Diplomarbeit

Experimente mit einem Decompiler im Hinblick auf die forensische Informatik

André Janz
5janz@informatik.uni-hamburg.de

28. Mai 2002

Universität Hamburg
Fachbereich Informatik

Betreuer:
Prof. Dr. Klaus Brunnstein
Dr. Martin Lehmann

Erklärung: Ich versichere, daß ich die vorliegende Arbeit selbständig und ohne fremde Hilfe angefertigt habe und keine außer den angegebenen Quellen und Hilfsmitteln benutzt habe.

Hamburg, den 28. Mai 2002

Inhaltsverzeichnis

1 Einleitung

Der forensischen Informatik kommt heutzutage aufgrund der gestiegenen Bedrohungen durch Hackerangriffe und bösartige Software eine immer größere Bedeutung zu. Ein sehr wichtiges Gebiet der forensischen Informatik bildet dabei die Malware-Forschung, da die durch Malware verursachten Schäden ständig zunehmen: Laut [20] betrugen diese alleine im Fall des im April des Jahres 2000 verbreiteten "LoveLetter"-Wurms 700 Millionen US-Dollar.

Zur Entdeckung und Bekämpfung von Malware ist es erforderlich, diese mittels verschiedener Reverse-Engineering-Methoden zu analysieren. Die betreffenden Methoden werden zwar auch in anderen Bereichen eingesetzt; es bestehen jedoch im Bereich der Malware-Forschung, mit dem sich diese Arbeit beschäftigt, einige besondere Anforderungen:

- Die Analyse muß möglichst schnell durchgeführt werden, damit durch Malware verursachte mögliche Schäden rechtzeitig abgewendet werden können;
- die Analyse sollte möglichst automatisiert, d. h. ohne großen manuellen Aufwand ablaufen, da häufig eine große Anzahl von Programmen zu untersuchen ist;
- die Funktionalität der zu untersuchenden Programme muß vollständig aufgeklärt werden, damit mögliche negative Auswirkungen exakt festgestellt werden können.

Verfahren, die regelhaft zum Zwecke des Reverse Engineerings eingesetzt werden, sind Debugging, Tracing (Aufzeichnung des Programmablaufs) und Disassemblierung [57]. Diese haben bzgl. der obengenannten Anforderungen unterschiedliche Nachteile:

- Die Untersuchung mit einem Debugger ist einerseits äußerst zeitaufwendig, andererseits hängt die Qualität der Ergebnisse stark von der Kompetenz der untersuchenden Person ab.

- Tracing-Methoden haben den Vorteil, daß sie sich schnell durchführen lassen, allerdings liefern sie lediglich Informationen über das Verhalten eines Programms in konkret untersuchten Fällen. Insbesondere kann von den Ergebnissen eines Tracings nicht auf das in der Zukunft zu erwartende Verhalten eines Programms geschlossen werden.

- Eine Disassemblierung läßt sich zwar größtenteils automatisieren; um ein verständliches Assemblerlisting zu erhalten ist jedoch in der Regel eine umfangreiche manuelle Nachbearbeitung nötig. Ebenso wie bei der Untersuchung mit einem Debugger spielt auch hier bei der Auswertung, z. B. bei der Interpretation des Listings, die Analyseerfahrung der untersuchenden Person eine große Rolle.

Aufgrund der genannten Nachteile dieser drei üblicherweise in der Praxis eingesetzten Methoden ist eine automatisierte Dekompilierung interessant, die ein vorliegendes Binärprogramm in ein äquivalentes Programm in einer höheren Programmiersprache übersetzt. Dieser Prozeß ähnelt zwar dem der Disassemblierung, doch ist der erzeugte Programmcode wesentlich kürzer und leichter verständlich als das entsprechende Assemblerlisting. Man wird dabei zwar nicht den ursprünglichen Quellcode des Programms vollständig rekonstruieren können, jedoch stellt dies kein größeres Problem dar, solange der resultierende Hochsprachencode dem ursprünglichen Quellcode semantisch äquivalent ist.

Dekompilierung wird bisher nur äußerst selten eingesetzt, was hauptsächlich daran liegt, daß zur Zeit für die meisten relevanten Hardwareplattformen keine brauchbaren Werkzeuge (*Tools*) zur Verfügung stehen. Gründe hierfür sind sicherlich, daß einerseits der potentielle Markt für Decompiler relativ klein ist (ein Debugger hingegen wird von jedem professionellen Entwickler benötigt) und andererseits die Komplexität eines Decompilers und insbesondere die seiner theoretischen Grundlagen deutlich die Komplexität der anderen Reverse Engineering-Werkzeuge übersteigt. Eine Ausnahme hiervon bilden lediglich Spezialfälle wie etwa die Dekompilierung von Java-Bytecode [15, 23, 45, 49, 54] oder des in der .NET-Architektur von Microsoft verwendeten MSIL[1]-Codes [24, 31, 46] (s. a. Kap. 2.12).

In der Literatur [6, 11, 17, 25, 30, 56] ist Dekompilierung seit längerem bekannt, wodurch ein theoretisches Fundament existiert; so wurde z. B. in [11] die Möglichkeit einer Umsetzung eines solchen Ansatzes durch die Implementierung eines prototypischen Decompilers gezeigt, den die Entwickler als `dcc` bezeichneten. Dieser war jedoch auf i8086-Programme unter MS-DOS beschränkt und ist damit nunmehr obsolet.

[1] MSIL = Microsoft Intermediate Language

Aufgabe der vorliegenden Arbeit war es daher, zum einen die grundsätzliche Anwendbarkeit von Dekompilierungstechniken auf typische Malware zu untersuchen, zum anderen einen existierenden Decompiler so anzupassen, daß aktuelle Malware für die weitverbreitete 32-Bit-Windows Plattform (*Win32*) analysiert werden kann.

Hierfür wurde der Decompiler `dcc` [11] ausgewählt. Für ihn sprachen folgende Gründe:

- Der Quellcode von `dcc` ist frei unter der GNU GPL[2] verfügbar.
- `dcc` unterstützt den Befehlssatz des Intel i80286-Prozessors, so daß eine Grundlage für neuere 32-Bit-x86-Prozessoren besteht.
- Im Vergleich zu anderen Decompilern für 16-Bit-x86-Maschinencode erscheinen seine Konzepte und theoretischen Grundlagen wesentlich ausgereifter.

