

Die Übernahme von Ideen und Grundsätzen eines Programms, die zumindest im Urheberrecht explizit vom Schutz ausgenommen sind (siehe Kapitel 3.1.2, § 69a), ist zum Teil bereits durch eine beobachtende Analyse möglich. Dies gilt natürlich hauptsächlich für Programmelemente, deren Ablauf auf dem Bildschirm oder einem anderen Ausgabegerät sichtbar ist und somit verfolgt werden kann. Beispiele hierfür wären die Benutzeroberfläche und Menüführung, mit Einschränkungen auch spezielle Grafik- oder Soundroutinen.

Der Aufwand besteht hierbei weniger in der Analyse, d. h. also in der Beobachtung, sondern vielmehr in der Neuimplementation. Hier wird zum einen die eigene Programmierfähigkeit benötigt, aber auch die eigene Kreativität, um die nicht beobachtbaren Teile des Programms zu ergänzen.

Der einfachste Fall besteht also darin, daß eine bestehende Menüstruktur in das eigene Programm übernommen wird; die tatsächliche Programmierleistung ist hierbei recht gering. Ein Beispiel hierfür wäre der bereits in Kapitel 3.6.1 vorgestellte Fall *Lotus Development Corporation v. Borland International, Inc.* , in dem die Firma Borland in ihrer Tabellenkalkulation Quattro Pro zusätzlich eine Emulation der Menüs von Lotus 1-2-3 implementiert hatte, um Umsteiger als Kunden zu gewinnen.

Ein anderes Gebiet, in dem die Ausgabe von Routinen beobachtet werden kann, ist das der Grafik. Während es früher teilweise noch möglich war, den Aufbau einer Grafik zu verfolgen, sind die Rechner inzwischen hierfür meist zu schnell oder der Aufbau wird absichtlich versteckt. Aus einer fertigen Grafik können zwar nur bedingt Rückschlüsse auf den zugrundeliegenden Algorithmus gezogen werden, insbesondere animierte Grafik kann aber als Idee für eigene grafische Effekte dienen. Außerdem wird sichtbar, welche Effekte in Echtzeit machbar sind.

Etwas ähnliches gilt für Algorithmen, bei denen bekannt ist, was sie prinzipiell tun

(z. B. Suchen, Sortieren, Kompression, Verschlüsselung[2] ...). Im allgemeinen wird es zwar nicht möglich sein, die genaue Funktionsweise zu bestimmen, es lassen sich aber Aussagen über die Existenz und Komplexität bzw. über die optimale erzielbare Performanz eines Algorithmus gewinnen.

### 4.1.4. Differenzanalyse bei Viren

Eine im Bereich der Virenforschung verbreitete Methode ist die sogenannte Differenzanalyse. Dabei wird versucht, die replikativen Eigenschaften eines unbekannten Programms zu bestimmen, indem man dieses in einer Testumgebung ausführt und anschließend (bei speicherresidenten Viren auch später) die in der Testumgebung vorhandenen weiteren Programmdateien auf Veränderungen untersucht. Um diese Untersuchung zu erleichtern, benutzt man hierbei üblicherweise spezielle Opferdateien (sog. goat files), die durch einen Generator systematisch erzeugt werden können.

Dabei können verschiedene Eigenschaften des mutmaßlichen Virus bestimmt werden:

**Viralität:** Ist das Programm überhaupt ein Virus, d. h. infiziert es z. B. andere Programmdateien?

**Größe:** Um wieviel Byte verlängern sich infizierte Dateien (dies wird als Länge des Virus betrachtet) bzw. ist diese Zahl konstant? Es kommt auch vor, daß die Länge tatsächlich nicht verändert wird (hiermit ist keine vorgetäuschte falsche Länge im Verzeichniseintrag gemeint); dies kann durch das Überschreiben des ursprünglichen Programmcodes (überschreibende Viren) oder durch das Ausnutzen von nicht belegten Bereichen im Programm (sog. Cavity-Viren) geschehen.

**Funktionserhaltung:** Funktioniert das ursprüngliche Programm weiter? Die Funktionalität wird z. B. durch überschreibende Viren gestört, aber auch Fehler in der Implementation können dazu führen, daß nicht alle Programme erfolgreich infiziert werden.

**Rekursive Replikation:** Gelingt die Infizierung auch in mehreren Generationen nacheinander? Üblicherweise wird davon ausgegangen, daß nach zwei Generationen die Infizierung auch weiterhin gelingen wird.

**Infektionsstrategie:** Nach welchem Schema werden andere Programmdateien infiziert bzw. nicht infiziert? Dies kann vom Namen der Datei, vom Verzeichnis, von der Größe, vom Aufbau der Datei und anderen Faktoren abhängen. Bei speicherresidenten Viren kann es auch zeitliche Abhängigkeiten geben; dieser Aspekt kann jedoch besser durch zusätzliche Werkzeuge untersucht werden (siehe Kapitel 4.2).

[2] Für eine Verschlüsselung gilt dies natürlich nur dann, wenn das Verschlüsselungsverfahren bekannt ist und dieses durch verschiedene Algorithmen implementiert werden kann.

**Selbstschutz:** Versucht der Virus sich vor der Entdeckung zu verbergen? Durch Beobachtung können u. a. Polymorphie, Verschlüsselung und auch Vortäuschung einer falschen Dateilänge entdeckt werden. Echte Stealth-Eigenschaften, für die der Virus sich in Betriebssystemfunktionen einklinkt, können ebenfalls entdeckt werden, wenn das System sowohl mit im Hauptspeicher geladenem Virus als auch ohne diesen untersucht und die Beobachtungen verglichen werden.

## 4.2. Tracing & Spying

Mit Tracing wird im allgemeinen das Verfolgen des dynamischen Verhaltens eines Programms durch spezielle Werkzeuge bezeichnet. Dies muß nicht auf Maschinenbefehlsebene erfolgen, sondern es können auch Ereignisse auf einer höheren Abstraktionsebene aufgezeichnet werden. Beispiele für derartige Ereignisse wären Betriebssystem- oder Bibliotheksaufrufe, aber auch Zugriffe auf bestimmte Hardwarekomponenten.

Der Begriff Spying hat eine ähnliche Bedeutung; mit „Spionieren“ ist hier gemeint, daß man – wiederum durch spezielle Werkzeuge – Vorgänge und Daten in einem Programm beobachtet, die normalerweise nicht sichtbar sind. Der Unterschied zum Tracing besteht darin, daß nicht hauptsächlich der dynamische Ablauf beobachtet wird, sondern insbesondere statische Daten untersucht werden, etwa durch Speicherabzüge oder eine weitergehende softwaregestützte Analyse der gespeicherten Strukturen.

Die beiden Verfahren, insbesondere das Tracing, stellen bereits eine Vorstufe zum Debugging dar, da die Umgebung des untersuchten Programms verändert wird bzw. im Speicher befindliche Datenstrukturen betrachtet werden, die normalerweise nicht sichtbar sind. Sie haben gegenüber dem Debugging einige Vorteile:

- Die rechtliche Situation ist, wie bereits bei der reinen Beobachtung, völlig unproblematisch, da nicht der Programmcode selbst untersucht wird.
- Die Abstraktion von der Maschinenebene ist höher, da man sich zwar u. U. mit der Betriebssystemschnittstelle, nicht aber mit den Prozessorbefehlen selbst befaßt. Aus diesem Grund reichen Kenntnisse über das Betriebssystem aus; beispielsweise ist das unten beschriebene *strace* unter Linux auf verschiedenen Prozessorarchitekturen identisch zu bedienen, selbst auf anderen UNIX-Plattformen sind die Unterschiede nicht sehr groß.
- Man erhält schneller einen Überblick über die Schnittstellen zwischen einem Programm und seiner Umgebung, da nur diesbezügliche Informationen dargestellt werden, während programminterne Funktionen verborgen bleiben.

### 4.2.1. Software

In den meisten Fällen ist für diese Verfahren keine besondere Hardware nötig, so daß ein Einsatz von Softwarewerkzeugen ausreicht. Dies gilt insbesondere für den Bereich

des Spyings, also die Analyse von statischen Daten; hier ist zusätzliche Hardware nur dann nötig, wenn man nicht über ein Programm an den Speicherinhalt herankommt, etwa wenn bei einem eingebetteten System keine interaktive Schnittstelle zur Verfügung steht oder prinzipiell oder aus Speichermangel keine weitere Software in das System geladen werden kann.

Ein softwaregestütztes Tracing geschieht üblicherweise, indem das Analyse-Programm („Werkzeug“) sich zwischen das zu untersuchende Programm und das Betriebssystem setzt und System- oder Bibliotheksaufrufe protokolliert. Damit sich ein aussagekräftiges Protokoll ergibt, müssen dem Werkzeug allerdings die zu untersuchenden Aufrufe bekannt sein, damit die Aufrufparameter sinnvoll interpretiert werden können bzw. damit überhaupt erst bekannt ist, welche Parameter für jeden Aufruf aufzuzeichnen sind.

Die genaue technische Funktionsweise hängt natürlich von der Architektur des Prozessors und des Betriebssystems ab. Die für die Protokollierung von Bibliotheksaufrufen meist verwendete Methode ist aber systemunabhängig und soll daher hier kurz beschrieben werden:

> Für jede der Bibliotheksdateien, deren Aufrufe aufgezeichnet werden sollen, wird eine sogenannte „Wrapper“-Bibliothek[3] generiert, deren Schnittstelle exakt der der Original-Bibliothek entspricht. Die Wrapper-Bibliothek tut nun bei jedem Aufruf einer Funktion nichts weiter, als den Aufruf mit den dazugehörigen Parametern zu protokollieren und ihn an die Original-Bibliothek weiterzuleiten.

Ein „Hineinschauen“ in laufende Programme kann im einfachsten Fall durch einen Speichermonitor geschehen. Dies wird aber nur in den Fällen genügen, in denen der zu untersuchende Datenbereich (der Codebereich ist üblicherweise nicht variabel und daher nicht von Interesse) recht klein ist und lediglich nach Zeichenketten oder bestimmten Binärwerten gesucht wird.

Bei komplexeren Systemen ist es sinnvoller, über Funktionen des Betriebssystems oder des Grafiksystems Informationen über die vom Programm verwendeten Systemressourcen zu erhalten; dies können geöffnete Dateien oder Netzwerkverbindungen sein, aber auch Fenster, belegte Speicherbereiche und referenzierte Module. Natürlich können diese Informationen auch durch ein Tracing derjenigen Systemfunktionen erhalten werden, die das Programm braucht, um die Ressourcen anzufordern; dennoch bietet der andere Blickwinkel einige Vorteile:

- Während des Programmablaufs steht der aktuelle Zustand des Programms zur Verfügung und läßt sich zu den interessanten Zeitpunkten gezielt untersuchen.

[3] Als Wrapper-Bibliotheken werden Bibliotheken bezeichnet, die im wesentlichen eine Schnittstelle auf eine andere abbilden, aber nur relativ wenig eigene Funktionalität besitzen. In dem hier behandelten Fall sind die beiden Schnittstellen identisch. Eine Abbildung auf eine andere Schnittstelle wird z. B. durchgeführt, wenn die ursprüngliche Schnittstelle auf dem System nicht vorhanden ist, die benötigte Funktionalität aber nicht völlig neu entwickelt werden soll, sondern eine Anpassung auf eine bereits im System existierende Funktionalität genügt.

Ein Tracing hingegen liefert eine Ablaufhistorie, die erst noch nachvollzogen werden muß.

- Ein Spying ist an den Objekten orientiert; es lassen sich etwa alle Informationen zu einem bestimmten geöffneten Fenster erfragen. Bei einem Tracing können diese Informationen über viele Aufrufe hinweg verstreut sein.
- Beide Verfahren können sich ergänzen, wenn das Trace-Log bereits während des Programmablaufs ausgewertet wird; man kann beispielsweise erfahren, auf welches Fenster sich ein durchgeführter Aufruf bezieht.

Im folgenden sollen nun beispielhaft einige Werkzeuge beschrieben werden. Diese Auswahl erhebt natürlich nicht den Anspruch, repräsentativ zu sein, sondern soll lediglich die genannten Eigenschaften an mehreren Betriebssystemen und Benutzeroberflächen illustrieren.

### Beispiel: Intspy

Intspy ist ein Programm, das auf MS-DOS und kompatiblen Betriebssystemen läuft. Es zeichnet Betriebssystemaufrufe auf, die unter diesen Systemen als Software-Interrupts implementiert sind. Die Interpretation der Aufrufe wird in einer Konfigurationsdatei festgelegt, in der sich für jeden bekannten Aufruf eine Klartextbezeichnung findet, die mit Syntax-Angaben für die Parameter versehen ist.

Technisch wird dies recht einfach realisiert: Intspy wird mit dem Namen des zu untersuchenden Programmes als Parameter gestartet, außerdem muß noch angegeben werden, welche Kategorien von Aufrufen aufgezeichnet werden sollen. Die Kategorien lassen sich dabei in einer Konfigurationsdatei erstellen, vordefiniert sind u. a. die Kategorien für Datei- und Netzwerkoperationen. Das Werkzeug lenkt nun die zu untersuchenden Interrupt-Vektoren auf sich um und startet das Programm. Bei jedem nun erfolgenden Interrupt-Aufruf wird die entsprechende Routine von Intspy aktiv, schreibt den Aufruf (wenn er in der Konfigurationsdatei definiert ist) im Klartext in eine Protokoll-Datei und führt dann den ursprünglichen Betriebssystem-Interrupt aus.

### Beispiel: strace

Strace ist ein Programm, das für verschiedene UNIX-Derivate zur Verfügung steht, unter anderem SunOS, Solaris, Linux und Irix. Es läßt ein Programm unter seiner Kontrolle laufen und zeichnet die von diesem gemachten Systemaufrufe sowie die an das Programm geschickten Signale auf; dabei kann die Ausgabe auf dem Bildschirm oder in eine Datei erfolgen. Die Ausgabe erfolgt im Klartext mit einer passenden Interpretation der Parameter. Im Gegensatz zu Intspy werden auch die Rückgabewerte der Aufrufe ausgegeben.

Weitere Möglichkeiten von strace umfassen das Verfolgen von Tochterprozessen, Aufzeichnen der Position im Programm bei jedem Systemaufruf, Aufzeichnen von

Zeitangaben, Aufzeichnen nur bestimmter Systemaufrufe bzw. Kategorien von Systemaufrufen, nachträgliches Tracing bereits laufender Prozesse und Erstellung einer Aufrufstatistik.

### Beispiel: ZAPdb

ZAPdb OpenGL Interactive Debugger von IBM ist ein Werkzeug für Microsoft Windows 9x und Windows NT, mit dem die OpenGL[4]-Aufrufe eines Programms aufgezeichnet und untersucht werden können. ZAPdb besteht aus einem Hauptprogramm mit einer grafischen Oberfläche und einer Bibliothek („DLL“, Dynamic Link Library), die (nach dem oben erläuterten Prinzip der Wrapper-Bibliothek) von dem zu untersuchenden Programm aufgerufen wird und die Aufrufe zum einen an die Original-DLL, zum anderen an das Hauptprogramm weiterleitet. Die Funktionalität umfaßt außer einem einfachen Trace-Protokoll auch die folgenden Möglichkeiten:

- Statistik über die Programmaufrufe
- Auslassen bestimmter Aufrufe bei der Ausführung
- Verlangsamung des Programmablaufs durch eine Zeitlupenfunktion. Die Ausführungsgeschwindigkeit wird allerdings in jedem Fall durch den Protokoll-Overhead stark gebremst.
- Diverse Filterungsmöglichkeiten bezüglich der angezeigten Aufrufe.
- Setzen von Breakpoints, d. h. das Unterbrechen des Programms bei bestimmten Aufrufen.
- Generieren eines C-Quelltextes aus den protokollierten Aufrufen. Das erzeugte Programm kann später die OpenGL-Aufrufe in der exakten Konstellation wiederholen.

### Beispiel: WinSight

WinSight ist ein Werkzeug, das bei einigen Entwicklungsumgebungen der Firma Borland mitgeliefert wurde und unter Microsoft Windows 3.x läuft. Es liefert Informationen über die existierenden Fenster und registrierten Fensterklassen und protokolliert an die Fenster gerichtete Botschaften mit, wobei sich diese nach Fenster, Klasse und Botschaftstyp filtern lassen.

Es ist zwar eigentlich als Hilfsmittel für das Debugging eigener Programme beabsichtigt, läßt sich aber auch für ein Reverse Engineering einsetzen, um z. B. die Kommunikation zwischen Programmen zu beobachten. Dies gilt unter Microsoft Windows ganz besonders, da Programme immer durch ein oder mehrere Fenster repräsentiert werden, auch wenn diese nicht auf dem Bildschirm sichtbar sein müssen.

[4] Eine Grafikbibliothek, die zuerst auf Silicon Graphics Workstations existierte, inzwischen aber für sehr viele Systeme zur Verfügung steht. OpenGL stellt zur Zeit die meistverbreitete Möglichkeit für plattformunabhängige 3D-Grafik dar.

#### Beispiel: xwininfo und xprop

Hierbei handelt es sich um zwei Werkzeuge, die dem X Window System beiliegen. Sie dienen dazu, Informationen über die von einem X-Server verwalteten Fenster anzuzeigen, wobei die Ausgabe von xwininfo etwas umfangreicher und detaillierter ist. Xprop kennt zusätzlich noch einen „Spy"-Modus, in dem es nach Ausgabe der Informationen nicht terminiert, sondern die Eigenschaften des untersuchten Fensters weiter beobachtet und etwaige Änderungen meldet.

Diese beiden Programme bieten keine Möglichkeit, an die Fenster geschickte Ereignisse zu verfolgen; dies ist allerdings auch nicht nötig, da hierfür unter X die gewöhnlichen UNIX-Signale verwendet werden, die mit dem oben beschriebenen strace verfolgt werden können.

### 4.2.2. Hardware

Ein Tracing[5] mit einer reinen Softwarelösung besitzt natürlich gewisse Einschränkungen; dies sind unter anderem

- Verlangsamung des untersuchten Programms aufgrund des Protokolloverheads.
- Mögliche Störung der Funktion, welche durch die Verlangsamung oder die geänderte Systemumgebung bedingt sein kann.
- Anfälligkeit für im untersuchten Programm vorhandene Anti-Debugging-Maßnahmen [6]; dies gilt insbesondere für Malware, aber auch für sonstige Software, die etwa kopiergeschützt sein kann.

Da die für eine Debugging-Unterstützung sinnvolle bzw. verfügbare Hardware ebenfalls für ein Tracing verwendbar ist (schließlich besitzen Debugger meist eine integrierte Trace-Funktion), sollen die verschiedenen Möglichkeiten der Hardware-Unterstützung an dieser Stelle gesammelt dargestellt werden. Dies soll exemplarisch anhand der 80x86-Prozessorreihe von Intel erfolgen, da hier nicht zuletzt aufgrund der von IBM geschaffenen offenen Systemarchitektur verschiedene Lösungen existieren, die sich bereits im Ansatz unterscheiden.

Aufgeführt werden also zum einen die integrierten Trace- und Debug-Möglichkeiten der verschiedenen Prozessormodelle, zum anderen zusätzliche Hardware-Ergänzungen, die es u. a. erlauben, ein Programm ohne Seiteneffekte zu beobachten. Zum besseren Verständnis folgen nun einige Begriffsdefinitionen.

**Breakpoint:** Ein Zustand im Ablauf eines Programms, an dem die Ausführung unterbrochen und eine Service-Routine aufgerufen werden soll. Diese kann einen

[5] Auf Spying wird in diesem Zusammenhang nicht eingegangen, da hierfür, wie bereits in Kapitel 4.2.1 erläutert, nur in Ausnahmefällen eine Hardwareunterstützung nötig ist. Bei Workstations oder PCs dürfte dies normalerweise nicht der Fall sein.

[6] Diese werden in Kapitel 4.3 näher erläutert, da nicht alle Maßnahmen auch für ein Tracing störend sind.

bereits laufenden Debugger aktivieren, aber auch eine Aufzeichnung für ein Tracing durchführen. Ein Breakpoint muß nicht notwendigerweise aus einer Programmadresse bestehen (in diesem Fall würde er als Code-Breakpoint bezeichnet werden), sondern kann auch Bedingungen wie veränderte Speicherzellen beinhalten. Die Abfrage der auslösenden Bedingung kann zum Teil oder sogar vollständig in Software gelöst sein; bei vollständiger Hardware-Unterstützung hingegen wird genau dann ein Interrupt ausgelöst, wenn ein definierter Breakpoint auftritt. In diesem Fall muß dann nur noch festgestellt werden, um welchen von möglicherweise mehreren definierten Breakpoints es sich handelt.

**Interrupt:** (Genauer: Hardware-Interrupt). Eine Unterbrechung des normalen Programmablaufs, die durch die Hardware (z. B. den Prozessor selbst, wenn ein bestimmter Zustand erreicht wird, oder die Aktivierung einer Signalleitung durch ein externes Gerät) ausgelöst wird. Je nachdem welcher Interrupt Request (IRQ, es gibt auf den üblichen IBM-kompatiblen Systemen 15 verschiedene) ausgelöst wurde bzw. welcher Interrupt vom Prozessor aktiviert wurde, wird der diesem zugeordnete Interrupt-Handler angesprungen und ausgeführt; nach Beendigung dieser Routine läuft das Programm normal weiter.

### Interne Möglichkeiten des Intel 8086

Die integrierte Debug-Unterstützung des Intel 8086 ist relativ beschränkt. Abgesehen von Code-Breakpoints, die durch ein Patchen des Codes, d. h. das Ersetzen eines Code-Bytes durch einen Interrupt-Aufruf implementiert werden, besitzt dieser Prozessor nur noch einen Single-Step-Modus[7]. Wenn dieser aktiviert ist, wird nach jedem ausgeführten Befehl ein Debug-Interrupt ausgelöst.

Dieser Modus wird hauptsächlich von Debuggern genutzt, um es dem Anwender zu erlauben, das Programm Befehl für Befehl zu verfolgen. Wie man nach den vorangegangenen Erläuterungen sehen kann, läßt sich dieses Feature auch zur Erstellung eines Protokolls verwenden, indem die Interrupt-Routine jeden ausgeführten Befehl speichert. Außerdem kann über den Single-Step-Modus jede Breakpoint-Bedingung bzw. jede beliebige Filterung eines Tracing-Protokolls implementiert werden, wobei dann aber nach jedem Schritt die Bedingungen durch die Software überprüft werden müssen. Mit dieser Technik ist es natürlich ebenfalls möglich, einen Code-Breakpoint um zusätzliche Bedingungen zu ergänzen.

Wie man sieht, ermöglichen die Eigenschaften des 8086 u. U. durchaus sinnvolle Tracing- und Debuggingmöglichkeiten, allerdings sind diese mit verschiedenen Nachteilen verbunden:

- Code-Breakpoints sind nur möglich, wenn die entsprechenden Stellen im Code

[7] Unter Single-Step wird im folgenden verstanden, daß jeder Befehl einzeln ausgeführt wird (Single-Instruction) und nicht, daß nach jedem Prozessortakt unterbrochen wird (Single-Cycle), um selbst die einzelnen Ausführungsphasen untersuchen zu können. Letzeres ließe sich auch nicht im Prozessor realisieren, da die Ausführung eines Befehls nicht zur Ausführung anderer Befehle unterbrochen und später wieder fortgesetzt werden kann.

veränderbar sind (ROM-Code kann also nicht analysiert werden), es sich nicht um selbstmodifizierenden Code handelt (bei Malware häufig, z. B. Verschlüsselung) und das Programm seine Integrität nicht überprüft (häufige Kopierschutzmaßnahme, die ein Cracken verhindern soll).

- Beim Tracing im Single-Step-Modus gibt es einen großen Overhead für die Interrupt-Behandlung, die Protokollierung und ggf. für die Filterung. Es muß davon ausgegangen werden, daß für jeden im untersuchten Programm selbst ausgeführten Maschinenbefehl mindestens 10, in praxisrelevanten Fällen wohl eher 100 oder mehr weitere Befehle nötig sind, so daß sich die Ausführungszeit um den entsprechenden Faktor verlangsamt.

- Die Verlangsamung beim Single-Step-Modus ist nicht nur lästig, sondern kann auch die Ausführung des Programms behindern, wenn es entweder zeitkritische Routinen gibt oder der Programmierer absichtlich eine Beobachtung verhindern wollte.

- Der Single-Step-Modus kann ggf. vom Programm erkannt oder sabotiert werden, etwa wenn es ihn selbst verwenden will (dies macht normalerweise bei Anwendungsprogrammen keinen Sinn).

### Zusatzhardware: Periscope Debugging Board

Um die genannten Beschränkungen zu umgehen, wurde von Drittanbietern Zusatzhardware entwickelt, die weitergehende Untersuchungsmöglichkeiten zur Verfügung stellte. In diesem Umfeld wurde ebenfalls allgemein die Debugging-Unterstützung betont, da der Anwendungsschwerpunkt in der Fehlersuche in eigenen Programmen und weniger im Reverse Engineering lag.

Als Beispiel hierfür sollen die Möglichkeiten des von der (inzwischen nicht mehr existierenden) Firma „The Periscope Company, Inc." hergestellten Periscope Model IV vorgestellt werden. Dabei stehen die theoretischen Möglichkeiten der Hardware im Vordergrund. Auf die Unterstützung durch die mitgelieferte Debugger-Software wird an dieser Stelle nicht eingegangen.

Es handelt sich bei dem Model IV um eine ISA-Einsteckkarte, die über ein Kabel mit einem sogenannten „Pod"[8] verbunden ist, das zwischen die CPU und den CPU-Sockel gesteckt wird und die Prozessorsignale weiterleitet. Auf diese Weise kann die Elektronik auf der Platine alle Bus-Transaktionen der CPU überwachen und aufzeichnen. Es ist auch möglich, die beiden Komponenten in zwei getrennten Systemen zu installieren, um so das beobachtete System nicht zu beeinflussen.

Das Model IV bietet folgende Möglichkeiten:

- Definition von verschiedenen Ereignissen wie etwa Speicher- oder I/O-Transaktionen an bestimmten Adressen oder das Auftreten bestimmter Datenwerte oder Bitmasken; beide Möglichkeiten können auch kombiniert werden.

[8] Prozessorspezifisch, verfügbar für Intel 80286/386/486

- Breakpoint: Durch ein definiertes Ereignis kann ein Interrupt[9] ausgelöst werden, der dann den Debugger aktiviert.

- Sequentieller Trigger: Die Karte implementiert einen Automaten mit mehreren Zuständen, wobei die Zustandsübergänge durch definierte Ereignisse ausgelöst werden. Es können hiermit kombinierte Ereignisse definiert werden.

- Aufzeichnung: Auf der Karte befindet sich ein Ringpuffer, der ständig alle Zustände der Prozessorpins aufzeichnet und so ein Trace-Protokoll führt. Der Puffer kann durch ein definiertes Ereignis eingefroren werden, wobei noch die Wahl besteht, ob die Aufzeichnung sofort beendet werden soll oder erst, wenn der Puffer schon wieder zur Hälfte oder vollständig mit neuen Zuständen gefüllt ist. Hierdurch können wahlweise Zustände vor oder nach dem gewählten Ereignis aufbewahrt werden. Nach dem Beenden der Aufzeichnung wird ggf. der Debugger aktiviert; natürlich wird bei Aktivierung des Debuggers in jedem Fall der Puffer eingefroren. Ein gefüllter Puffer kann ebenfalls ein Ereignis darstellen, d. h. das untersuchte Programm läuft so lange, bis kein Platz mehr für die Speicherung weiterer Ereignisse ist.

- Selektive Aufzeichnung: Es werden nur die durch ein definiertes Ereignis bestimmten Zustände aufgezeichnet. Es ist dadurch z. B. möglich, den Inhalt einer Speicherzelle zu verfolgen oder I/O-Zugriffe auf ein Gerät zu protokollieren. Die Kapazität des Puffers wird gegenüber einer nachträglichen Filterung durch die Software wesentlich erhöht; dies ist insbesondere bei Echtzeitanforderungen wichtig, da der Puffer ohne selektive Aufzeichnung spätestens nach einigen Millisekunden voll ist.

Die Stärke dieser Lösung liegt eindeutig darin, daß das untersuchte Programm nicht oder nur so wenig wie möglich gestört wird. Dies wird durch die prozessorunabhängige Aufzeichnung erreicht, die ein Tracing in Echtzeit ermöglicht und wenn nötig auf einem Zweitsystem durchgeführt werden kann. Die normale Breakpoint-Funktion der Karte ist sicher ebenfalls nützlich; allerdings ist es hier von Nachteil, daß es sich um eine externe Lösung handelt, da ein Interrupt immer erst mit einer gewissen Verzögerung *nach* dem auslösenden Ereignis zum Tragen kommt; außerdem werden Instruktionen von der CPU meist einige Zeit vor ihrer Ausführung aus dem Speicher geholt („Prefetch“), was zu einer frühzeitigen oder unnötigen Auslösung von Code-Breakpoints führen kann. Weiterhin ist es beim i486 nötig, den prozessorinternen Cache auszuschalten, da sonst nicht alle Transaktionen auf dem Prozessorbus reflektiert werden und somit nicht von der Karte beobachtet werden können.

### Zusätzliche Möglichkeiten des Intel 80386

Nachdem zum einen das Interesse an einer besseren Debugging-Unterstützung durch die Hardware immer größer wurde und zum anderen durch die verbesserte Ferti-

[9] In diesem Fall der Non Maskable Interrupt (NMI)

gungstechnologie wesentlich mehr Funktionen im Prozessor integriert werden konnten, versah Intel die 80386-CPU [20] mit speziellen Debugging-Features. Außerdem lassen sich einige der Eigenschaften, die sich durch die diversen Schutzmechanismen des Prozessors ergeben, ebenfalls für ein Debugging einsetzen.

Der Intel 80386 bietet folgende Möglichkeiten:

**Code- und Daten-Breakpoints:** Durch die Programmierung von speziellen Debug-Registern können bis zu 4 verschiedene Breakpoints definiert werden, bei denen ein Debug-Interrupt ausgelöst werden soll. Ein Breakpoint kann entweder auf das Erreichen einer bestimmten Code-Adresse, das Schreiben von Daten an einer bestimmten Adresse oder das Lesen *und* Schreiben an einer bestimmten Adresse reagieren. Die Daten-Breakpoints erstrecken sich jeweils über 1, 2 oder 4 Bytes, die Adressen müssen dabei der Wortgröße entsprechend ausgerichtet sein.

Folgende Effekte sind zu beachten: Ein Code-Breakpoint wird ausgelöst, *bevor* der betreffende Befehl ausgeführt wird, ein Daten-Breakpoint hingegen erst *nach* dem Zugriff auf die entsprechende Adresse. Außerdem werden Daten-Breakpoints aufgrund des prozessorinternen Pipelinings erst verspätet ausgelöst. Dieses Verhalten kann zwar durch das Setzen eines Flags in einem Kontrollregister korrigiert werden, allerdings wird dadurch der Prozessor gebremst, d. h. ein Befehl wird erst ausgeführt, wenn der vorige abgeschlossen ist. Wie man sich leicht vorstellen kann, ist dieser Effekt bei modernen superskalaren Prozessoren noch ausgeprägter.

**I/O-Breakpoints:** Die diversen Schutzmechanismen des 80386 erlauben es u. a., den Zugriff einzelner Tasks[10] auf die Hardware zu kontrollieren, d. h. der Zugriff kann ganz verboten oder auf bestimmte I/O-Ports eingeschränkt werden. Beim Auftreten eines verbotenen Zugriffs wird ein entsprechender Interrupt ausgelöst. Der hauptsächliche Sinn dieser Maßnahme besteht darin, eine Virtualisierung der Hardware durch das Betriebssystem zu ermöglichen, aber natürlich lassen sich hiermit ebenfalls Breakpoints implementieren.