Im einzelnen waren für die Anpassung von `dcc` auf die Dekompilierung von Win32-Programmen folgende Erweiterungen nötig:

- Erkennung und Verarbeitung des für Win32-Programme verwendeten *Portable-Executable*-Dateiformats (*PE*-Dateiformats);
- Disassemblierung von 32-Bit-x86 Maschinenbefehlen;
- Anpassung an eine durchgängig 32-bittige Verarbeitung;
- *optional:* Erkennung des Startup-Codes verschiedener verbreiteter Win32-Compiler durch Signaturen;
- *optional:* Erkennung von Bibliotheksroutinen dieser Compiler durch Signaturen;
- *optional:* Verbesserung des Algorithmus zur Signaturerzeugung und -verwendung.

Die zum Verständnis dieser Arbeit notwendigen Fachausdrücke werden im folgenden Kapitel definiert und näher erläutert. In Kapitel 3 wird die Architektur des zugrundeliegenden Decompilers `dcc` näher beschrieben. Im Anschluß daran werden in Kapitel 4 eigene Untersuchungen an einer Auswahl von Win32-Malware im Hinblick auf die Anwendbarkeit der hier vorgestellten Dekompilierungstechniken näher ausgeführt. Die an `dcc` vorgenommenen Änderungen und die mit dem resultierenden Decompiler `ndcc` erzielten Ergebnisse werden in Kapitel 5 diskutiert und erläutert. In Kapitel 6 werden die erzielten Ergebnisse zusammengefaßt und ein Ausblick gegeben.

[2] GNU GPL = GNU General Public License

2 Grundlagen – Begriffsdefinitionen

In diesem Kapitel werden einige Begriffe eingeführt und erläutert, die für das Verständnis der Thematik notwendig sind. Einige weitere Begriffe und Abkürzungen, die keiner ausführlichen Definition bedürfen, sind im Glossar auf S. 71 bzw. im Abkürzungsverzeichnis auf S. 73 aufgeführt.

2.1 Forensische Informatik

Unter forensischer Informatik wird eine Reihe von Informatikmethoden verstanden, die dazu dienen können, computerbezogene Unfälle bzw. Verbrechen aufzuklären. Hierunter sind Vorgänge zu verstehen, bei deren Hergang Computer *als solche*, d.h. in ihrer spezifischen Eigenschaft als Computer und nicht lediglich als Gegenstände, eine Rolle spielen.

Die Methoden der forensischen Informatik beschränken sich häufig auf die Wiederherstellung von absichtlich gelöschten Daten zur Aufklärung von Computerkriminalität. Die forensische Informatik beinhaltet aber auch die Analyse von Hackertools und bösartiger Software zur Aufklärung von sicherheitsrelevanten Vorfällen [8].

2.2 Malware

Der Begriff *Malware* steht allgemein für bösartige Software (*malicious software*) [7]. Er dient als Oberbegriff für die verschiedenen Arten bösartiger Software, insbesondere Viren, Würmer, Trojanische Pferde und Hintertüren (*Backdoors*). Eine formale Definition des Begriffs Malware liefert [9]:

> „A software or module is called 'malicious' ('malware') if it is intentionally dysfunctional, and if there is sufficient evidence (e.g. by observation of behaviour at execution time) that dysfunctions may adversely influence the usage or the behaviour of the original software. “

2.3 Virus

Der Begriff des “Computer-Virus“ wurde erstmalig von Fred Cohen im Jahre 1984 wie folgt definiert [12]:

> „We define a computer ’virus’ as a program that can ’infect’ other programs by modifying them to include a possibly evolved copy of itself. With the infection property, a virus can spread throughout a computer system or network using the authorizations of every user using it to infect their programs. Every program that gets infected may also act as a virus and thus the infection grows.“

Ein Computer-Virus ist demzufolge ein Programm, das andere Programme infizieren kann und sich dadurch verbreitet. Diese Definition trifft heute immer noch zu; es sollte allerdings ergänzt werden, daß nicht nur in Binärdateien vorliegende ausführbare Programme befallen werden können, sondern auch z. B. Bootsektoren von Festplatten oder Office-Dokumente, in denen Makros enthalten sein können.

Formalisiert wird dies in einer neueren Definition [9]:

> „Any software that reproduces (or ’self-replicates’), possibly depending on specific conditions, at least in 2 consecutive steps upon at least one module each (called ’host’) on a single system on a given platform, is called a ’virus’ (for that platform). A viral host may be compiled (e.g. boot and file virus) or interpreted (e.g. script virus).“

Programmcode wird, sofern er die beschriebenen Eigenschaften besitzt, als *viral* bezeichnet.

2.4 Wurm

Als *Wurm* wird ein Programm bezeichnet, das sich verbreitet, indem es Kopien von sich selbst erzeugt, entweder auf lokalen Laufwerken oder über Netzwerke, worunter z. B. auch eine Ausbreitung über E-Mail fällt. Eine formale Definition, in der der Schwerpunkt auf die Netzwerkeigenschaften gelegt wird, liefert [9]:

> „Any software that reproduces (or ’propagates’), possibly depending on specific conditions, in at least two parts (nodes) of a connected system on a given platform is called a (platform) ’worm’, if it has the ability to communicate with other instances of itself, and if it is able to transfer itself to other parts of the network using the same platform.“

2.5 Trojanisches Pferd

Unter einem Trojanischen Pferd wird eine Software verstanden, in der zusätzlich zu den spezifizierten Funktionen weitere, verborgene Funktionen enthalten sind [9]:

> „A 'Trojan Horse' is a software or module that, in addition to its specified functions, has one or more additional hidden functions (called 'Trojanic functions') that are added to a given module in a contamination process ('trojanization') usually unobservable for a user. These hidden functions may activate depending upon specific (trigger) conditions.

Zumeist werden jedoch solche Programme als Trojanische Pferde bezeichnet, bei denen die verborgene Funktion (*Payload*) die einzige Funktionalität darstellt und die lediglich oberflächlich als nutzbringendes Programm getarnt werden; oft geschieht diese Tarnung lediglich durch die Benennung und meist rudimentäre Dokumentation.