**Task-Breakpoints:** Für jede Task läßt sich festlegen, ob bei ihrer Aktivierung ein Debug-Interrupt ausgelöst werden soll.

**Resume-Flag:** Bei der Fortsetzung eines Programms nach einem aufgetretenen Code-Breakpoint wird automatisch dafür gesorgt, daß der betreffende Befehl nicht sofort wieder einen Breakpoint auslöst.

**Paging:** Der 80386 unterstützt eine Verwaltung des Hauptspeichers durch eine Aufteilung in 4 KB große „Seiten“. Dieses Verfahren dient hauptsächlich der virtuellen Speicherverwaltung, indem bei Zugriff auf nicht im Speicher befindliche

[10] Eine Task wird auf dem Intel 80386 in Hardware implementiert und kann einen Prozeß oder einen Thread darstellen.

Seiten ein Interrupt ausgelöst wird. Es läßt sich aber ebenfalls für die Implementation von Daten-Breakpoints nutzen, wobei die Granularität von 4 KB ggf. durch passende Abfragen im Interrupt-Handler verfeinert werden muß. Ebenso kann hiermit natürlich der Zugriff auf Geräte überwacht werden, die mit sog. Memory-Mapped-I/O[11] arbeiten.

**Virtueller 8086-Modus:** Wenn der Prozessor sich im Protected Mode[12] befindet, können normalerweise Programme nicht direkt ausgeführt werden, die für den 8086 bzw. den Real Mode[13] späterer Prozessoren geschrieben wurden. Der virtuelle 8086-Modus erlaubt es, diese unter Kontrolle eines Monitors trotzdem auszuführen. Damit eine komplette Systememulation zur Verfügung gestellt werden kann, können alle Software- und Hardware-Interrupts gefangen werden. Außerdem stehen die bereits vorgestellten Möglichkeiten der I/O-Zugriffskontrolle und des Pagings weiterhin zur Verfügung.

Dieses Verfahren wird, außer natürlich für die Emulation von MS-DOS unter Protected Mode-Betriebssystemen, hauptsächlich von Debuggern eingesetzt. Hiermit können zwar nur Real Mode-Programme untersucht werden, es wird aber verhindert, daß das untersuchte Programm den Debugger sabotiert, da zum einen dessen Speicher vor Zugriffen geschützt werden kann, zum anderen Änderungen an den Debugging-Interrupt-Vektoren die Ausführung des Debuggers nicht mehr beeinträchtigen. Denkbar wäre auch, mit dieser Methode ein Tracing von Interrupts durchzuführen, was den Vorteil hätte, daß das Tracing-Werkzeug nicht durch Manipulation der Interrupt-Vektoren abgehängt werden kann. Weiterhin werden erst durch den virtuellen 8086-Modus die Möglichkeiten von I/O-Breakpoints und Paging für die Untersuchung von Real Mode-Programmen nutzbar gemacht.

Die erweiterten Debug-Möglichkeiten des 80386 stellen gegenüber dem 8086 einen großen Fortschritt dar. Besonders innovativ sind dabei die Daten- und I/O-Breakpoints, aber auch die Code-Breakpoints können nützlich sein, da sie die bereits genannten Nachteile des 8086 aufheben; lediglich die Beschränkung auf 4 Code-Breakpoints stellt gegenüber dem Patchen des Codes einen Nachteil dar.

Die beste Unterstützung eines reinen Tracings bieten die I/O-Breakpoints, die auch keinen quantitativen Einschränkungen unterliegen. Code- und Daten-Breakpoints hingegen sind besser für ein Debugging oder ein Tracing innerhalb eines Debuggers zu gebrauchen, da normalerweise keine programminternen Adressen bekannt sind.

Im Vergleich mit einer externen Hardware wie der Periscope-Lösung zeigen sich Vor- und Nachteile: Ein Tracing mit dem 80386 ist nur über Breakpoints und eine

[11] Die Geräte werden nicht über einen getrennten I/O-Adreßraum angesprochen, sondern über normale Speicherzugriffe.
[12] Der Modus, in dem die diversen Schutzmechanismen aktiv sind.
[13] 8086-Kompatibilitätsmodus, in dem sich alle 80x86-Prozessoren nach dem Systemstart befinden.

softwaregestützte Protokollierung möglich, wobei aber aufgrund der Ereignisüberwachung durch die Hardware relativ wenig Overhead entsteht; dennoch kann es bei zeitkritischem Code zu Problemen kommen, insbesondere dann, wenn die Ereignisse stellenweise gehäuft auftreten, was z. B. bei Hardware-Zugriffen häufig der Fall ist. Die reine Breakpoint-Funktion hingegen ist im Prozessor selbst besser aufgehoben, da dann die bei einer externen Lösung problematische Verzögerung nicht auftritt; außerdem muß der ab dem i486 vorhandene prozessorinterne Cache nicht abgeschaltet werden.

### Zusätzliche Möglichkeiten des Intel Pentium: Performance Monitoring

Die eigentliche Debugging-Unterstützung der 80x86-Prozessoren wurde bei den Prozessorgenerationen nach dem 80386 kaum weiter verbessert oder erweitert. Die einzige wesentliche Neuerung, die allerdings nicht für ein Debugging und nur bedingt für ein Tracing geeignet ist, wurde mit dem Intel Pentium in Form des Performance Monitoring [35] eingeführt. Mit dieser Technik ist es möglich, verschiedene Ereignisse, die beim Ausführen von Code im Prozessor auftreten, zu zählen.

Im Gegensatz zu den bisher aufgeführten Debugging-Features, die bei späteren Prozessorgenerationen weiterhin zur Verfügung standen, existiert das Performance Monitoring des Pentium nur noch im Pentium MMX und im Cyrix/IBM 6x86MX. Andere Prozessoren wie etwa der Pentium Pro erlauben zwar oft ebenfalls ein Performance Monitoring, da dieses aber an die Funktionseinheiten der CPU angepaßt sein muß, ist es bei einer grundlegend verschiedenen Prozessorarchitektur kaum kompatibel zu halten; es erlaubt jedoch grundsätzlich ähnliche Beobachtungen.

Die Ereignisse, die gezählt werden können, sind im einzelnen:

- Gleitkomma-Befehle: Alle Befehle an die FPU (floating point unit). Langwierige und komplizierte Befehle werden teilweise mehrfach gezählt.
- Sprungbefehle (bedingte und unbedingte)
- Cache-Fehlzugriffe beim Lesen von Daten
- Cache-Fehlzugriffe beim Schreiben von Daten
- Cache-Fehlzugriffe beim Lesen von Code
- Befehle in den beiden Pipelines. Läßt Rückschlüsse über die Parallelisierung bei der Befehlsausführung zu.
- MMX-Befehle. Läßt erkennen, ob und wann ein Programm MMX einsetzt.
- MMX-Gleitkomma-Wechsel. Aufgrund der Prozessorarchitektur können MMX- und Gleitkomma-Befehle nicht gleichzeitig eingesetzt werden; die nötige Umschaltung braucht beim Pentium recht lange und sollte daher möglichst vermieden werden.

- Treffer bei der Sprungvorhersage
- Fehlzugriffe im TLB (Translation Look-aside-Buffer)
- Zurückschreiben von Datencache-Lines
- Probleme bei der Wortausrichtung (Alignment). Nicht passend ausgerichtete Datenworte sind zwar in der 80x86-Reihe zulässig (im Gegensatz zu Motorola 680x0 oder Sun SPARC), können aber zu Verzögerungen führen.
- Locked Bus-Zyklen

Wie man an dieser Liste – ohne die verschiedenen Ereignisse im einzelnen zu behandeln – sieht, eignet sich das Performance Monitoring sehr gut für das Optimieren von Code, insbesondere für einen spezifischen Prozessor, da Flaschenhälse bei der Programmausführung erkannt werden können. Es lassen sich aber auch Erkenntnisse über ein fremdes Programm gewinnen, etwa, ob und wann und wieviel dieses MMX- oder Gleitkomma-Befehle einsetzt, warum es auf dem einen Prozessor langsamer läuft als auf dem anderen, oder wie effizient es allgemein programmiert ist.

Da dieses Feature nur selten verwendet wird, sei hier noch auf das in [35] beschriebene ctpaul (c't Performance Monitor Utility) hingewiesen, das mit den genannten Prozessoren unter Windows 9x und NT eine entsprechende Messung für das gesamte System durchführen und grafisch darstellen kann.

#### Weitere Möglichkeiten: Schnittstellentracing

Zusätzlich zu den bis jetzt genannten Verfahren zur Beobachtung der CPU gibt es natürlich diverse Möglichkeiten, an den verschiedenen Systembussen und Schnittstellen anzusetzen und dort die Signale zu verfolgen. Dies kann sinnvoll sein, wenn in erster Linie die Kommunikation mit bzw. zwischen Geräten verfolgt werden soll und dies nicht durch eine Beobachtung (wie etwa in Kapitel 3.4.1 für das ZIP-Laufwerk beschrieben) des Prozessors zu erreichen ist.

Hauptsächlich sind dabei folgende Hardware-Hilfsmittel zu unterscheiden:

**Zusätzlicher Rechner:** Bei symmetrisch ausgerichteten Schnittstellen, die Ein- und Ausgabe nach demselben Protokoll verarbeiten, kann zwischen einen Rechner und ein daran angeschlossenes Gerät ein zusätzlicher Rechner geschaltet werden, der die Kommunikation beobachtet. Dies kann zum einen dadurch geschehen, daß der Rechner über zwei Schnittstellen desselben Typs verfügt und die Daten aktiv weiterleitet und dabei protokolliert (dieses Verfahren ist bei der seriellen Schnittstelle nach RS-232 möglich), zum anderen kann er mit einer Schnittstelle an einen Kommunikationsbus angeschlossen werden und die Daten abhören („Sniffing“, häufig bei Ethernet praktiziert).

**Spezial-Hardware:** Für viele verbreitete Schnittstellen existieren spezielle Geräte, die meist für die Fehleranalyse auf physikalischer Ebene (RS-232, Ethernet) gedacht sind, aber auch Protokollfunktionen bieten können. Nach einem Ausdruck oder einer Übertragung des Protokolls auf einen weiteren Rechner kann eine Analyse vorgenommen werden.

Dieses Verfahren bietet sich auch für die Protokollierung von Bus-Zugriffen an; in diesem Fall ist das Diagnosegerät meist als Einsteckkarte für den betreffenden Steckplatz implementiert (Ein Beispiel hierfür wäre die oben beschriebene Periscope-Hardware). Wenn die Abfrage der Daten dieser Karte über denselben Bus erfolgen kann, ist dabei keine weitere externe Hardware nötig; allerdings muß die Diagnosesoftware dann auf dem untersuchten System laufen.

**Vielzweck-Logikanalysatoren:** Wenn kein Gerät zur Verfügung steht, das die zu untersuchende Schnittstelle direkt unterstützt, muß auf einen Logikanalysator ausgewichen werden. Dies wird normalerweise der letzte Ausweg sein, da hierbei zum einen der passende physikalische Anschluß meist erst hergestellt werden muß und zum anderen das logische Schnittstellenprofil[14] zu definieren ist. Andere Gründe für die Verwendung eines Logikanalysators bestehen dann, wenn etwa Timing- oder andere Abweichungen festgestellt werden sollen oder wenn die aktuell untersuchte Schnittstelle inkompatibel zum vorgesehenen Standard ist.

Die Vorteile liegen also hauptsächlich darin, daß die Untersuchung für eine spezifische Schnittstelle unabhängig vom jeweiligen System und ohne dieses zu stören durchgeführt werden kann. Der Einsatz von Zusatzhardware ist aber aufgrund des nicht unerheblichen Aufwands nur dann sinnvoll, wenn das Ziel anders nicht oder nur schwer erreicht werden kann. Mögliche Gründe wären folgende:

- Das beobachtete System kann (z. B. Embedded Systems), darf (z. B. aufgrund von Sicherheitsbedenken) oder soll (z. B. aufgrund von Seiteneffekten der Beobachtung) nicht verändert werden.
- Das System selbst bietet keine ausreichende Tracing-Unterstützung.
- Es steht für die Plattform kein passendes Software-Werkzeug zur Verfügung.
- Es soll die Kommunikation zwischen mehreren Geräten beobachtet werden; dies ist in erster Linie bei Netzwerken der Fall. Selbst wenn bei einer Untersuchung in einem Netzwerk immer nur die Kommunikation zwischen jeweils zwei Geräten beobachtet werden soll, kommt eine vorhandene Heterogenität erschwerend hinzu, da in diesem Fall bei einem systeminternen (in diesem Zusammenhang auch host-basiert genannten) Tracing mehrere Plattformen unterstützt werden müssen.

[14] D. h. die Benennung der Signale, die Klassifizierung als Daten-, Steuer- und Taktleitungen, die Zusammenfassung von Datenleitungen zu Worten, Semantik inklusive Zeitverhalten etc.

## 4.3. Debugging

Debugging ist die dynamische Untersuchungsmethode, die potentiell am meisten in das untersuchte Programm eingreift. Dies kann durch die folgenden Debugging-Techniken geschehen:

- Der Programmfluß kann durch Maßnahmen geändert werden, die nicht der normalen Bedienung entsprechen. Dies ist durch direkte Manipulation der Position im Programm und durch Veränderung von Variablen möglich.
- Das Programm wird in den meisten Fällen partiell disassembliert. Wenn der Quellcode vorhanden ist, kann zwar darauf verzichtet werden, allerdings wird dann auch nur selten ein Reverse Engineering durchgeführt werden. Eine Ausnahme tritt allerdings bei der Überprüfung eines Compilers auf, bei der das erzeugte Kompilat untersucht wird.
- Da Debugging der Fehlerbehebung dient, kann das Programm temporär modifiziert werden, um eine Korrektur zu testen. In manchen Fällen kann es auch interessant sein, Korrekturen permanent abzuspeichern, was manche Debugger erlauben. Dies gilt insbesondere beim Reverse Engineering, da man es normalerweise vorziehen wird, den Quellcode zu ändern.
- Das Programm kann durch Breakpoints im Ablauf unterbrochen werden, um den Zustand zu den jeweiligen Zeitpunkten untersuchen zu können. Dies stellt aber noch keine Änderung des Programmflusses dar. Es besteht ein deutlicher Unterschied zur Verwendung von Breakpoints beim Tracing, wo diese lediglich als Hilfsmittel zur Protokollierung dienen.
- Durch das Single-Step-Verfahren kann ein Teil des Programms schrittweise durchlaufen werden, um das Verhalten detailliert nachvollziehen zu können.

Wie ein Debugger eingesetzt wird, hängt natürlich zum Teil von dessen Möglichkeiten ab, wobei die oben genannten Verfahren bis auf die Anzeige des Quellcodes von fast allen Debuggern unterstützt werden. Der Funktionsumfang umfaßt, je nach der bereits in Kapitel 4.2.2 beschriebenen verfügbaren Hardwareunterstützung, eher noch mehr als die genannten Möglichkeiten, es bestehen aber Unterschiede im Bedienungskomfort.

Die Vorgehensweise unterscheidet sich aber auch je nach Zweck des Reverse Engineerings. Aufgrund seiner Universalität gibt es eigentlich keine diesbezügliche Aufgabe, deren Lösung mit einem Debugger nicht prinzipiell möglich wäre; auch wenn er nicht das Mittel der Wahl darstellt, so ist er oft leicht verfügbar und relativ ausgereift.

Da die Verwendung eines Debuggers zur Fehlersuche dessen ursprünglicher Zweck ist, kommen hierbei auch alle Features zum Einsatz. Wie bereits oben erwähnt, können Fehlerkorrekturen auch getestet und ggf. permanent gemacht werden. Eine genauere Anleitung zum Einsatz von Debuggern würde hier zu weit führen und wird bereits in den Anleitungen zu den meisten Debuggern gegeben. Daher sei an

dieser Stelle auf die diversen verfügbaren Tutorien-Texte (z. B. [13, 26]) verwiesen. Die genannten Texte beziehen sich allerdings größtenteils auf die Fehlerbeseitigung in eigenen Programmen, da sich Alternativen mit direktem Bezug auf Reverse Engineering leider meist mit dem „Cracken“ kopiergeschützter Software oder anderen illegalen Zielen beschäftigen.

Auf dieselbe Art muß auch bei der Überprüfung von Compilerergebnissen vorgegangen werden, wenn es um einen konkreten Verdacht geht; ansonsten muß eine Disassemblierung vorgenommen werden. Zu betonen ist hierbei, daß das Ziel primär nicht das Finden eines Fehlers im Kompilat ist, sondern die Isolierung des verursachenden Fehlers im Compiler selbst.

Bei der Analyse zum Ermitteln der Ideen, Grundsätze und Schnittstellen wird man auf die Modifikation des Programmcodes verzichten können. Eine Disassemblierung wird natürlich benötigt und wäre vielleicht auch ausreichend, es ist aber hilfreich, interessante Stellen im Single-Step-Verfahren zu untersuchen, da Algorithmen hierdurch oft verständlicher werden. Eine Änderung des Programmflusses und Breakpoints sind dabei nützlich, um diese Stellen zu erreichen und Testfälle zu konstruieren.

Für die Portierung eines Programms – falls dies überhaupt per Hand versucht werden sollte – wird im allgemeinen eine Disassemblierung genügen, da zum einen das ganze Programm betrachtet werden muß, so daß eine Isolierung einzelner Programmteile keinen Nutzen hat, zum anderen ein echtes Verständnis für eine reine $1:1$ Umsetzung nicht unbedingt nötig ist.

Auf eine Disassemblierung wird man sich auch häufig bei Malware beschränken, da mit der Ausführung von bösartigem Code natürlich Risiken verbunden sind. Da man aber zum einen diese Risiken durch gekapselte Testsysteme in den Griff bekommen kann, zum anderen gerade Viren oft verschlüsselt sind oder eine Disassemblierung anderweitig erschweren, erscheint eine dynamische Untersuchung möglich und vielfach auch sinnvoll.

Diese wird bei den meist recht kleinen Viren in einem Single-Step-Durchgang bestehen, wobei erkennbar bösartige Funktionen, deren Wirkungsweise sofort ersichtlich ist und deren Fehlen den weiteren Programmablauf nicht beeinträchtigt, übersprungen werden können. Zusätzliche Debugging-Techniken wird man bei Viren benötigen, die im Speicher resident verbleiben oder die Anti-Debugging-Maßnahmen verwenden. Eine komplette Disassemblierung wird aber auch durch eine Untersuchung des dynamischen Verhaltens nicht verzichtbar, da nicht bei jedem Programmlauf auch alle Funktionen verwendet werden, was insbesondere für getriggerte Schadensfunktionen gilt.

Der letzte Punkt gilt ebenfalls für die meisten Trojaner; statt einer Disassemblierung können natürlich auch weitere Debugging-Techniken eingesetzt werden, um gezielt bestimmte Programmteile zu untersuchen. Eine Disassemblierung (dies gilt verstärkt bei einer Dekompilierung) ist aber meist einfacher als bei Viren, da Trojaner selten verschlüsselt oder anderweitig geschützt sind und häufig in höheren Programmiersprachen geschrieben werden, während für Viren oft Assembler verwendet wird. Dies trägt auch dazu bei, daß Trojaner meist deutlich mehr Code enthalten, was

ebenfalls gegen ein Debugging spricht; dieses ist nur dann zu empfehlen, wenn die Semantik des Codes anders nicht erschlossen werden kann.

Bei einer Sicherheitsanalyse von „normaler“ Software kommt es sehr auf die Zielsetzung an: Wird bereits ein bestimmter Sicherheitsmangel vermutet – sei dieser nun vom Autor beabsichtigt oder nicht – so kann er mit den Methoden der Fehlersuche gefunden werden. Eine Fehlerbehebung dürfte dabei aber nicht im Vordergrund stehen und nur provisorisch vorgenommen werden. Sie ist zwar rechtlich zulässig (siehe Kapitel 3.1.2, § 69d), dennoch wird man es im allgemeinen vorziehen, den Urheber zu kontaktieren. Wenn der Sicherheitsmangel allerdings absichtlich in Form einer Schadensfunktion oder Hintertür in das Programm eingebaut wurde, ist natürlich nicht mit einer Unterstützung durch den Urheber zu rechnen, und eine Fehlerkorrektur kann in Erwägung gezogen werden, obwohl in diesem Fall ein anderes Produkt vorzuziehen wäre.

Wenn noch kein konkreter Verdacht besteht, ist für eine vollständige Analyse eigentlich eine Disassemblierung nötig, die aber bei komplexeren Anwendungen an der Codegröße scheitern kann[15]. Ein gezieltes Debugging kann zwar unternommen werden, stößt aber auf dasselbe Problem. Hier kann man bei der Suche nach beabsichtigten verborgenen Funktionen versuchen, Aufrufe von potentiell gefährlichen Systemaufrufen zu beobachten oder verdächtige Trigger-Bedingungen zu untersuchen; unbeabsichtigte Sicherheitsmängel können dabei nur schwer entdeckt werden, da sie meist nur durch fehlerhafte Abfragen und nicht durch die Aufrufe entstehen.

### 4.3.1. Anti-Debugging-Maßnahmen

Ohne zu sehr in die technischen Details zu gehen, sollen an dieser Stelle einige gängige Möglichkeiten vorgestellt werden, mit denen in der Praxis versucht wird, ein Reverse Engineering und insbesondere ein Debugging zu behindern. Tatsächlich greifen die genannten Maßnahmen aufgrund des technischen Fortschritts bei Debuggern und Hardware meist nicht mehr, was hier aber keine Rolle spielt, da nur die prinzipielle Machbarkeit vorgestellt werden soll.

- Deaktivieren von Interrupts: Wenn z. B. der Tastatur-Interrupt deaktiviert wird, kann der Debugger nicht mehr bedient werden. Hierfür sind verschiedene Möglichkeiten denkbar, z. B. das hardwareseitige Maskieren des Interrupts oder das Ersetzen des Interrupthandlers.

- Beobachtung der Systemzeit: Daß ein Programm unter der Kontrolle eines Debuggers nicht normal läuft, sondern zwischendurch angehalten wird, läßt sich softwareseitig durch eine Beobachtung der Systemzeit feststellen. Diese wird aus Sicht des untersuchten Programms Sprünge vollziehen, oder bestimmte Routinen brauchen länger als normal.

[15] Eine Dekompilierung würde zwar den Codeumfang reduzieren, kann aber u. U. die Erkennung unbeabsichtigter Sicherheitsmängel verhindern; siehe hierzu die Erwägungen in Kapitel 2.9.

- Nutzen von Hardware-Debugging-Features: Wenn der Debugger normalerweise z. B. Breakpoints mit Interrupt 3-Befehlen implementiert, so kann dies durch das Einstreuen von solchen Befehlen in den Code zur „Verwirrung“ des Debuggers ausgenutzt werden, d. h. dieser meldet Breakpoints an Stellen, an denen keine gesetzt wurden. Damit dies funktioniert, muß natürlich vom Programm eine Behandlungsroutine für den Interrupt 3 eingerichtet werden.

- Auslösen einer Ausnahmebedingung: Wenn während des Ablaufs unter Kontrolle eines Debuggers eine Ausnahmebedingung auftritt, so wird der Debugger aktiviert. Wenn das Programm die alleinige Kontrolle besitzt, so hat es die Möglichkeit, die Ausnahmebehandlung selbst zu übernehmen und den Ablauf normal fortzusetzen. Eine einfache Möglichkeit zum Erzeugen einer solchen Ausnahmebedingung besteht in der Division durch Null, aber auch andere Methoden sind praktikabel.

- Verschlüsselung: Die Verschlüsselung von Teilen des Codes verhindert nicht nur, daß sie im verschlüsselten Zustand disassembliert werden können, sondern macht auch das Setzen eines Code-Breakpoints durch Patchen des Programmcodes in diesem Bereich unmöglich.

- Überprüfung der Prozessor-Flags: Wenn das Programm im Single-Step-Modus ausgeführt wird (was normalerweise nicht der Fall ist), so läßt sich dies programmseitig feststellen.

- Just-In-Time-Entschlüsselung: Durch das Aktivieren des Single-Step-Features und Installation einer Behandlungsroutine für den Single-Step-Interrupt kann erreicht werden, daß stets nur der nächste auszuführende Befehl bzw. die nächsten `n` Bytes des Codes entschlüsselt vorliegen und nach Ausführen des Befehls sofort wieder verschlüsselt werden.

## 4.4. Disassemblierung

Eine Disassemblierung im engeren Sinn besteht lediglich in der Umsetzung von Objektcode in Assemblerbefehle; dies kann recht einfach mit Hilfe von Übersetzungstabellen erfolgen. Hierbei wird in erster Linie gefordert, daß eine Rückübersetzung durch einen Assembler wieder die ursprüngliche Binärdarstellung liefert. Teilweise ist dies nicht oder nur schwer möglich, z. B. kommen in der 80x86-Reihe einige Assemblerbefehle vor, für die mehrere binäre Äquivalente existieren und die durch verschiedene Assembler auch unterschiedlich übersetzt werden. Dies spielt aber nur in Spezialfällen eine Rolle; für eine reine Analyse ohne spätere Reassemblierung ist der Erhalt der Semantik ausreichend.

Zu einer Disassemblierung in diesem Sinne sind bereits Debugger in der Lage; es kommt lediglich vor, daß diese aufgrund von technischen Neuerungen oder undokumentierten Befehlen nicht den kompletten Befehlssatz eines bestimmten Prozessors

beherrschen. Weiterhin werden gelegentlich Befehle bei nachfolgenden Prozessor-Generationen wieder entfernt und die freigewordenen Bytekombinationen („Opcodes“) wieder neu belegt; für einen Debugger wäre es hier sinnvoll, sie für den Prozessor zu interpretieren, auf dem er gerade läuft.

Die eigentliche Schwierigkeit besteht also nicht in der Übersetzung der Opcodes, sondern darin, ein verständliches Assemblerlisting zu erzeugen, das dann weiter untersucht werden kann. Die hierzu verwendeten Techniken bilden gleichzeitig die Grundlage für eine Dekompilierung, da die erhaltenen Informationen auch dort benötigt werden. Der Übergang ist hierbei fließend, da umgekehrt auch einige Dekompilierungstechniken wie die Identifizierung von Bibliotheksfunktionen die Lesbarkeit eines Assemblerlistings verbessern. Der Vorgang muß nicht vollständig durch die Software erfolgen[16], sondern kann manuell unterstützt werden; im Extremfall kann man auch einen Debugger für die Disassemblierung der einzelnen Befehle verwenden und den Rest der Analyse per Hand durchführen. Im einzelnen sind die folgenden Schritte notwendig bzw. sinnvoll:

**Verarbeitung von Dateiformaten:** Ein binäres Programm besteht nur in Ausnahmefällen aus reinem Objektcode[17], sondern enthält diverse Verwaltungsinformationen, z. B. für die Relokation des Codes, die Bindung an Bibliotheken, die Aufteilung in Code- und Datensegmente, das dynamische Laden von Programmteilen etc. Diese Strukturen müssen natürlich ausgewertet werden, die enthaltenen Informationen können außerdem die weitere Analyse erleichtern, indem etwa die Aufteilung in Code und Daten vorgegeben wird. Dieser Schritt wird für gewöhnlich automatisch durchgeführt, was bei ausreichender Dokumentation kein Problem darstellt; auch Debugger müssen dies für ihre jeweilige Zielplattform beherrschen.

**Trennung von Code und Daten:** Damit ein sinnvolles Assemblerlisting erzeugt werden kann, muß eine Analyse des Programmflusses durchgeführt werden, um nicht versehentlich Daten-Bytes als Befehle zu disassemblieren bzw. Befehls-Bytes als Daten zu behandeln. Die Analyse beginnt bei einem Startpunkt, der im verwendeten Dateiformat des Programms implizit oder explizit abgelegt ist, und folgt dem Programmfluß. Bei Verzweigungen wird rekursiv vorgegangen, und bei Erreichen von bereits analysiertem Code bricht die Rekursion ab. Problematisch sind hierbei zum einen selbstmodifizierender Code, zum anderen die dynamische Berechnung von Sprungadressen.

Die zweite Einschränkung läßt sich durch eine (u. U. manuelle) Befehls-Simulation bei der Analyse umgehen, in der die Berechnung nachvollzogen wird; das Programm kann aber auch parallel in einem Debugger beobachtet werden, um die Sprungziele zu finden. Denkbar ist auch ein automatisches Tracing auf

[16] Dies ist teilweise gar nicht möglich, insbesondere wenn das Programm absichtlich so gestaltet wurde, daß eine Analyse erschwert wird, was bei Malware häufig der Fall ist.

[17] Dies ist etwa bei MS-DOS .COM-Dateien der Fall.

Maschinenbefehlsebene, bei dem die Adressen des durchlaufenen Codes oder sogar nur die Ziele der erfolgten Sprünge aufgezeichnet werden.

**Analyse von Datentypen:** Moderne Prozessoren unterstützen bereits verschiedene Datentypen, im allgemeinen sind mindestens Byte, 2-Byte-Wort, 4-Byte-Wort und verschiedene Fließkommaformate vorhanden. Diese Datentypen werden dementsprechend auch vom Assembler angeboten und sollten daher im erzeugten Assemblerlisting verwendet werden, anstatt alle Datenbereiche als Folge von Bytes darzustellen. Hierzu muß eine Kreuz-Referenzierung durchgeführt werden, indem für jede Datenadresse vermerkt wird, mit welcher Wortbreite und mit welchem Befehlstyp sie angesprochen wird.

Weiterhin kann auch eine Analyse der Semantik der Daten vorgenommen werden, da ja durch ein Speicherwort Werte, Adressen oder Zeichen dargestellt werden können. Dies kann teilweise direkt anhand der zugreifenden Befehle geschehen. Teilweise gibt eine Analyse der umgebenden Befehle Aufschluß über eine etwaige Verwendung der Datenworte als Adressen. Zeichenketten lassen sich auch über Heuristiken im Datenbereich finden. Meist ist aber in diesem Bereich eine manuelle Intervention nötig, da Daten nicht immer über konstante Adressen angesprochen werden; insbesondere bei großen Datenstrukturen werden Adressen dynamisch berechnet, lediglich die Anfangsadresse der Struktur kann konstant sein. Für im Code eingebettete konstante Daten gilt prinzipiell dasselbe, lediglich die Wortbreite muß nicht erst bestimmt werden, und es treten üblicherweise statt Datenstrukturen nur einfache Datentypen auf.

**Kommentierung von System- oder Bibliotheksaufrufen:** Wenn neben dem Ziel-Prozessor auch das Laufzeitsystem des analysierten Programms bekannt ist (was im allgemeinen der Fall sein dürfte), können Aufrufe des Betriebssystems oder von Hilfsbibliotheken identifiziert und kommentiert werden. Dies ist insbesondere dann sehr sinnvoll, wenn die Aufrufe nicht über symbolische Namen, sondern über numerische Indizes erfolgen, wie es z. B. unter MS-DOS üblich ist.

Aber auch bei der Verwendung sprechender Namen ist eine Kommentierung hilfreich; selbst wenn die grundlegende Bedeutung des jeweiligen Aufrufs sich aus dem Namen ergibt, kann sie zusätzlich Aufschluß über die Belegung der Parameter liefern. Die erhaltenen Typinformationen über die Parameter können außerdem im Programm propagiert werden und so das Listing lesbarer machen.

**Identifizierung von Bibliotheksfunktionen:** Da ein dynamisches Binden von Bibliotheken zur Laufzeit nicht unter allen Betriebssystemen möglich [18] oder üblich ist

[18] Insbesondere unter älteren Systemen wie MS-DOS muß diese Funktionalität „zu Fuß“ implementiert werden.

und selbst auf aktuellen Systemen aus Performanz-[19] oder anderen Gründen[20] nicht immer praktiziert wird, werden die benötigten Bibliotheksfunktionen oft statisch in das Binärprogramm eingebunden. Dies kann dazu führen, daß der größte Teil des untersuchten Programms nicht aus originärem Code, sondern aus Bibliotheks- und Startup-Routinen besteht, deren Analyse keinen Informationsgewinn liefert.