2.6 CARO-Namenskonvention

1991 setzte sich auf einem CARO[1]-Treffen ein Komitee zusammen, um die bei der Benennung von Computerviren auftretende mangelnde Einheitlichkeit zu reduzieren; das Resultat dieser Bemühungen war die CARO-Namenskonvention [3].

Da im Laufe der Jahre viele neue Arten von Viren auftraten, wurde eine Erweiterung dieser Namenskonvention nötig, wie sie z. B. in der Computer-Makro-Virenliste [5] verwendet wird.

Das aktuelle Format für CARO-konforme Namen sieht wie folgt aus:
`Typ://Plattform/Stamm[.Variante][@M/MM]`

Typ: Dieses Element beschreibt die grundlegende Eigenschaft des bösartigen Programms. Die CARO-Konvention kennt die Kategorien *worm* (Wurm), *trojan* (Trojanisches Pferd), *virus*, *intended*, *joke*, *generator* und *dropper*.

Plattform: Unter "Plattform" versteht man die Soft- bzw. Hardwareplattform, auf der ein bösartiges Programm lauffähig ist. Dies kann ein Betriebssystem sein (wie z. B. *DOS* für MS-DOS und *W32* für 32-Bit-Windows), aber auch die Makrosprache eines Office-Pakets (z. B. steht *W97M* für die von Word 97 verwendete Makrosprache *Visual Basic for Applications*).

Stamm: der Name der Programmfamilie.

Variante: Falls mehrere Untertypen eines Programms existieren, werden diese mit Buchstaben- oder Nummernkombinationen ergänzt.

[1] CARO = *Computer Antivirus Researchers' Organization*

@M/MM: Diese Suffixe beziehen sich auf die Verbreitungsweise von Würmern. Sie sind für die Systematik nicht zwingend notwendig.

`@M` bedeutet, daß sich ein Wurm per E-Mail verbreitet. `@MM` bedeutet, daß es sich um einen *mass mailer* handelt, der sehr viele E-Mails auf einmal verschickt.

2.7 Reverse Engineering

Reverse Engineering wird im Kontext der Software-Entwicklung und -Wartung folgendermaßen definiert [10] :

> „Reverse engineering is the process of analyzing a subject system to
>
> - identify the system's components and their interrelationships and
> - create representations of the system in another form or at a higher level of abstraction."

Im vorliegenden Fall des Einsatzes von Reverse Engineering zu forensischen bzw. sicherheitsrelevanten Zwecken soll hierunter der Analysevorgang verstanden werden, in dem bestehende, binär vorliegende Programme untersucht werden, um ihre Funktionsweise verstehen und dokumentieren zu können. (Bei einer Sicherheitsanalyse kann durchaus auch bestehender Quellcode auf Sicherheitsmängel hin untersucht werden; dieser Fall wird hier allerdings nicht betrachtet.) Das Resultat dieser Analyse kann z. B. als Assemblerlisting, Darstellung in einer höheren Programmiersprache oder als natürlichsprachliche Verhaltensbeschreibung vorliegen.

Reverse Engineering bezieht sich hier, sofern nicht anders erläutert, speziell auf die Analyse von in binärer Form vorliegenden Computerprogrammen mit dem Ziel, eine Repräsentation auf einer höheren Abstraktionsebene zu gewinnen.

2.8 Quellcode

Unter dem Quellcode eines Programms oder Programmfragments versteht man dessen ursprüngliche Darstellung vor der Übersetzung in eine binäre Form. Der Quellcode kann in einer höheren Programmiersprache oder auch in Assembler (s. Kap. 2.9) verfaßt sein.

2.9 Assembler

Mit dem Begriff "Assembler" werden im Deutschen zwei unterschiedliche Dinge bezeichnet: Zum einen eine maschinenorientierte Programmiersprache, die exakt auf einen Prozessortyp bzw. eine Prozessorfamilie zugeschnitten ist und zum anderen

der zugehörige Übersetzer. Ein Assemblerprogramm besteht zum einen aus Pseudobefehlen, die Hinweise an den Übersetzer darstellen (z. B. Reservierung von Datenbereichen) und zum anderen aus sogenannten *Mnemonics* oder Assemblerbefehlen, die im Normalfall 1:1 in Maschinenbefehle übersetzt werden. Es kann vorkommen, daß es mehrere Assemblerbefehle gibt, die auf denselben Maschinenbefehl abgebildet werden; z. B. sind auf der x86-Architektur die Befehle `JNZ` und `JNE` äquivalent. Es können ebenfalls mehrere zulässige Maschinenkodierungen für einen Befehl existieren, z. B. kann der x86-Befehl `AND AX,0FFF0h` auf Maschinenebene als `25 F0 FF`, `81 E0 F0 FF` oder `83 E0 FF` abgebildet werden.

Ein Assemblerbefehl kann u. U. auch durch mehrere Maschinenbefehle dargestellt werden, d. h. es existiert auf Maschinenebene keine direkte Entsprechung, so daß der Befehl durch mehrere andere Befehle nachgebildet werden muß. Z.B. gibt es auf der SPARC-Architektur sog. synthetische Instruktionen, die durch den Assembler auf andere Instruktionen abgebildet werden. So ist dort u. a. der Befehl `set value,`$\texttt{reg}_{\texttt{rd}}$ zu der Befehlsfolge `sethi %hi(value),`$\texttt{reg}_{\texttt{rd}}$`; or `$\texttt{reg}_{\texttt{rd}}$`,%lo(value),`$\texttt{reg}_{\texttt{rd}}$ äquivalent.

2.10 Höhere Programmiersprachen

Unter höheren Programmiersprachen versteht man Sprachen, die eine gewisse Abstraktion von der Maschinenebene erlauben und die somit nicht in erster Linie maschinen-, sondern problemorientiert sind. Sie können daher nicht mehr direkt vom Prozessor ausgeführt werden, sondern müssen erst durch einen Compiler (s. Kap. 2.12) oder Interpreter übersetzt werden.

2.11 Objektcode

Der Maschinencode eines Programms, welcher in binärer Form vorliegt, wird als Objektcode bezeichnet. Er ist unabhängig von spezifischen Dateiformaten; insbesondere sollte dieser Begriff nicht mit dem der Objektdatei (s. Kap. 2.12) verwechselt werden. Zum Objektcode im engeren Sinne gehören daher auch keine weiteren Debug-, Struktur- und Bindungsinformationen, die keine Befehle an den Prozessor darstellen, sondern durch das Betriebssystem interpretiert werden.

2.12 Compiler

Als Compiler wird ein Werkzeug bezeichnet, das ein in einer höheren Programmiersprache vorliegendes Programm in eine maschinennähere Darstellungsform überführt; der zugehörige Prozeß wird als Kompilierung bezeichnet. Als Ziel einer solchen

Übersetzung können u. a. folgende Möglichkeiten gewählt werden:

Assemblerprogramm: Das Quellprogramm wird für einen konkreten Prozessor übersetzt, liegt allerdings noch nicht in binärer Form vor. Die Bezeichner und die Trennung in Code und Daten sind noch vorhanden, Typinformationen hingegen meist nur noch dann, wenn die entsprechenden Datentypen direkt vom Prozessor unterstützt werden.

Objektdatei: Das Quellprogramm wird in den Binärcode für einen spezifischen Prozessor übersetzt, die erzeugte Datei ist aber nicht eigenständig lauffähig. Eine Objektdatei muß erst noch mit anderen Objektdateien oder Bibliotheken zusammengebunden werden (*Linking*), um ein lauffähiges Programm zu erzeugen. In Objektdateien sind üblicherweise viele Strukturinformationen über das Programm enthalten, was eine Dekompilierung erleichtern kann.

Binärprogramm: Ein Binärprogramm ist eine fertig gebundene binäre Datei, die direkt ausführbar ist. Im weiteren Sinne umfaßt der Begriff auch alle Systemkomponenten wie Treiber, DLLs (Dynamic Link Libraries) etc., die nicht wie Programme eigenständig gestartet werden, die aber dennoch vom Prozessor ausführbaren Maschinencode enthalten. Je nach Betriebssystem wird das endgültige Binärprogramm beim Aufruf noch mit den von ihm benötigten Bibliotheken gebunden.

Wieviel Strukturinformation noch vorhanden ist, hängt allerdings 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[2]/PE[3], UNIX ELF[4]) können zudem noch Bezeichner und insbesondere Verknüpfungen mit Bibliotheken enthalten sein.

Die Auswertung dieser Strukturinformationen stellt bei einer Dekompilierung zwar einen gewissen Mehraufwand dar, doch können so in der Regel bessere Ergebnisse erzielt werden, da diese Informationen während des eigentlichen Dekompilierungsprozesses genutzt werden können.

P-Code: kurz für Pseudo-Code. Das Programm wird für eine virtuelle Maschine übersetzt, so daß der erzeugte Code später noch interpretiert werden muß, dies aber aufgrund der binären Darstellung relativ effizient geschehen kann. Vorteile von P-Code sind in der Regel Platzeffizienz und die Möglichkeit der Plattformunabhängigkeit. Heute wird P-Code z. B. bei Java, in der .NET-Architektur von Microsoft oder auch in einigen Versionen von Visual Basic verwendet.

[2]NE = New Executable
[3]PE = Portable Executable
[4]ELF = Enhanced Link Format

Da der verwendete P-Code meist stark auf die Bedürfnisse der zugrundeliegenden Programmiersprache zugeschnitten ist und sehr viele Strukturinformationen vorliegen, gestaltet sich die Entwicklung eines Decompilers für P-Code-Programme wesentlich einfacher als bei Binärprogrammen. Dies sieht man sowohl im Fall von Java als auch bei der .NET-Architektur von Microsoft, wo mit den bereits vorhandenen Decompilern (Java: [15, 23, 45, 49, 54], .NET MSIL: [24, 31, 46]) sehr gute Resultate erzielt wurden.

2.13 Disassemblierung

Disassemblierung bedeutet in der Grundform zunächst die Umsetzung von Objektcode in einzelne Assemblerbefehle; dieser Vorgang kann z. B. mit Hilfe einer Übersetzungstabelle erfolgen. Zu einer Disassemblierung nach dieser Definition ist bereits ein Debugger in der Lage; oft kann dieser zusätzlich symbolische Informationen anzeigen, welche zur Erleichterung der Fehlersuche vom Compiler mit abgelegt wurden.

Um ein verständliches Assemblerlisting zu erhalten, werden bereits viele Techniken benötigt, die sich auch in der Dekompilierung wiederfinden; der Aufwand ist also nicht unerheblich.

2.14 Dekompilierung

Eine Dekompilierung ist der einer Kompilierung entgegengesetzte Prozeß, in dem aus einem Assemblerprogramm, einer Objektdatei, einem Binärprogramm oder aus P-Code wieder eine hochsprachliche Darstellung gewonnen wird. Dies muß nicht notwendigerweise automatisiert, d. h. durch einen Decompiler geschehen, sondern kann auch manuell durchgeführt werden; hierfür ist dann allerdings ein Disassembler oder Debugger sinnvoll, da eine manuelle Disassemblierung sehr aufwendig und fehlerträchtig wäre.

Idealerweise ist der resultierende Programmtext mit dem ursprünglichen Quellcode identisch, wobei allerdings Informationen wie Bezeichner, Kommentare, Formatierung, Aufteilung auf mehrere Dateien etc. normalerweise bei der Kompilierung verlorengehen; lediglich in speziellen Fällen wie etwa Java oder .NET bleiben einige dieser Informationen erhalten.

Dieses Ergebnis kann natürlich schon allein deshalb nicht erreicht werden, da es für viele Operationen mehrere semantisch gleichwertige Konstrukte gibt, mit Hilfe derer sie ausgedrückt werden können, die vom Compiler aber identisch umgesetzt werden. Ein Beispiel: Eine Multiplikation mit 2^n wird von einem Compiler oft optimiert und durch eine Schiebeoperation um n Bits nach links ersetzt. Das Resultat läßt sich nicht mehr von einer bereits im Quelltext vorhandenen Schiebeoperation unterscheiden,

vorausgesetzt, dieses Sprachelement ist in der Quellsprache verfügbar.