Es ist daher wünschenswert, mit Hilfe von Compiler- und Funktions-Signaturen den originären Code automatisch von Hilfsroutinen zu unterscheiden, um keine Zeit mit deren Analyse zu verschwenden. Dabei ist es nicht nur so, daß die automatische Identifizierung Arbeit erspart, sondern eine Benennung und Kommentierung wie bei externen Bibliotheksaufrufen ist noch wesentlich hilfreicher als das Ergebnis einer manuellen Analyse. Die exakte Semantik kann dann bei Bedarf in einer gewöhnlichen Programmier-Referenz nachgeschlagen werden, und man wird nicht von vornherein von der Codemenge „erschlagen". Der Startup-Code des Compilers hingegen kann normalerweise ganz vernachlässigt werden; er ist höchstens dann von Interesse, wenn in ihm die Ursache eines Fehlers zu finden ist.

Die Erstellung von Signaturen kann glücklicherweise automatisiert werden; die Arbeit bei der Implementation einer automatischen Erstellung besteht hauptsächlich darin, die relevanten Bibliotheksformate zu unterstützen und eine sinnvolle Hash-Funktion zu finden [8]. Bei der Analyse eines konkreten Programmes müssen dabei nicht alle Signaturen für den Vergleich verwendet werden, sondern es kann eine Vorauswahl anhand von Compilersignaturen stattfinden, die allerdings üblicherweise manuell erstellt werden müssen.

Als Nachweis für die praktische Nutzbarkeit der beschriebenen Techniken kann neben [8] auch der kommerzielle Disassembler IDA Pro [19] dienen, in dem diese implementiert sind und auf dessen Website sich weitere Informationen über die konkrete Anwendung, insbesondere über die Erstellung und Verwendung von Funktions-Signaturen, finden.

## 4.5. Dekompilierung

Wie bereits erläutert, sind die bei einer Disassemblierung vorgenommenen Schritte größtenteils auch für eine Dekompilierung nötig. Verzichtet werden kann dabei lediglich auf die Kommentierung von Systemaufrufen, da diese entweder im Quelltext

[19] Die Performanz kann sich beim statischen Binden durch einen geringeren Overhead beim Funktionsaufruf verbessern. Allerdings kehrt sich der Effekt durch den erhöhten Speicherbedarf meist ins negative, sobald die Bibliothek von mehreren, gleichzeitig laufenden Prozessen benötigt wird.

[20] Wenn die Verfügbarkeit der benötigten Bibliothek nicht vorausgesetzt werden kann, müßte diese mitgeliefert werden. Dies ist aber relativ unsinnig, wenn nur ein kleiner Teil der Bibliothek gebraucht wird, was zur Übertragung bzw. Installation von unnötigem Ballast führen würde; auch lizenzrechtliche Probleme wie bei der UNIX-Motif-Bibliothek können eine Rolle spielen.

nicht vorkommen und daher auch nicht vom Compiler erzeugt[21] oder wie Bibliotheksfunktionen durch ihren Namen angesprochen[22] werden. Die Identifizierung von eingebundenen Bibliotheksfunktionen hat hier hingegen einen besonderen Stellenwert, da eine Dekompilierung von Startup- und Bibliothekscode noch weniger Sinn als eine Disassemblierung macht. Zum einen wird derartiger Code häufig in Assembler geschrieben, zum anderen gibt es keine andere Möglichkeit, um im resultierenden Quelltext die ursprünglichen Funktionsbezeichner zu erhalten.

Die folgende Darstellung der verschiedenen Dekompilierungsphasen basiert hauptsächlich auf [5]. Es finden sich in der Literatur zwar auch andere Ansätze, die aber größtenteils in Bezug auf Maschinen, Sprachen und Compiler nicht generisch genug sind, prinzipbedingt nur eine ineffiziente Emulation der Maschinenbefehle in einer Hochsprache liefern oder sonstige Nachteile besitzen.

Dem Compilerbau ähnlich wird eine Einteilung in drei Abschnitte vorgenommen, die jeweils maschinenspezifische, generische und sprachspezifische Phasen zusammenfassen. Dies dient ähnlich wie bei Compilern der leichteren Einbindung von neuen Maschinen und Sprachen, wobei natürlich gegenüber Compilern Quelle und Ziel vertauscht sind.

### 4.5.1. Maschinen- und systemspezifische Phasen

#### Syntaktische Analyse

Die Aufgaben dieser Phase decken sich größtenteils mit denen eines Disassemblers. Im einzelnen finden hier das Laden des Binärprogramms, die Gruppierung von Code-Bytes zu Befehlen und die Trennung von Code und Daten statt. Die Signaturen für Compiler und Bibliotheken werden ebenfalls bereits in dieser Phase auf den Code angewendet, da durch das Überspringen des Startup-Codes die Analyse direkt mit dem eigentlichen Programm beginnen kann und Bibliotheksfunktionen ebenfalls von der Analyse ausgenommen werden können.

Das Ergebnis dieser Phase besteht aus Assemblerbefehlen, die allerdings wegen der erforderlichen Weiterverarbeitung nicht als Klartext, sondern als Strukturen mit einer 3-Adreß-Notation abgelegt sind. Diese Darstellung erhält alle Informationen, so daß an dieser Stelle noch die Möglichkeit besteht, ein Assemblerlisting zu erzeugen. Die 3-Adreß-Notation dient dazu, auch bei 2-Adreß-Maschinen alle Parameter explizit zu nennen, da dort oft einer der Parameter als Quelle *und* Ziel dienen muß.

#### Semantische Analyse

In dieser Phase findet eine erste heuristische Analyse des Codes statt, indem Gruppen von Befehlen zusammengefaßt werden. Dies kann mit Hilfe von sog. Idiomen geschehen, d. h. Befehlsfolgen, die auf einer bestimmten Maschine häufig für bestimmte Aufgaben eingesetzt werden und deren Semantik sich nicht direkt aus den einzelnen

[21] Unter MS-DOS oder Linux werden Betriebssystemaufrufe in Hochsprachenprogrammen meist durch Bibliotheksfunktionen gekapselt.

[22] Unter MS-Windows werden Kernel-Funktionen auf diese Weise aufgerufen.

Befehlen ergibt. Beispiele hierfür wären die Multiplikation mit einer Zweierpotenz, die auf Maschinenebene aus Effizienzgründen meist als bitweises Schieben dargestellt wird, oder der Umgang mit Datentypen, die nicht direkt mit den zur Verfügung stehenden Maschinenbefehlen manipuliert werden können, wie etwa die Addition von 32-Bit-Zahlen auf einer 16-Bit-Maschine.

In diesem Zusammenhang werden auch Typinformationen erfaßt und propagiert. Wenn also z. B. festgestellt wird, daß zwei aufeinanderfolgende 16-Bit-Speicherworte als eine 32-Bit-Zahl behandelt werden, so wird diese Information gespeichert und der entsprechende Typ im entsprechenden Gültigkeitsbereich propagiert. Die Ermittlung des Gültigkeitsbereiches geschieht durch spezielle Algorithmen, je nachdem, ob es sich um Register, lokale Variablen und Parameter oder globale Variablen handelt.

### Erzeugung einer Zwischendarstellung

Hier werden die gespeicherten Assemblerbefehle in eine Form überführt, die den Hochsprachen näher steht; statt den vielen einzelnen Opcodes gibt es jetzt nur noch die Kategorien Zuweisung, bedingter Sprung, unbedingter Sprung, Subroutinenaufruf, Subroutinenrücksprung, Push und Pop. Ein arithmetischer Befehl nimmt also jetzt z. B. statt `add x,y` (in einer 3-Adreß-Notation `add x,x,y`) die Form `asgn`[23] `x,x+y` an. Die Anzahl der Befehle wird gegenüber dem Ergebnis der semantischen Analyse nicht verändert, die neue Darstellungsweise bereitet dies lediglich vor.

### Erzeugung eines Kontrollfluß-Graphen

Um einen Kontrollfluß-Graphen zu konstruieren, wird das Programm zuerst in Unterroutinen aufgeteilt. Jede Unterroutine wird dann weiter in sog. Basis-Blöcke unterteilt, wobei ein Basis-Block jeweils die maximale Folge von Instruktionen ist, die genau einen Eintritts- und einen Endpunkt hat [5]. Jeder Basis-Block wird also immer von Anfang bis Ende durchlaufen; die Menge der Instruktionen eines Programms läßt sich eindeutig in eine Menge von zueinander disjunkten Basis-Blöcken aufteilen.

Der Kontrollfluß-Graph besteht nun aus den Basis-Blöcken als Knoten und den Kontrollfluß-Beziehungen als Kanten. Es gibt folgende Typen von Blöcken:

- Block mit einem unbedingten Sprung am Ende: Der Block hat eine ausgehende Kante.
- Block mit einem bedingten Sprung am Ende: Der Block hat zwei ausgehende Kanten.
- Block mit einem indizierten Sprung am Ende: Der Block hat $n$ ausgehende Kanten.
- Block mit einem Subroutinenaufruf am Ende: Der Block hat zwei ausgehende Kanten, nämlich zum Folgebefehl und zur Subroutine.

[23] Assignment, d. h. Zuweisung; entspricht `x:=x+y`

- Block mit einem Subroutinenrücksprung am Ende: Der Block hat keine ausgehenden Kanten.

- Block mit einem normalen Befehl am Ende: Die Abtrennung eines solchen Blocks ist nötig, wenn die nächste Adresse ein Sprungziel darstellt. Die abgehende Kante führt zum Folgeblock.

In dieser Phase findet auch eine Optimierung des Graphen statt, da aufgrund von Beschränkungen des Compilers oder des Prozessors oft redundante Sprünge auftreten. Dabei kann ohne weiteres das Sprungziel von bedingten oder unbedingten Sprüngen, die auf unbedingte Sprungbefehle verweisen, durch das endgültige Sprungziel ersetzt werden.

### 4.5.2. Universeller Kern

#### Datenfluß-Analyse

Durch die Zusammenfassung von mehreren Operationen können hochsprachliche Ausdrücke gebildet werden, wodurch die Darstellung des Programms verbessert wird. In dem resultierenden Code werden keine temporären Register[24] mehr verwendet, und auch bedingte Sprungbefehle hängen nicht mehr von den Status-Flags des Prozessors ab, sondern enthalten den kompletten Vergleichsausdruck.

An dieser Stelle werden auch die Prototyp-Informationen für die eingebundenen Bibliotheksfunktionen ausgewertet, so daß Funktions-Parameter und -Rückgabewerte die korrekten Typen erhalten können. Die entsprechenden Typen können natürlich, soweit dies möglich ist, im Programm weiter propagiert werden und verbessern so die Darstellung.

#### Kontrollfluß-Analyse

Der bisherige Stand der Darstellung des Programms benutzt nicht die Hochsprachenkonstrukte für die Flußkontrolle wie etwa Schleifen und `if...then...else`-Anweisungen, sondern basiert noch auf den maschinenorientierten Sprunganweisungen. Da dieser `goto`-Stil in Hochsprachen zwar oft möglich, aber weder üblich noch wünschenswert ist, werden in dieser Phase die Graph-Strukturen so weit wie möglich auf Hochsprachenkonstrukte abgebildet. Damit dies sprachunabhängig geschehen kann, werden hierbei nur diejenigen Konstrukte verwendet, die in allen gängigen Programmiersprachen zur Verfügung stehen.

Die Strukturierung der Graphen und die Abbildung auf Hochsprachenkonstrukte geschieht durch aus der Graphentheorie stammende Algorithmen, auf die hier nicht weiter eingegangen werden soll, die aber in [5] ausführlich dargestellt werden.

[24] Dies gilt nicht für Variablen, Parameter oder Rückgabewerte von Funktionen, die während ihrer ganzen Lebensdauer in Registern gehalten werden.

### 4.5.3. Sprachspezifische Endstufe

#### Codeerzeugung

Da an diesem Punkt bis auf die Bezeichner alle notwendigen Informationen rekonstruiert worden sind, können die gespeicherten Strukturen ohne weitere Analyse in ein Hochsprachenprogramm überführt werden. Die Bezeichner werden dabei systematisch generiert, wobei es nicht sinnvoll erscheint, mit der Benennung zuviel semantische Information[25] auszudrücken; ein Benennungsschema nach Funktionen, lokalen und globalen Variablen sowie Parametern scheint aber gerechtfertigt. Die Codeerzeugung kann folgendermaßen rekursiv geschehen:

- Der Code für einen Basis-Block kann sehr einfach erzeugt werden, da es sich hauptsächlich um Zuweisungsoperationen handelt. Subroutinenaufrufe und Subroutinenrücksprünge stellen ebenfalls kein Problem dar, da hierfür ebenfalls direkte Hochsprachenäquivalente zur Verfügung stehen.
- Bedingte oder unbedingte Sprünge werden im Normalfall nicht direkt umgesetzt, sondern es wird die in der Kontrollfluß-Analyse gewonnene Information über das jeweilige Konstrukt betrachtet. Wenn kein bekanntes Sprachkonstrukt erkannt wurde, muß natürlich trotzdem ein `goto`[26] verwendet werden; in diesem Fall wird ein Bezeichner für ein Label dynamisch generiert und bei Erreichen des entsprechenden Sprungziels erneut ausgegeben. Für jedes der zur Verfügung stehenden Konstrukte existiert eine Schablone, in die lediglich die entsprechenden Bedingungen und (ggf. rekursiv) der Text für den oder die Untergraphen der Struktur einzusetzen sind.

Ein weiteres, bisher nicht beschriebenes Problem tritt auf, wenn ein Programm in eine andere Sprache dekompiliert werden soll als die, in der es ursprünglich geschrieben wurde. Da die durch die Anwendung von Bibliothekssignaturen erhaltenen Bezeichner für die Bibliotheksfunktionen in der neuen Sprache nicht vorhanden sind, kann das dekompilierte Programm zwar durchaus dem Verständnis dienen, wird sich aber nicht kompilieren lassen. Eine Lösung stellen die sog. „Library Bindings“ dar; in diesem Verfahren werden den Bibliotheksfunktionen einer Sprache bereits vorab die analogen Funktionen einer anderen Sprache zugeordnet, so daß bei einer Dekompilierung falls nötig eine Abbildung der Bezeichner (und ggf. der Aufruf-Syntax) erfolgen kann.

### 4.5.4. Nachbearbeitung

Aufgrund der Beschränkung auf die allen Hochsprachen gemeinsamen Kontrollkonstrukte kann es vorkommen, daß unnötigerweise `goto`-Statements erzeugt wurden, obwohl die aktuell verwendete Zielsprache andere Ausdrucksmöglichkeiten zur Verfügung stellt; ein Beispiel hierfür wäre das sofortige Beenden einer Schleife, das in

[25] Dies ist wohl nur durch zusätzliche Interaktion mit dem Benutzer zu erreichen.
[26] Daher kann eine Dekompilierung in eine `goto`-lose Sprache unter Umständen unmöglich sein.

C durch `break` möglich ist. Ebenso sind oft mehrere äquivalente Darstellungsmöglichkeiten vorhanden, so läßt sich etwa eine `for`-Schleife stets mittels einer `while`-Schleife nachbilden[27].

Ob diese Möglichkeiten im Quellcode des zu analysierenden Programms verwendet wurden, läßt sich zwar meist nicht feststellen[28], da ein guter Compiler bei semantisch äquivalentem Quelltext denselben Binärcode erzeugen sollte. Da sich die Lesbarkeit des erzeugten Codes durch ihre Nutzung aber deutlich erhöhen würde, erscheint es sinnvoll, die vom Decompiler erzeugten generischen Konstrukte durch sprachspezifische Kontrollstrukturen zu verbessern.