Darüber hinaus besteht das Problem, daß nicht in allen Fällen bekannt ist, in welcher Quellsprache ein Programm ursprünglich geschrieben wurde, oder daß diese Sprache vom Analysten nicht ausreichend beherrscht wird, um das Resultat der Dekompilierung interpretieren zu können. Im Falle einer automatischen Dekompilierung kann es außerdem zu aufwendig sein, einen Decompiler zu schreiben, der die Quellsprache nicht nur in jedem Fall identifizieren, sondern auch Programmcode in genau dieser Quellsprache generieren kann. Es sollte jedoch zumindest für die gängigen prozeduralen Programmiersprachen möglich sein, unabhängig von der Quellsprache als Ziel stets dieselbe ausreichend universelle Sprache zu verwenden. Diese Wahl stellt nicht unbedingt eine Einschränkung dar, wenn man die hier untersuchten Einsatzzwecke in der forensischen Informatik betrachtet:

- Für eine Sicherheitsanalyse einer „normalen“ Software im Sinne einer Untersuchung auf absichtlich verborgene Dysfunktionalitäten wie etwa eine Trojanisierung ist die gewählte Sprache eher unwichtig, solange der Untersuchende sie beherrscht.

- Für eine weitergehende Sicherheitsanalyse in Hinblick auf unbeabsichtigte Safety- und/oder Security-relevante Fehler durch Implementationsmängel gilt die Sprachunabhängigkeit aufgrund sprachspezifischer sicherheitsrelevanter Eigenschaften nur eingeschränkt. (Beispiel: Die in C-Programmen auftretenden Buffer-Overflows sind in Sprachen wie Java [35, 48, 55] oder LISP [22, 27, 55] grundsätzlich nicht möglich.) Allerdings ist in solchen Fällen u. U. auch eine Disassemblierung angebracht, da derartige Sicherheitsmängel auch in Abhängigkeit vom Compiler (etwa bei bestimmten Optimierungsstufen) oder insbesondere von den Compiler-Bibliotheken auftreten können.

- Bei der Analyse bestimmter Arten von Malware, insbesondere von Viren, muß man sich teilweise ohnehin auf eine Disassemblierung beschränken, da diese oft in Assembler geschrieben werden. Wenn das zu untersuchende Programm in einer Hochsprache geschrieben wurde, spielt die Wahl der Zielsprache der Dekompilierung allerdings auch hier nur eine untergeordnete Rolle (s. a. Kap. 4).

3 Der Decompiler `dcc`

Die Basis für den im Rahmen dieser Arbeit implementierten Win32-Decompiler (vgl. Kap. 5) bildet der Decompiler `dcc`[11]. Im folgenden sollen daher die bestehende Architektur vorgestellt und ihre Beschränkungen dargelegt werden.

3.1 Zielsetzung

`dcc` ist als prototypischer Decompiler entwickelt worden, um die in [11] vorgestellten Dekompilierungstechniken zu implementieren und die erzielten Resultate zu untersuchen. Da keine kommerzielle Anwendung angestrebt wurde, sondern lediglich die Machbarkeit untersucht werden sollte, konnten bei der Entwicklung die Quell-Plattform und die Zielsprache beliebig gewählt werden. Als Eingabe verarbeitet `dcc` lediglich `.com` und `.exe`-Dateien, die unter DOS im *Real-Mode* auf Intel-Prozessoren lauffähig sind; dabei wird der Befehlssatz des Intel i80286 unterstützt. Als Ziel der Dekompilierung werden C-Programme erzeugt.

3.1.1 Konsequenzen der bei der Implementation getroffenen Entscheidungen

Die Wahl der Kombination DOS/i80286 für die Klasse der zu dekompilierenden Programme hatte einige direkte Auswirkungen auf die Entwicklung von `dcc`, sowohl in Bezug auf besonders zu berücksichtigende als auch auf fehlende und daher nicht zu behandelnde Eigenschaften.

- Da es sich bei dem i80286 um einen 16-Bit-Prozessor handelt, kann dieser nicht direkt mit 32-Bit-Werten umgehen. Aus diesem Grund verarbeiten Programme für diese CPU 32-Bit-Werte üblicherweise in mehreren Schritten, wobei die oberen und die unteren 16 Bit jeweils in einem 16-Bit-Register gespeichert werden. Um die im ursprünglichen Programm verwendeten 32-Bit-Variablen rekonstruieren zu können, ist daher ein nicht unerheblicher Aufwand nötig.

  Dies gilt analog auch für 64-Bit-Variablen auf einer 32-Bit-CPU. Diese werden jedoch wesentlich seltener verwendet, da 32 Bit für die meisten praktischen

Anwendungen ausreichen, so daß eine Behandlung dieses Spezialfalles dort weniger wichtig erscheint.

- Das Dateiformat von DOS-Programmdateien (insbesondere von `.com`-Dateien) ist im Vergleich zu den Programmformaten vieler anderer Betriebssysteme relativ einfach gehalten, so daß sich der Aufwand beim Laden des Programms in Grenzen hält. Andererseits sind in diesen Dateiformaten auch weniger Informationen über die Struktur des Programms enthalten, die bei einer Dekompilierung von Nutzen sein könnten.

- Da DOS keine dynamischen Bibliotheken unterstützt, müssen alle verwendeten Bibliotheksroutinen vom Compiler statisch in das Programm eingebunden werden. Dies hat zur Folge, daß der Erkennung von Bibliotheksroutinen mit Hilfe von Signaturen besondere Bedeutung zukommt; allerdings entfällt hierdurch die Notwendigkeit der Analyse von Funktionen, die aus dynamischen Bibliotheken importiert worden sind.

Die Wahl von `C` als Zielsprache ist insofern günstig, als daß die Zielsprache ausreichend flexibel sein muß, um die Kontrollstruktur und die zugrundeliegenden Operationen aller dekompilierten Programme darstellen zu können. Wie in Kapitel 3.3.3 näher ausgeführt werden wird, stellt es kein Problem dar, wenn das zu dekompilierende Programm ursprünglich in einer anderen Sprache geschrieben wurde; in diesem Fall ist lediglich keine Rekompilierung möglich, eine weitere Analyse hingegen schon. Zudem wäre es mit relativ geringem Aufwand möglich, eine andere Zielsprache zu implementieren.

Es wird für die Analyse allerdings davon ausgegangen, daß die Quellsprache über ähnlich mächtige Ausdrucksmittel wie `C` als eine prozedurale Programmiersprache verfügt; u. a. wird die Möglichkeit von selbstmodifizierendem Code ausgeschlossen. Eine Optimierung durch den Compiler hingegen wird explizit berücksichtigt, z. B. wird durch die Datenflußanalyse in Kapitel 3.3.2 selbst eine Art von Optimierung durchgeführt, wodurch lokale Compiler-Optimierungen teilweise nivelliert werden. Natürlich betrifft dies nicht alle Arten von Optimierungen, z. B. kann bei der Verwendung von `inline`-Funktionen nicht mehr rekonstruiert werden, daß es sich ursprünglich um eine Funktion gehandelt hat; es sollte allerdings in jedem Fall semantisch korrekter Code erzeugt werden.

3.2 Verwendete Techniken und Hilfswerkzeuge

Im folgenden werden einige in `dcc` verwendete Techniken sowie für die Erstellung von Bibliothekssignaturen und Prototyp-Informationen verwendete Hilfswerkzeuge dargestellt, die für das Verständnis der Architektur notwendig sind. Die Einbindung

dieser Werkzeuge in das Dekompilierungssystem wird in Abbildung 3.1 dargestellt. Dabei erfolgt die Verwendung der Signatur- und Prototyp-Informationen bereits in einer frühen Phase der Dekompilierung, damit Bibliotheks-Funktionen – wie in Kapitel 3.3.1 beschrieben – von einer weitergehenden Analyse ausgenommen werden können.

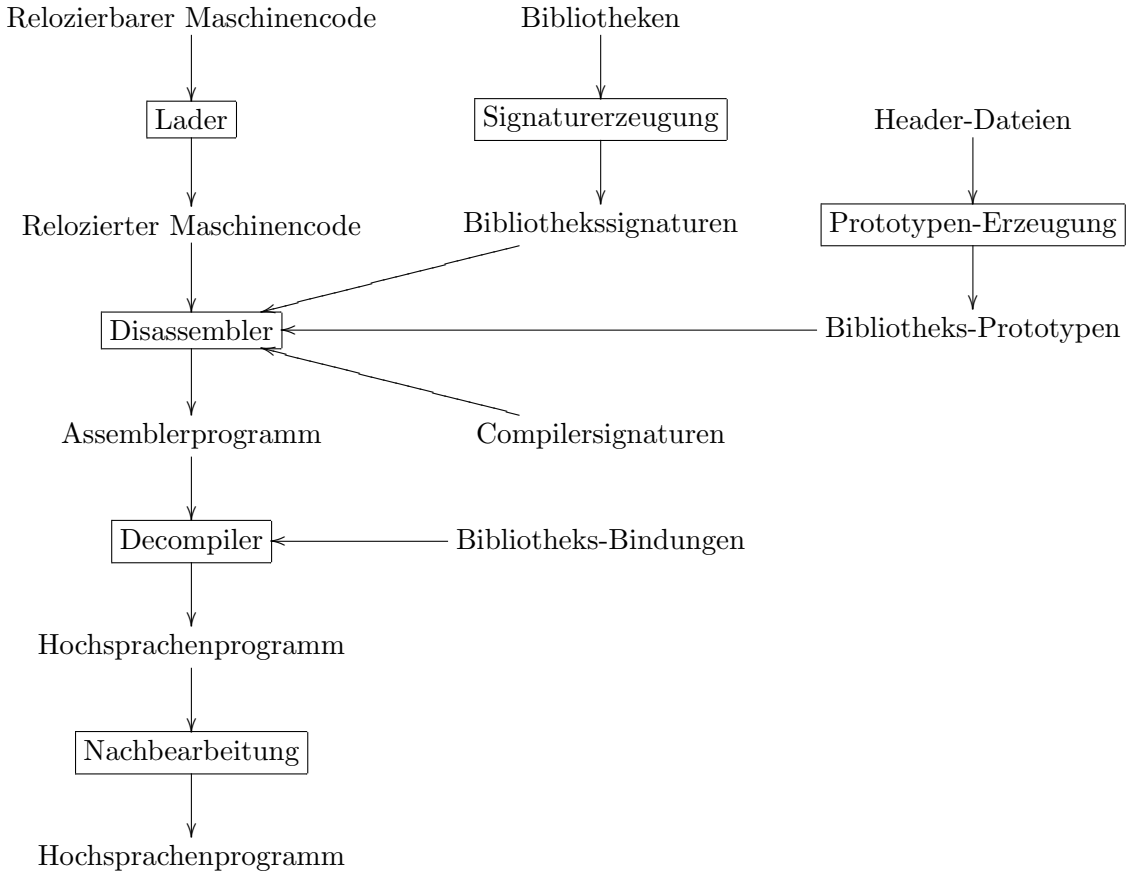

Abbildung 3.1: Dekompilierungssystem

3.2.1 Bibliotheks-Signaturen

Bei der Erzeugung von Binärprogrammen durch Compiler werden üblicherweise dem Compiler beiliegende oder von Drittherstellern gelieferte Laufzeitbibliotheken bzw. Teile davon in das Kompilat eingebunden. Dies gilt insbesondere für Plattformen, die kein dynamisches Binden beim Laden des Programms unterstützen, wie z. B.

das Betriebssystem MS-DOS, unter dem die von `dcc` dekompilierbaren Programme laufen.

In den meisten Fällen ist bei der Analyse eines Binärprogramms und insbesondere bei einer Dekompilierung allerdings nur derjenige Teil des Objektcodes von Interesse, der durch die Übersetzung des vom Programmierer geschriebenen Programmcodes resultiert. Ausnahmen hiervon treten lediglich dann auf, wenn beispielsweise ein Fehler in der Laufzeitbibliothek vermutet wird oder verschiedene, inkompatible Versionen einer Bibliothek verwendet wurden.