[27] Dies hat auch dazu geführt, daß in der ersten Definition der Sprache Oberon die `for`-Schleife fehlte. Allerdings wurde sie in der nächsten Sprachversion wieder eingeführt, da der Nutzen der erweiterten Ausdrucksmöglichkeiten offenbar den des geringeren Sprachumfangs überwog.

[28] Insbesondere müßte berücksichtigt werden, ob diese in der ursprünglichen Sprache überhaupt zur Verfügung standen, obwohl der erzeugte Code sonst keine Rücksicht auf die ursprüngliche Quellsprache nimmt.

# 5. Ausblick

Die in der EU für das Reverse Engineering geltenden rechtlichen Rahmenbedingungen sind aus informatischer Sicht sicherlich nicht völlig befriedigend. Besonders auffällig ist das Defizit im Bereich der IT-Sicherheit, da nicht vertrauenswürdige Software nicht ausreichend auf ihre sicherheitsrelevanten Eigenschaften hin untersucht werden kann. Dies war wahrscheinlich mit ausschlaggebend für die erörterte Änderung im australischen Urheberrecht, die den Sicherheitsaspekt zwar gesondert berücksichtigt, diesbezüglich aber etwas unklar formuliert ist.

Die heutigen Einsatzgebiete des Reverse Engineerings in der IT-Sicherheit, der Softwarewartung und der Herstellung von Interoperabilität werden wahrscheinlich auch in Zukunft den Großteil der Anwendungen stellen. Um dabei der zunehmenden Softwarevielfalt und der ebenfalls wachsenden Größe der Programme zu begegnen, wird es erforderlich sein, die Analyse weiter zu automatisieren. Dies muß nicht in jedem Fall eine vollständige maschinelle Analyse bedeuten, es ist jedoch nötig, den Analysten besser bei seiner Arbeit zu unterstützen.

Recht gut sehen die Möglichkeiten zur Automatisierung bei der Auswertung von Beobachtungsergebnissen und beim Tracing und Spying aus. Hier kann die Informationsflut durch entsprechende Filtermechanismen soweit reduziert werden, daß der Rest der Analyse nicht mehr viel Zeit in Anspruch nimmt[1] oder zumindest entschieden werden kann, ob sich eine weitergehende Analyse lohnt.

Die derzeit verwendeten Debugger sind zwar technisch sehr weit entwickelt und bieten eine für die meisten Fälle ausreichende Unterstützung, stellen jedoch trotz allem Komfort lediglich mächtige Hilfsmittel dar, die einer qualifizierten Bedienung bedürfen. Hier ist wohl – wenn man von der Unterstützung neuer Plattformen u. ä. absieht – nur noch wenig Fortschritt zu erwarten, jedoch dürften Debugger für spezielle Detailfragen unverzichtbar bleiben, ganz zu schweigen vom Haupteinsatzzweck, der Fehlersuche in eigenen Programmen, die allerdings auch nur bedingt als Reverse Engineering bezeichnet werden kann.

Die im Einsatz befindlichen Disassembler (namentlich IDA Pro und Sourcer) scheinen inzwischen recht ausgereift zu sein. Allerdings ist selbst bei einer vollständigen und korrekten Disassemblierung (die ja insbesondere bei Malware oft nicht ohne weiteres möglich ist) noch ein menschlicher Experte nötig, da Assembler heute nicht mehr sehr oft als Programmiersprache eingesetzt wird und die nötigen Kennt-

[1] Dies hängt zwar von den gewünschten Informationen ab; wenn jedoch ein tiefergehendes Verständnis eines Programmes gewünscht wird, sind von vornherein eher Debugging, Disassemblierung oder Dekompilierung die Methoden der Wahl.

nisse daher wenig verbreitet sind. Abgesehen davon ist die reine Disassemblierung nur der erste Schritt der Analyse; aussagekräftig wird ein Assemblerlisting erst durch Kommentierung und passende Umbenennung der Bezeichner, was nur teilweise automatisierbar ist, da dies ein wirkliches Verständnis erfordert.

Fortschritte sind hingegen bei der Dekompilierung zu erwarten; die Fähigkeiten der verschiedenen kommerziell verfügbaren Decompiler konnten zwar in dieser Arbeit nicht untersucht werden, jedoch lassen sich bereits aus der in Kapitel 4.5 vorgestellten Architektur eines Decompilers weiterführende Möglichkeiten erarbeiten:

- Die Analyse von dynamisch berechneten Sprungzielen kann verbessert werden [5].
- Es können zusätzlich zu den einfachen Datentypen auch komplexere wie Arrays, Records und Zeiger berücksichtigt werden [5].
- Die vorgestellte Architektur beschränkt sich auf imperative Sprachen. Da heute vielfach objektorientierte Sprachen eingesetzt werden, wäre auch dies ein zu berücksichtigender Punkt[2].

Da Software heute – außer für Spezialzwecke wie etwa Gerätetreiber – nicht mehr in Assembler geschrieben wird, besteht die Hoffnung, daß aufgrund des von Compilern produzierten „sauberen[3]“ Codes bei entsprechender Weiterentwicklung der Dekompilierungstechniken fast alle Programme automatisch soweit dekompiliert werden können, daß der resultierende Hochsprachencode semantisch äquivalent zu dem ursprünglichen Quelltext ist. Bei den heute zur Verfügung stehenden Prozeß- und Speicherkapazitäten ist außerdem selbst bei großen Programmen nicht zu befürchten, daß ein diesbezüglicher Engpaß auftritt; noch vor einigen Jahren hätte dies den Kreis der potentiellen Anwender möglicherweise eingeschränkt. Zusammenfassend läßt sich also sagen, daß in der automatischen Dekompilierung noch sehr viel Potential steckt, wobei aber nicht sicher ist, ob dieses auch ausgeschöpft werden kann – sei es aufgrund der rechtlichen Beschränkungen, des mangelnden Bedarfs, da ja Alternativen zur Verfügung stehen, oder des doch beträchtlichen Aufwands.

[2] Es existieren bereits Decompiler für die objektorientierte Sprache Java [10]. Der für Java verwendete Binärcode („Bytecode“) ist allerdings exakt auf diese Sprache zugeschnitten und spiegelt die Objektorientierung bereits wieder, so daß zu vermuten ist, daß sich bei auf konventionellen Maschinenarchitekturen implementierten objektorientierten Sprachen trotzdem Probleme ergeben. Beispielsweise werden die objektorientierten Mechanismen – wie etwa die Vererbung – in verschiedenen Sprachen oder auch nur verschiedenen Compilern oft unterschiedlich implementiert; dies spiegelt sich auch darin wieder, daß aufgrund eines fehlenden Binärstandards ein sog. „Mixed Language Programming“ (Mischen von Modulen in verschiedenen Programmiersprachen) oft nicht oder nur rein imperativ möglich ist.

[3] D. h. nicht selbstmodifizierend usw.

# 6. Literaturverzeichnis


[1] BGH. *Kostenlose Kassenzahnarztsoftware,* Urteil vom 8. Juli 1993, abgedruckt in JurPC 11/93

[2] Bochs Software Company. *A highly portable x86 PC emulator,* `http://www.bochs.com`

[3] Bögeholz, Harald. *Bilder in Ketten – Kopierschutz bei der DVD-Video,* c't 20/1999

[4] Brinkley, D. *Intercomputer transportation of assembly language software through decompilation,* 1981

[5] Cifuentes, Cristina. *Reverse Compilation Techniques,* PhD dissertation, Queensland University of Technology, School of Computing Science, 1994.

[6] Cifuentes, Cristina, et al. *Preliminary Experiences with the Use of the UQBT Binary Translation Framework,* The University of Queensland, Department of Computer Science and Electrical Engineering, 1999

[7] *DOSEMU, the PC Emulator for x86 based *nix,* `http://www.dosemu.org`

[8] Emmerik, Mike Van. *Signatures for Library Functions in Executable Files,* Queensland University of Technology, Faculty of Information Technology

[9] Feldmann, Börries von. *Notwehr,* Münchener Kommentar zum Bürgerlichen Gesetzbuch, Band 1, 3. Auflage, München 1993

[10] Ford, Daniel. *Jive : A Java decompiler,* 1996

[11] Friedman, F. *Decompilation and the transfer of mini-computer operating systems : Portability of systems oriented assembly language programs,* 1974

[12] Fromm, Friedrich Karl/Nordemann, Wilhelm. *Kommentar zum Urheberrechtsgesetz und zum Urheberrechtswahrnehmungsgesetz,* W. Kohlhammer, 9. Auflage 1998

[13] Gerber, Richard. *NERSC Tutorials: Debugging,* `http://hpcf.nersc.gov/training/tutorials/debug/`

[14] Guenther, Grant R., Campbell, David J. *The IOMEGA PPA3 parallel port SCSI Host Bus Adapter as embedded in the ZIP drive,* `/usr/src/linux-2.0.38/drivers/scsi/README.ppa`

[15] Heymann, Thomas. *Bringt die EG-Richtlinie in Wahrheit nichts Neues? Zweifel und Unklarheiten um das neue Software-Urheberrecht,* Computerwoche Nr. 44 vom 01.11.1991

[16] Himmelein, Gerald. *Mit edlen Motiven – Hintergründe zum Crack der DVD-Verschlüsselung,* c't 24/1999

[17] Hofbauer, Robert. *Die Überlassung von Standardsoftware,* Projektgruppenarbeit 1997, `http://www.fu-berlin.de/jura/netlaw/publikationen/beitraege/ss97-hofbauer.html#IV`

[18] Hollander, C. *Decompilation of object programs,* 1973

[19] *IDA Pro – The Interactive Disassembler,* `http://www.datarescue.com/ida.htm`

[20] Intel Corporation. *Intel 80386 Programmer's Reference Manual 1986,* 1987

[21] Jaeger, Stefan. *Legal oder illegal? Der rechtliche Status des DVD-Hackertools,* c't 14/2000

[22] König, Dr. M. Michael. *Dongle oder nicht Dongle ... (Das kleine Ärgernis),* `http://home.germany.net/100/55655/aufs69.htm` , in bearbeiteter Form auch in c't 12/1995

[23] LG Düsseldorf. *Dongle-Umgehung,* Urteil vom 20.3.1996, abgedruckt in CR 12/1996

[24] LG Mannheim. *Dongle,* Urteil vom 20.1.1995, abgedruckt in CR 9/1995

[25] Marly, Jochen. *Stellungnahme zum Diskussionsentwurf des Bundesjustizministeriums zur Änderung des Urheberrechtsgesetzes (Teil 2),* JurPC 7/1992

[26] Matloff, Norm. *Norm Matloff's Debugging Tutorial,* `http://heather.cs.ucdavis.edu/~matloff/debug.html`

[27] Mishra, Rohan. *Reverse Engineering in Japan and the Global Trend Towards Interoperability,* 1997, `http://www.murdoch.edu.au/elaw/issues/v4n2/mishra42.html`

[28] OLG Düsseldorf. *Dongle-Umgehung,* Urteil vom 27.3.1997, abgedruckt in CR 6/1997

[29] OLG Karlsruhe. *Entfernung von Dongle-Schutz – Dongle-Abfrage,* Urteil vom 10.1.1996, abgedruckt in NJW 39/1996

[30] OLG München. *Dongle,* Urteil vom 22.6.1995, abgedruckt in CR 1/1996

[31] One Hundred Fifth Congress of the United States of America. *Digital Millennium Copyright Act,* `http://www.eff.org/ip/DMCA/hr2281_dmca_law_19981020_pl105-304.html`

[32] The Parliament of Australia. *Copyright Amendment (Computer Programs) Bill 1999,* `http://scaletext.law.gov.au/html/comact/10/6017/top.htm`

[33] Pavey, D. J., Winsborrow, L. A. *Demonstrating equivalence of source code and PROM contents,* The Computer Language, 36(7), 1993

[34] Plex86. *An extensible open source PC virtualization software program,* `http://www.plex86.org`, ehemalig FreeMWare

[35] Post, Uwe. *Gleitzeit – Performance Monitoring deckt Gleitkommanutzung auf,* c't 9/97

[36] Raubenheimer, Andreas. *Softwareschutz nach den Vorschriften des UWG,* CR 5/1994

[37] Raubenheimer, Andreas. *Die neuen urheberrechtlichen Vorschriften zum Schutz von Computerprogrammen,* Mitteilungen der deutschen Patentanwälte 12/1994

[38] Schricker, Gerhard. *Urheberrecht, Kommentar,* München, C. H. Beck, 2. Auflage 1999

[39] Schulzki-Haddouti, Christiane. *Die rechtlichen Konsequenzen des DeCSS-Falls,* Telepolis, `http://www.heise.de/tp/deutsch/inhalt/te/5728/1.html`

[40] Transmeta. *Crusoe Technology,* `http://www.transmeta.com/crusoe/technoloy.html`

[41] United States Court Of Appeals For The Federal Circuit. *Atari Games Corp. and Tengen, Inc., vs. Nintendo Of America Inc. And Nintendo Co., Ltd.,* `http://www.ps.uci.edu/~jhkim/rpg/copyright/cases/atari_vs_nintendo.html`

[42] United States Court of Appeals For The First Circuit. *Lotus Development Corporation v. Borland International, Inc.,* `http://samsara.law.cwru.edu/comp_law/lotus.html`, 1995

[43] United States Court of Appeals For The Ninth Circuit. *Sega Enterprises Ltd. v. Accolade, Inc.,* `http://www.virtualrecordings.com/sega.htm`

[44] VMWare, Inc. `http://www.vmware.com/`

[45] Wine. *Windows Emulator for x86 *nix,* `http://www.winehq.com`

[46] Yoo, C. *A study of the program disassembly using flow analysis techniques,* 1985

Die JurPC-Artikel sind unter `http://www.makrolog.de/jurpc.nsf` als Faksimile herunterzuladen.

Die Veröffentlichungen von Cristina Cifuentes sind unter `http://tempura.cs.uq.edu.au/personal/cristina/pubs.html` zu finden.

# A. Softwarerichtlinie

## RICHTLINIE DES RATES vom 14. Mai 1991 über den Rechtsschutz von Computerprogrammen (91/250/EWG)

DER RAT DER EUROPÄISCHEN GEMEINSCHAFTEN –

- gestützt auf den Vertrag zur Gründung der Europäischen Wirtschaftsgemeinschaft, insbesondere auf Artikel 100a,
- auf Vorschlag der Kommission [1],
- in Zusammenarbeit mit dem Europäischen Parlament [2],
- nach Stellungnahme des Wirtschafts- und Sozialausschusses [3],
- in Erwägung nachstehender Gründe:

1. Derzeit ist nicht in allen Mitgliedstaaten ein eindeutiger Rechtsschutz von Computerprogrammen gegeben. Wird ein solcher Rechtsschutz gewährt, so weist er unterschiedliche Merkmale auf.
2. Die Entwicklung von Computerprogrammen erfordert die Investition erheblicher menschlicher, technischer und finanzieller Mittel. Computerprogramme können jedoch zu einem Bruchteil der zu ihrer unabhängigen Entwicklung erforderlichen Kosten kopiert werden.
3. Computerprogramme spielen eine immer bedeutendere Rolle in einer Vielzahl von Industrien. Die Technik der Computerprogramme kann somit als von grundlegender Bedeutung für die industrielle Entwicklung der Gemeinschaft angesehen werden.
4. Bestimmte Unterschiede des in den Mitgliedstaaten gewährten Rechtsschutzes von Computerprogrammen haben direkte und schädliche Auswirkungen auf das Funktionieren des Gemeinsamen Marktes für Computerprogramme; mit der Einführung neuer Rechtsvorschriften der Mitgliedstaaten auf diesem Gebiet könnten sich diese Unterschiede noch vergrößern.
5. Bestehende Unterschiede, die solche Auswirkungen haben, müssen beseitigt und die Entstehung neuer Unterschiede muß verhindert werden. Unterschiede, die das Funktionieren des Gemeinsamen Marktes nicht in erheblichem Maße beeinträchtigen, müssen jedoch nicht beseitigt und ihre Entstehung muß nicht verhindert werden.

[1] ABl. Nr. C 91 vom 12. 4. 1989, S. 4, und ABl. Nr. C 320 vom 20. 12. 1990, S. 22.
[2] ABl. Nr. C 231 vom 17. 9. 1990, S. 78, und Beschluß vom 17. April 1991 (noch nicht im Amtsblatt veröffentlicht).
[3] ABl. Nr. C 329 vom 30. 12. 1989, S. 4.

6. Der Rechtsrahmen der Gemeinschaft für den Schutz von Computerprogrammen kann somit zunächst darauf beschränkt werden, grundsätzlich festzulegen, daß die Mitgliedstaaten Computerprogrammen als Werke der Literatur Urheberrechtsschutz gewähren. Ferner ist festzulegen, wer schutzberechtigt und was schutzwürdig ist, und darüber hinaus sind die Ausschließlichkeitsrechte festzulegen, die die Schutzberechtigten geltend machen können, um bestimmte Handlungen zu erlauben oder zu verbieten, sowie die Schutzdauer.

7. Für die Zwecke dieser Richtlinie soll der Begriff „Computerprogramm" Programme in jeder Form umfassen, auch solche, die in die Hardware integriert sind; dieser Begriff umfaßt auch Entwurfsmaterial zur Entwicklung eines Computerprogramms, sofern die Art der vorbereitenden Arbeit die spätere Entstehung eines Computerprogramms zuläßt.

8. Qualitative oder ästhetische Vorzüge eines Computerprogramms sollten nicht als Kriterium für die Beurteilung der Frage angewendet werden, ob ein Programm ein individuelles Werk ist oder nicht.

9. Die Gemeinschaft fühlt sich zur Förderung der internationalen Standardisierung verpflichtet.

10. Die Funktion von Computerprogrammen besteht darin, mit den anderen Komponenten eines Computersystems und den Benutzern in Verbindung zu treten und zu operieren. Zu diesem Zweck ist eine logische und, wenn zweckmäßig, physische Verbindung und Interaktion notwendig, um zu gewährleisten, daß Software und Hardware mit anderer Software und Hardware und Benutzern wie beabsichtigt funktionieren können.

11. Die Teile des Programms, die eine solche Verbindung und Interaktion zwischen den Elementen von Software und Hardware ermöglichen sollen, sind allgemein als „Schnittstellen" bekannt.

12. Diese funktionale Verbindung und Interaktion ist allgemein als „Interoperabilität" bekannt. Diese Interoperabilität kann definiert werden als die Fähigkeit zum Austausch von Informationen und zur wechselseitigen Verwendung der ausgetauschten Informationen.

13. Zur Vermeidung von Zweifeln muß klargestellt werden, daß der Rechtsschutz nur für die Ausdrucksform eines Computerprogramms gilt und daß die Ideen und Grundsätze, die irgendeinem Element des Programms einschließlich seiner Schnittstellen zugrunde liegen, im Rahmen dieser Richtlinie nicht urheberrechtlich geschützt sind.

14. Entsprechend diesem Urheberrechtsgrundsatz sind Ideen und Grundsätze, die der Logik, den Algorithmen und den Programmsprachen zugrunde liegen, im Rahmen dieser Richtlinie nicht urheberrechtlich geschützt.

15. Nach dem Recht und der Rechtsprechung der Mitgliedstaaten und nach den internationalen Urheberrechtskonventionen ist die Ausdrucksform dieser Ideen und Grundsätze urheberrechtlich zu schützen.

16. Im Sinne dieser Richtlinie bedeutet der Begriff „Vermietung" die Überlassung eines Computerprogramms oder einer Kopie davon zur zeitweiligen Verwendung und zu Erwerbszwecken; dieser Begriff beinhaltet nicht den öffentlichen Verleih, der somit aus dem Anwendungsbereich der Richtlinie ausgeschlossen bleibt.

17. Zu dem Ausschließlichkeitsrecht des Urhebers, die nicht erlaubte Vervielfältigung seines Werks zu untersagen, sind im Fall eines Computerprogramms begrenzte Ausnahmen für die Vervielfältigung vorzusehen, die für die bestimmungsgemäße Verwendung des Programms durch den rechtmäßigen Erwerber technisch erforderlich sind. Dies bedeutet, daß das Laden und Ablaufen, sofern es für die Benutzung einer Kopie eines rechtmäßig erworbenen Computerprogramms erforderlich ist, sowie die Fehlerberichtigung nicht vertraglich untersagt werden dürfen. Wenn spezifische vertragliche Vorschriften nicht vereinbart worden sind, und zwar auch im Fall des Verkaufs einer Programmkopie, ist jede andere Handlung eines rechtmäßigen Erwerbers einer Programmkopie zulässig, wenn sie für eine bestimmungsgemäße Benutzung der Kopie notwendig ist.

18. Einer zur Verwendung eines Computerprogramms berechtigten Person sollte nicht untersagt sein, die zum Betrachten, Prüfen oder Testen des Funktionierens des Programms notwendigen Handlungen vorzunehmen, sofern diese Handlungen nicht gegen das Urheberrecht an dem Programm verstoßen.

19. Die nicht erlaubte Vervielfältigung, Übersetzung, Bearbeitung oder Änderung der Codeform einer Kopie eines Computerprogramms stellt eine Verletzung der Ausschließlichkeitsrechte des Urhebers dar.

20. Es können jedoch Situationen eintreten, in denen eine solche Vervielfältigung des Codes und der Übersetzung der Codeform im Sinne des Artikels 4 Buchstaben a) und b) unerläßlich ist, um die Informationen zu erhalten, die für die Interoperabilität eines unabhängig geschaffenen Programms mit anderen Programmen notwendig sind.

21. Folglich ist davon auszugehen, daß nur in diesen begrenzten Fällen eine Vervielfältigung und Übersetzung seitens oder im Namen einer zur Verwendung einer Kopie des Programms berechtigten Person rechtmäßig ist, anständigen Gepflogenheiten entspricht und deshalb nicht der Zustimmung des Rechtsinhabers bedarf.

22. Ein Ziel dieser Ausnahme ist es, die Verbindung aller Elemente eines Computersystems, auch solcher verschiedener Hersteller, zu ermöglichen, so daß sie zusammenwirken können.

23. Von einer solchen Ausnahme vom Ausschließlichkeitsrecht des Urhebers darf nicht in einer Weise Gebrauch gemacht werden, die die rechtmäßigen Interessen des Rechtsinhabers beeinträchtigt oder die im Widerspruch zur normalen Verwendung des Programms steht.

24. Zur Wahrung der Übereinstimmung mit den Bestimmungen der Berner Übereinkunft über den Schutz literarischer und künstlerischer Werke sollte die Dauer des Schutzes auf die Lebenszeit des Urhebers und 50 Jahre ab dem 1. Januar des auf sein Todesjahr folgenden Jahres oder im Fall eines anonymen Werkes auf 50 Jahre nach dem 1. Januar des Jahres, das auf das Jahr der Erstveröffentlichung des Werkes folgt, festgesetzt werden.

25. Der Schutz von Computerprogrammen im Rahmen des Urheberrechts sollte unbeschadet der Anwendung anderer Schutzformen in den relevanten Fällen erfolgen. Vertragliche Regelungen, die im Widerspruch zu Artikel 6 oder den Ausnahmen nach Artikel 5 Absätze 2 und 3 stehen, sollten jedoch unwirksam sein.

26. Die Bestimmungen dieser Richtlinie lassen die Anwendung der Wettbewerbsregeln nach den Artikeln 85 und 86 des Vertrages unberührt, wenn ein marktbeherrschender

Anbieter den Zugang zu Informationen verweigert, die für die in dieser Richtlinie definierte Interoperabilität notwendig sind.

27. Die Bestimmungen dieser Richtlinie sollten unbeschadet spezifischer Auflagen bereits bestehender gemeinschaftlicher Rechtsvorschriften für die Veröffentlichung von Schnittstellen im Telekommunikationssektor oder von Ratsbeschlüssen betreffend die Normung im Bereich der Informations- und Telekommunikationstechnologie gelten.

28. Diese Richtlinie berührt nicht die in den einzelstaatlichen Rechtsvorschriften in Übereinstimmung mit der Berner Übereinkunft vorgesehenen Ausnahmeregelungen für Punkte, die nicht von der Richtlinie erfaßt werden –

– HAT FOLGENDE RICHTLINIE ERLASSEN:

## Artikel 1 – Gegenstand des Schutzes

(1) Gemäß den Bestimmungen dieser Richtlinie schützen die Mitgliedstaaten Computerprogramme urheberrechtlich als literarische Werke im Sinne der Berner Übereinkunft zum Schutze von Werken der Literatur und der Kunst. Im Sinne dieser Richtlinie umfaßt der Begriff „Computerprogramm“ auch das Entwurfsmaterial zu ihrer Vorbereitung.

(2) Der gemäß dieser Richtlinie gewährte Schutz gilt für alle Ausdrucksformen von Computerprogrammen. Ideen und Grundsätze, die irgendeinem Element eines Computerprogramms zugrunde liegen, einschließlich der den Schnittstellen zugrundeliegenden Ideen und Grundsätze, sind nicht im Sinne dieser Richtlinie urheberrechtlich geschützt.

(3) Computerprogramme werden geschützt, wenn sie individuelle Werke in dem Sinne darstellen, daß sie das Ergebnis der eigenen geistigen Schöpfung ihres Urhebers sind. Zur Bestimmung ihrer Schutzfähigkeit sind keine anderen Kriterien anzuwenden.

## Artikel 2 – Urheberschaft am Programm

(1) Der Urheber eines Computerprogramms ist die natürliche Person, die Gruppe natürlicher Personen, die das Programm geschaffen hat, oder, soweit nach den Rechtsvorschriften der Mitgliedstaaten zulässig, die juristische Person, die nach diesen Rechtsvorschriften als Rechtsinhaber gilt. Soweit kollektive Werke durch die Rechtsvorschriften eines Mitgliedstaats anerkannt sind, gilt die Person als Urheber, die nach den Rechtsvorschriften des Mitgliedstaats als Person angesehen wird, die das Werk geschaffen hat.

(2) Ist ein Computerprogramm von einer Gruppe natürlicher Personen gemeinsam geschaffen worden, so stehen dieser die ausschließlichen Rechte daran gemeinsam zu.

(3) Wird ein Computerprogramm von einem Arbeitnehmer in Wahrnehmung seiner Aufgaben oder nach den Anweisungen seines Arbeitgebers geschaffen, so ist ausschließlich der Arbeitgeber zur Ausübung aller wirtschaftlichen Rechte an dem so geschaffenen Programm berechtigt, sofern keine andere vertragliche Vereinbarung getroffen wird.

## Artikel 3 – Schutzberechtigte

Schutzberechtigt sind alle natürlichen und juristischen Personen gemäß dem für Werke der Literatur geltenden innerstaatlichen Urheberrecht.

## Artikel 4 – Zustimmungsbedürftige Handlungen

Vorbehaltlich der Bestimmungen der Artikel 5 und 6 umfassen die Ausschließlichkeitsrechte des Rechtsinhabers im Sinne des Artikels 2 das Recht, folgende Handlungen vorzunehmen oder zu gestatten:

a) die dauerhafte oder vorübergehende Vervielfältigung, ganz oder teilweise, eines Computerprogramms mit jedem Mittel und in jeder Form. Soweit das Laden, Anzeigen, Ablaufen, Übertragen oder Speichern des Computerprogramms eine Vervielfältigung erforderlich macht, bedürfen diese Handlungen der Zustimmung des Rechtsinhabers;

b) die Übersetzung, die Bearbeitung, das Arrangement und andere Umarbeitungen eines Computerprogramms sowie die Vervielfältigung der erzielten Ergebnisse, unbeschadet der Rechte der Person, die das Programm umarbeitet;

c) jede Form der öffentlichen Verbreitung des originalen Computerprogramms oder von Kopien davon, einschließlich der Vermietung. Mit dem Erstverkauf einer Programmkopie in der Gemeinschaft durch den Rechtsinhaber oder mit seiner Zustimmung erschöpft sich in der Gemeinschaft das Recht auf die Verbreitung dieser Kopie; ausgenommen hiervon ist jedoch das Recht auf Kontrolle der Weitervermietung des Programms oder einer Kopie davon.

## Artikel 5 – Ausnahmen von den zustimmungsbedürftigen Handlungen

(1) In Ermangelung spezifischer vertraglicher Bestimmungen bedürfen die in Artikel 4 Buchstaben a) und b) genannten Handlungen nicht der Zustimmung des Rechtsinhabers, wenn sie für eine bestimmungsgemäße Benutzung des Computerprogramms einschließlich der Fehlerberichtigung durch den rechtmäßigen Erwerber notwendig sind.

(2) Die Erstellung einer Sicherungskopie durch eine Person, die zur Benutzung des Programms berechtigt ist, darf nicht vertraglich untersagt werden, wenn sie für die Benutzung erforderlich ist.

(3) Die zur Verwendung einer Programmkopie berechtigte Person kann, ohne die Genehmigung des Rechtsinhabers einholen zu müssen, das Funktionieren dieses Programms beobachten, untersuchen oder testen, um die einem Programmelement zugrundeliegenden Ideen und Grundsätze zu ermitteln, wenn sie dies durch Handlungen zum Laden, Anzeigen, Ablaufen, Übertragen oder Speichern des Programms tut, zu denen sie berechtigt ist.

## Artikel 6 – Dekompilierung

(1) Die Zustimmung des Rechtsinhabers ist nicht erforderlich, wenn die Vervielfältigung des Codes oder die Übersetzung der Codeform im Sinne des Artikels 4 Buchstaben a) und b) unerläßlich ist, um die erforderlichen Informationen zur Herstellung der Interoperabilität eines unabhängig geschaffenen Computerprogramms mit anderen Programmen zu erhalten, sofern folgende Bedingungen erfüllt sind:

a) Die Handlungen werden von dem Lizenznehmer oder von einer anderen zur Verwendung einer Programmkopie berechtigten Person oder in deren Namen von einer hierzu ermächtigten Person vorgenommen;

b) die für die Herstellung der Interoperabilität notwendigen Informationen sind für die unter Buchstabe a) genannten Personen noch nicht ohne weiteres zugänglich gemacht; und

c) die Handlungen beschränken sich auf die Teile des ursprünglichen Programms, die zur Herstellung der Interoperabilität notwendig sind.

(2) Die Bestimmungen von Absatz 1 erlauben nicht, daß die im Rahmen ihrer Anwendung gewonnenen Informationen

a) zu anderen Zwecken als zur Herstellung der Interoperabilität des unabhängig geschaffenen Programms verwendet werden;

b) an Dritte weitergegeben werden, es sei denn, daß dies für die Interoperabilität des unabhängig geschaffenen Programms notwendig ist;

c) für die Entwicklung, Herstellung oder Vermarktung eines Programms mit im wesentlichen ähnlicher Ausdrucksform oder für irgendwelche anderen, das Urheberrecht verletzenden Handlungen verwendet werden.

(3) Zur Wahrung der Übereinstimmung mit den Bestimmungen der Berner Übereinkunft zum Schutz von Werken der Literatur und der Kunst können die Bestimmungen dieses Artikels nicht dahin gehend ausgelegt werden, daß dieser Artikel in einer Weise angewendet werden kann, die die rechtmäßigen Interessen des Rechtsinhabers in unvertretbarer Weise beeinträchtigt oder im Widerspruch zur normalen Nutzung des Computerprogramms steht.

## Artikel 7 – Besondere Schutzmaßnahmen

(1) Unbeschadet der Artikel 4, 5 und 6 sehen die Mitgliedstaaten gemäß ihren innerstaatlichen Rechtsvorschriften geeignete Maßnahmen gegen Personen vor, die eine der nachstehend unter den Buchstaben a), b) und c) aufgeführten Handlungen begehen:

a) Inverkehrbringen einer Kopie eines Computerprogramms, wenn die betreffende Person wußte oder Grund zu der Annahme hatte, daß es sich um eine unerlaubte Kopie handelt;

b) Besitz einer Kopie eines Computerprogramms für Erwerbszwecke, wenn diese betreffende Person wußte oder Grund zu der Annahme hatte, daß es sich um eine unerlaubte Kopie handelt;

c) das Inverkehrbringen oder der Erwerbszwecken dienende Besitz von Mitteln, die allein dazu bestimmt sind, die unerlaubte Beseitigung oder Umgehung technischer Programmschutzmechanismen zu erleichtern.