Eine Analyse der eingebundenen Laufzeitbibliotheken ist daher nur insofern erwünscht, als daß diese dazu dienen sollte, die ursprünglichen Bezeichner der aufgerufenen Funktionen zu ermitteln. Die Kenntnis des Funktionsbezeichners ist in diesem Fall für das Verständnis der Semantik eines Funktionsaufrufs ausreichend, solange gewährleistet ist, daß einer Funktion im Binärprogramm nur dann der Bezeichner einer Bibliotheksfunktion zugeordnet wird, wenn sie exakt mit dem Original übereinstimmt. Wie eine solche Funktion hingegen tatsächlich implementiert ist, ist für das Verständnis des resultierenden Quellprogramms in der Regel unerheblich.

Da im Binärprogramm die Funktionsbezeichner üblicherweise nicht mehr vorhanden sind, müssen sie über einen Vergleich des Binärcodes der Funktionen im kompilierten Programm mit dem Binärcode in den ursprünglichen Bibliotheken rekonstruiert werden. Es ist allerdings aus verschiedenen Gründen nicht sinnvoll, ein solches Verfahren direkt umzusetzen:

- Das binäre Abbild der ursprünglichen Funktion in der Bibliotheksdatei und ihrer Darstellung im Binärprogramm stimmen nicht unbedingt überein, da Offsets beim Binden des Binärprogramms verändert werden können.

- Aus Speicherplatzgründen sollten nicht die kompletten Bibliotheksdateien verwendet werden. Dies ist bereits bei der ausschließlichen Unterstützung von DOS-Compilern relevant, wenn mehrere Compiler und verschiedene Versionen jedes Compilers unterstützt werden sollen: Die bei Turbo C 2.01 mitgelieferten Bibliotheken umfassen insgesamt 748 kB, bei Turbo C++ 1.01 sind dies bereits 1.209 kB. Für Win32-Compiler liegen diese Zahlen noch um mehrere Zehnerpotenzen höher [28], so daß die vollständige Übernahme bei dem heutzutage typischerweise verfügbaren Speicherplatz zwar technisch machbar, aber nicht sinnvoll erscheint.

- Der Umfang des Codes in einer Bibliothek und die Anzahl der Funktionen machen ebenfalls ein effizientes Verfahren für die Erkennung der Bibliotheksfunktionen erforderlich. Eine reine *Brute-force*-Suche ist spätestens bei Win32-Programmen, bei deren Entwicklung Klassenbibliotheken verwendet wurden,

nicht zu vertreten, da hier leicht mehrere tausend Bibliotheksfunktionen vorkommen können [28].

- Aus Urheberrechtsgründen dürfen nicht die vollständigen Compiler-Bibliotheken mit dem Decompiler verbreitet werden.

Für die Erkennung von Bibliotheks-Routinen werden daher Bibliotheks-Signaturen verwendet; dabei stellt eine Signatur für eine Bibliotheksfunktion ein binäres Muster dar, das eine Funktion eindeutig unter allen Funktionen in ein und derselben Bibliothek identifiziert [11]. Dabei können Bibliothekssignaturen durchaus unterschiedlich konstruiert werden [17, 28]; im Extremfall degenerieren sie zur Binärdarstellung der Bibliotheksfunktionen selbst. Auch wenn Bibliothekssignaturen grundsätzlich ebenfalls manuell erzeugt werden können, so ist dies aufgrund der Anzahl der in einer Bibliotheksdatei vorhandenen Funktionen und der Vielzahl an zu unterstützenden Compilern nicht bzw. nur als unterstützende Maßnahme sinnvoll.

Im folgenden soll daher das automatisierte Verfahren für die Erzeugung der von `dcc` verwendeten Signaturen skizziert werden. Besonders zu beachten ist dabei, daß dasselbe Verfahren von `dcc` während der Laufzeit auf jede analysierte Funktion angewendet wird, um das Ergebnis anschließend mit den vorher generierten Bibliothekssignaturen zu vergleichen.

- Jede Signatur besteht grundsätzlich aus den ersten n Bytes einer Funktion. (Der konkret verwendete Wert für n wurde experimentell ermittelt.)

- Die Signatur wird nach dem ersten unbedingten Kontroll-Transfer (unbedingter Sprung oder Prozedur-Rücksprung) abgeschnitten, da nicht bekannt ist, ob nachfolgende Befehle noch zu der Funktion gehören bzw. später im Binärprogramm ebenfalls an dieser Stelle vorhanden sind.

- Ist die Signatur nun kürzer als n Bytes, so wird sie mit Null-Bytes aufgefüllt (*Padding*); der für das Auffüllen gewählte Wert ist dabei irrelevant und muß lediglich durchgehend verwendet werden.

- Da in Bibliotheksfunktionen vorkommende Offsets beim Binden verändert werden können und somit im Binärprogramm andere Werte als in der Bibliotheksdatei annehmen können, dürfen die betreffenden varianten Bytes nicht in die Signatur aufgenommen werden; dabei erfolgt die Bestimmung aller möglicherweise varianten Bytes durch eine Analyse des Maschinencodes der Funktion. Die entsprechenden Bytes werden in der Signatur durch ein *Wildcard* ersetzt; konkret wurde hierfür in `dcc` der Wert `0F4 hex` verwendet, da dieser dem selten benötigten Opcode `HLT` entspricht.

> Nach Meinung des Autors dieser Arbeit ist der gewählte Wert allerdings irrelevant, da sein Vorkommen in der Bibliotheks-Signatur – zumindest in der vorliegenden Implementation – nicht tatsächlich dazu führt, daß der Wert des Bytes an der entsprechenden Position der im Binärprogramm untersuchten Funktion ignoriert wird. Vielmehr wird die gewünschte Wildcard-Funktionalität dadurch erreicht, daß der Algorithmus bei der Anwendung auf eine Funktion im Binärprogramm *dieselben Bytes* durch den Wildcard-Wert ersetzt, so daß bei dem nachfolgenden Vergleich mit der ursprünglichen Signatur eine Übereinstimmung festgestellt wird.