(2) Jede unerlaubte Kopie eines Computerprogramms kann gemäß den Rechtsvorschriften des betreffenden Mitgliedstaats beschlagnahmt werden.

(3) Die Mitgliedstaaten können die Beschlagnahme der in Absatz 1 Buchstabe c) genannten Mittel vorsehen.

## Artikel 8 – Schutzdauer

(1) Die Schutzdauer umfaßt die Lebenszeit des Urhebers und 50 Jahre nach seinem Tod bzw. nach dem Tod des letzten noch lebenden Urhebers; für anonym oder pseudonym veröffentlichte Computerprogramme oder für Computerprogramme, als deren Urheber in Übereinstimmung mit Artikel 2 Absatz 1 aufgrund der einzelstaatlichen Rechtsvorschriften eine juristische Person anzusehen ist, endet die Schutzdauer 50 Jahre, nachdem das Programm erstmals erlaubterweise der Öffentlichkeit zugänglich gemacht worden ist. Die Dauer des Schutzes beginnt am 1. Januar des Jahres, das auf die vorgenannten Ereignisse folgt.

(2) Die Mitgliedstaaten, in denen bereits eine längere Schutzdauer gilt als die, die in Absatz 1 vorgesehen ist, dürfen ihre gegenwärtige Schutzdauer so lange beibehalten, bis die Schutzdauer für urheberrechtlich geschützte Werke durch allgemeinere Rechtsvorschriften der Gemeinschaft harmonisiert ist.

## Artikel 9 – Weitere Anwendung anderer Rechtsvorschriften

(1) Die Bestimmungen dieser Richtlinie stehen sonstigen Rechtsvorschriften, so für Patentrechte, Warenzeichen, unlauteres Wettbewerbsverhalten, Geschäftsgeheimnisse und den Schutz von Halbleiterprodukten, sowie dem Vertragsrecht nicht entgegen. Vertragliche Bestimmungen, die im Widerspruch zu Artikel 6 oder zu den Ausnahmen nach Artikel 5 Absätze 2 und 3 stehen, sind unwirksam.

(2) Die Bestimmungen dieser Richtlinie finden unbeschadet etwaiger vor dem 1. Januar 1993 getroffener Vereinbarungen und erworbener Rechte auch auf vor diesem Zeitpunkt geschaffene Programme Anwendung.

## Artikel 10 – Schlußbestimmungen

(1) Die Mitgliedstaaten erlassen die erforderlichen Rechts - und Verwaltungsvorschriften, um dieser Richtlinie vor dem 1. Januar 1993 nachzukommen.

Wenn die Mitgliedstaaten diese Vorschriften erlassen, nehmen sie in ihnen selbst oder durch einen Hinweis bei der amtlichen Veröffentlichung auf diese Richtlinie Bezug. Sie regeln die Einzelheiten der Bezugnahme.

(2) Die Mitgliedstaaten teilen der Kommission die innerstaatlichen Rechtsvorschriften mit, die sie auf dem unter diese Richtlinie fallenden Gebiet erlassen.

## Artikel 11

Diese Richtlinie ist an die Mitgliedstaaten gerichtet.

Geschehen zu Brüssel am 14. Mai 1991. Im Namen des Rates
Der Präsident
J. F. POOS

# B. Relevante deutsche Gesetzestexte

## B.1. Gesetz über Urheberrecht und verwandte Schutzrechte, Urheberrechtsgesetz (UrhG)

### Erster Teil, Achter Abschnitt. Besondere Bestimmungen für Computerprogramme

#### § 69a. Gegenstand des Schutzes.

(1) Computerprogramme im Sinne dieses Gesetzes sind Programme in jeder Gestalt, einschließlich des Entwurfsmaterials.

(2) Der gewährte Schutz gilt für alle Ausdrucksformen eines Computerprogramms. Ideen und Grundsätze, die einem Element eines Computerprogramms zugrunde liegen, einschließlich der den Schnittstellen zugrundeliegenden Ideen und Grundsätze, sind nicht geschützt.

(3) Computerprogramme werden geschützt, wenn sie individuelle Werke in dem Sinne darstellen, daß sie das Ergebnis der eigenen geistigen Schöpfung ihres Urhebers sind. Zur Bestimmung ihrer Schutzfähigkeit sind keine anderen Kriterien, insbesondere nicht qualitative oder ästhetische, anzuwenden.

(4) Auf Computerprogramme finden die für Sprachwerke geltenden Bestimmungen Anwendung, soweit in diesem Abschnitt nichts anderes bestimmt ist.

#### § 69b. Urheber in Arbeits- und Dienstverhältnissen.

(1) Wird ein Computerprogramm von einem Arbeitnehmer in Wahrnehmung seiner Aufgaben oder nach den Anweisungen seines Arbeitgebers geschaffen, so ist ausschließlich der Arbeitgeber zur Ausübung aller vermögensrechtlichen Befugnisse an dem Computerprogramm berechtigt, sofern nichts anderes vereinbart ist.

(2) Absatz 1 ist auf Dienstverhältnisse entsprechend anzuwenden.

#### § 69c. Zustimmungsbedürftige Handlungen.

Der Rechtsinhaber hat das ausschließliche Recht, folgende Handlungen vorzunehmen oder zu gestatten:

1. die dauerhafte oder vorübergehende Vervielfältigung, ganz oder teilweise, eines Computerprogramms mit jedem Mittel und in jeder Form. Soweit das Laden, Anzeigen, Ablaufen, Übertragen oder Speichern des Computerprogramms eine Vervielfältigung erfordert, bedürfen diese Handlungen der Zustimmung des Rechtsinhabers;

2. die Übersetzung, die Bearbeitung, das Arrangement und andere Umarbeitungen eines Computerprogramms sowie die Vervielfältigung der erzielten Ergebnisse. Die Rechte derjenigen, die das Programm bearbeiten, bleiben unberührt;
3. jede Form der Verbreitung des Originals eines Computerprogramms oder von Vervielfältigungsstücken, einschließlich der Vermietung. Wird ein Vervielfältigungsstück eines Computerprogramms mit Zustimmung des Rechtsinhabers im Gebiet der Europäischen Gemeinschaften oder eines anderen Vertragsstaates des Abkommens über den Europäischen Wirtschaftsraum im Wege der Veräußerung in Verkehr gebracht, so erschöpft sich das Verbreitungsrecht in bezug auf dieses Vervielfältigungsstück mit Ausnahme des Vermietrechts.

## § 69d. Ausnahmen von den zustimmungsbedürftigen Handlungen.

(1) Soweit keine besonderen vertraglichen Bestimmungen vorliegen, bedürfen die in § 69c Nr. 1 und 2 genannten Handlungen nicht der Zustimmung des Rechtsinhabers, wenn sie für eine bestimmungsgemäße Benutzung des Computerprogramms einschließlich der Fehlerberichtigung durch jeden zur Verwendung eines Vervielfältigungsstücks des Programms Berechtigten notwendig sind.

(2) Die Erstellung einer Sicherungskopie durch eine Person, die zur Benutzung des Programms berechtigt ist, darf nicht vertraglich untersagt werden, wenn sie für die Sicherung künftiger Benutzung erforderlich ist.

(3) Der zur Verwendung eines Vervielfältigungsstücks eines Programms Berechtigte kann ohne Zustimmung des Rechtsinhabers das Funktionieren dieses Programms beobachten, untersuchen oder testen, um die einem Programmelement zugrundeliegenden Ideen und Grundsätze zu ermitteln, wenn dies durch Handlungen zum Laden, Anzeigen, Ablaufen, Übertragen oder Speichern des Programms geschieht, zu denen er berechtigt ist.

## § 69e. Dekompilierung.

(1) Die Zustimmung des Rechtsinhabers ist nicht erforderlich, wenn die Vervielfältigung des Codes oder die Übersetzung der Codeform im Sinne des § 69c Nr. 1 und 2 unerläßlich ist, um die erforderlichen Informationen zur Herstellung der Interoperabilität eines unabhängig geschaffenen Computerprogramms mit anderen Programmen zu erhalten, sofern folgende Bedingungen erfüllt sind:

1. Die Handlungen werden von dem Lizenznehmer oder von einer anderen zur Verwendung eines Vervielfältigungsstücks des Programms berechtigten Person oder in deren Namen von einer hierzu ermächtigten Person vorgenommen;
2. die für die Herstellung der Interoperabilität notwendigen Informationen sind für die in Nummer 1 genannten Personen noch nicht ohne weiteres zugänglich gemacht;
3. die Handlungen beschränken sich auf die Teile des ursprünglichen Programms, die zur Herstellung der Interoperabilität notwendig sind.

(2) Bei Handlungen nach Absatz 1 gewonnene Informationen dürfen nicht

1. zu anderen Zwecken als zur Herstellung der Interoperabilität des unabhängig geschaffenen Programms verwendet werden,

2. an Dritte weitergegeben werden, es sei denn, daß dies für die Interoperabilität des unabhängig geschaffenen Programms notwendig ist,
3. für die Entwicklung, Herstellung oder Vermarktung eines Programms mit im wesentlichen ähnlicher Ausdrucksform oder für irgendwelche anderen das Urheberrecht verletzenden Handlungen verwendet werden.

(3) Die Absätze 1 und 2 sind so auszulegen, daß ihre Anwendung weder die normale Auswertung des Werkes beeinträchtigt noch die berechtigten Interessen des Rechtsinhabers unzumutbar verletzt.

### § 69f. Rechtsverletzungen.

(1) Der Rechtsinhaber kann von dem Eigentümer oder Besitzer verlangen, daß alle rechtswidrig hergestellten, verbreiteten oder zur rechtswidrigen Verbreitung bestimmten Vervielfältigungsstücke vernichtet werden. § 98 Abs. 2 und 3 ist entsprechend anzuwenden.

(2) Absatz 1 ist entsprechend auf Mittel anzuwenden, die allein dazu bestimmt sind, die unerlaubte Beseitigung oder Umgehung technischer Programmschutzmechanismen zu erleichtern.

### § 69g. Anwendung sonstiger Rechtsvorschriften; Vertragsrecht.

(1) Die Bestimmungen dieses Abschnitts lassen die Anwendung sonstiger Rechtsvorschriften auf Computerprogramme, insbesondere über den Schutz von Erfindungen, Topographien von Halbleitererzeugnissen, Marken und den Schutz gegen unlauteren Wettbewerb einschließlich des Schutzes von Geschäfts- und Betriebsgeheimnissen, sowie schuldrechtliche Vereinbarungen unberührt.

(2) Vertragliche Bestimmungen, die in Widerspruch zu § 69d Abs. 2 und 3 und § 69e stehen, sind nichtig.

## Vierter Teil, Zweiter Abschnitt. Rechtsverletzungen

### 1. Bürgerlich-rechtliche Vorschriften; Rechtsweg

### § 97. Anspruch auf Unterlassung und Schadenersatz.

(1) Wer das Urheberrecht oder ein anderes nach diesem Gesetz geschütztes Recht widerrechtlich verletzt, kann vom Verletzten auf Beseitigung der Beeinträchtigung, bei Wiederholungsgefahr auf Unterlassung und, wenn dem Verletzer Vorsatz oder Fahrlässigkeit zur Last fällt, auch auf Schadenersatz in Anspruch genommen werden. An Stelle des Schadenersatzes kann der Verletzte die Herausgabe des Gewinns, den der Verletzer durch die Verletzung des Rechts erzielt hat, und Rechnungslegung über diesen Gewinn verlangen.

(2) Urheber, Verfasser wissenschaftlicher Ausgaben (§ 70), Lichtbildner (§ 72) und ausübende Künstler (§ 73) können, wenn dem Verletzer Vorsatz oder Fahrlässigkeit zur Last fällt, auch wegen des Schadens, der nicht Vermögensschaden ist, eine Entschädigung in Geld verlangen, wenn und soweit es der Billigkeit entspricht.

(3) Ansprüche aus anderen gesetzlichen Vorschriften bleiben unberührt.

### 2. Strafrechtliche Vorschriften

#### § 106. Unerlaubte Verwertung urheberrechtlich geschützter Werke.

(1) Wer in anderen als den gesetzlich zugelassenen Fällen ohne Einwilligung des Berechtigten ein Werk oder eine Bearbeitung oder Umgestaltung eines Werkes vervielfältigt, verbreitet oder öffentlich wiedergibt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

(2) Der Versuch ist strafbar.

#### § 109. Strafantrag.

In den Fällen der §§ 106 bis 108 wird die Tat nur auf Antrag verfolgt, es sei denn, daß die Strafverfolgungsbehörde wegen des besonderen öffentlichen Interesses an der Strafverfolgung ein Einschreiten von Amts wegen für geboten hält.

## B.2. Strafprozeßordnung (StPO)

#### § 153. Absehen von der Strafverfolgung bei Bagatellsachen

(1) Hat das Verfahren ein Vergehen zum Gegenstand, so kann die Staatsanwaltschaft mit Zustimmung des für die Eröffnung des Hauptverfahrens zuständigen Gerichts von der Verfolgung absehen, wenn die Schuld des Täters als gering anzusehen wäre und kein öffentliches Interesse an der Verfolgung besteht. Der Zustimmung des Gerichtes bedarf es nicht bei einem Vergehen, das nicht mit einer im Mindestmaß erhöhten Strafe bedroht ist und bei dem die durch die Tat verursachten Folgen gering sind.

(2) Ist die Klage bereits erhoben, so kann das Gericht in jeder Lage des Verfahrens unter den Voraussetzungen des Absatzes 1 mit Zustimmung der Staatsanwaltschaft und des Angeschuldigten das Verfahren einstellen. Der Zustimmung des Angeschuldigten bedarf es nicht, wenn die Hauptverhandlung aus den in § 205 angeführten Gründen nicht durchgeführt werden kann oder in den Fällen des § 231 Abs. 2 und der §§ 232 und 233 in seiner Abwesenheit durchgeführt wird. Die Entscheidung ergeht durch Beschluß. Der Beschluß ist nicht anfechtbar.

## B.3. Bürgerliches Gesetzbuch (BGB)

#### § 227. Notwehr

(1) Eine durch Notwehr gebotene Handlung ist nicht widerrechtlich.

(2) Notwehr ist diejenige Verteidigung welche erforderlich ist, um einen gegenwärtigen rechtswidrigen Angriff von sich oder einem anderen abzuwenden.

#### § 228. Notstand

Wer eine fremde Sache beschädigt oder zerstört, um eine durch sie drohende Gefahr von sich oder einem anderen abzuwenden, handelt nicht widerrechtlich, wenn die Beschädigung oder die Zerstörung zur Abwendung der Gefahr erforderlich ist und der Schaden nicht außer Verhältnis zu der Gefahr steht. Hat der Handelnde die Gefahr verschuldet, so ist er zum Schadensersatze verpflichtet.

## B.4. Strafgesetzbuch (StGB)

### § 32. Notwehr

(1) Wer eine Tat begeht, die durch Notwehr geboten ist, handelt nicht rechtswidrig.

(2) Notwehr ist die Verteidigung, die erforderlich ist, um einen gegenwärtigen rechtswidrigen Angriff von sich oder einem anderen abzuwenden.

### § 34. Rechtfertigender Notstand

Wer in einer gegenwärtigen, nicht anders abwendbaren Gefahr für Leben, Leib, Freiheit, Ehre, Eigentum oder ein anderes Rechtsgut eine Tat begeht, um die Gefahr von sich oder einem anderen abzuwenden, handelt nicht rechtswidrig, wenn bei Abwägung der widerstreitenden Interessen, namentlich der betroffenen Rechtsgüter und des Grades der ihnen drohenden Gefahren, das geschützte Interesse das beeinträchtigte wesentlich überwiegt. Dies gilt jedoch nur, soweit die Tat ein angemessenes Mittel ist, die Gefahr abzuwenden.