Dabei können identische Signaturen entstehen, wenn die ursprünglichen Funktionen tatsächlich identisch implementiert waren oder durch das Verfahren soweit gekürzt und durch Wildcards verändert wurden, daß sie hinterher übereinstimmen. Bei identisch implementierten Funktionen lassen sich identische Signaturen natürlich nicht vermeiden. Für den Fall, daß trotz unterschiedlicher Implementation identische Signaturen erzeugt werden, wird in [11] vorgeschlagen, die betreffenden Signaturen manuell zu generieren. Es ist allerdings nicht plausibel, wie dies geschehen soll, da bei der Analyse durch `dcc` jede Funktion dem beschriebenen Algorithmus unterzogen wird, um die so erzeugte Signatur anschließend auf exakte Übereinstimmung mit einer der ursprünglichen Signaturen zu prüfen; jegliche abweichend erzeugte Bibliotheks-Signatur führt daher zwangsläufig dazu, daß keine Übereinstimmung festgestellt werden kann.

Um die Bibliotheksfunktionen in der Binärdatei später effizient identifizieren zu können, wird eine Hash-Funktion[1] generiert und die mit ihrer Hilfe aus den Signaturen erzeugte Hash-Tabelle zusammen mit den eigentlichen Signatur-Informationen in einer Signatur-Datei abgespeichert. Bei der späteren Identifikation der Bibliotheksfunktionen wird jede analysierte Funktion dem beschriebenen Algorithmus unterzogen und anschließend der Hash-Wert der so erzeugten Signatur bestimmt.

Die Verwendung einer Hash-Funktion dient dabei ausschließlich dazu, möglichst schnell die richtige Signatur zu finden; prinzipiell ließe sich das beschriebene Verfahren zur Signaturerzeugung auch mit einer binären Suche o. ä. verwenden. Der Zeitbedarf für die Erzeugung der Signatur und die Bestimmung des Hash-Werts beträgt in der vorliegenden Implementation $O(n)$ und hängt somit nicht von der Anzahl der in der Bibliothek vorkommenden Funktionen ab. Dabei ist der Zeitbedarf aufgrund des kleinen, konstanten Wertes für n und des geringen Berechnungsaufwandes zu vernachlässigen, da das Verfahren nur einmal für jede Subroutine des Binärprogramms durchgeführt wird. Da das Hash-Verfahren stets einen gültigen Index in die Hash-Tabelle liefert, wird abschließend überprüft, ob die Signatur der Funktion im

[1] Laut [17] handelt es sich dabei sogar um eine minimal perfekte Hash-Funktion, was den benötigten Speicherplatz verringert und eine Kollisionsbehandlung unnötig macht.

Binärprogramm mit der ursprünglichen Signatur übereinstimmt.

Wie nun zu erkennen ist, werden die Eigenschaften des oben beschriebenen Verfahrens, stets gleich lange Signaturen zu erzeugen und für eine Funktion im Binärprogramm dasselbe Bitmuster zu liefern wie für die entsprechende Funktion in der Bibliotheksdatei, durch das Hash-Verfahrens erforderlich. Bei Wegfall dieser Anforderungen durch Verwendung eines anderen Suchverfahrens wären auch andere Methoden zur Signaturerzeugung denkbar.

Sobald die syntaktische Analyse erstmalig auf eine Funktion stößt, wird überprüft, ob es sich bei dieser Funktion um eine Bibliotheksfunktion handelt. Wenn dies der Fall ist, kann die Funktion sofort klassifiziert werden und muß damit nicht weiter untersucht werden; in bestimmten Fällen muß die Funktion allerdings trotzdem analysiert werden (s. Kap. 3.3.1).

Werkzeuge zur Erzeugung der Signatur-Dateien: `makedsig`, `makedstp`

Um die Bibliotheks-Signaturen erzeugen zu können, liegen dem `dcc`-Paket zwei Werkzeuge bei, die die Generierung für verschiedene Typen von Bibliotheksdateien übernehmen können. Sie unterscheiden sich lediglich durch das unterstützte Dateiformat; das eigentliche zur Erzeugung der Signaturen verwendete Verfahren ist identisch und entspricht dem oben beschriebenen Algorithmus.

- `makedsig` ermöglicht die Erzeugung von Bibliotheks-Signaturen für die von verschiedenen C-Compilern unter MS-DOS verwendeten `.lib`-Dateien.
- `makedstp` unterstützt die von den Borland Turbo Pascal Compilern verwendete Datei `turbo.tpl` in den Versionen 4.0 und 5.0.

3.2.2 Compiler-Signaturen

Da üblicherweise jeder Compiler seine eigene Laufzeit-Bibliothek mitbringt, ist es notwendig, den für die Erzeugung eines Binärprogramms verwendeten Compiler zu identifizieren, um die passenden Bibliotheks-Signaturen auswählen zu können. Zudem ist der vom Compiler am Anfang eines Programms eingefügte Startup-Code, der verschiedene Initialisierungsaufgaben übernimmt, für die Dekompilierung nicht von Interesse; vielmehr ist es erwünscht, daß die eigentliche Analyse mit dem Hauptprogramm – im Falle von `C` mit der Funktion `main()` – beginnt.

Aus diesen Gründen werden in `dcc` Compiler-Signaturen verwendet, mit deren Hilfe sich folgende Informationen erschließen lassen:

- Der verwendete Compiler (einschließlich der Versionsnummer),
- das vom Compiler verwendete Speichermodell,

- die Adresse des Hauptprogramms bzw. der `main()`-Funktion und
- die Adresse des Datensegments.

Die Kenntnis des verwendeten Compilers und des von diesem verwendeten Speichermodells dienen dazu, die passende Sammlung von Bibliotheks-Signaturen auszuwählen. (DOS-Compiler, die mehrere Speichermodelle unterstützen, benötigen für jedes Speichermodell eine eigene Fassung der Bibliotheken.)

Die Anwendung der Compiler-Signaturen wird als erster Schritt nach dem Laden des Binärprogramms durchgeführt, da sie es ermöglicht, den Startup-Code von der Analyse auszunehmen und die Dekompilierung an der erhaltenen Adresse des Hauptprogramms zu beginnen.

Da der Startup-Code somit nicht näher vom Decompiler analysiert wird, ist zunächst nicht bekannt, welchen Wert das Register `DS`, das auf das Datensegment zeigt, zu Beginn des Hauptprogramms annimmt. Die Kenntnis der Lage des Datensegments ist jedoch in den meisten Fällen von essentieller Bedeutung für die weitere Analyse des Programms; daher wird die betreffende Adresse während der Anwendung der Compiler-Signaturen direkt aus dem Objektcode gewonnen. Dies wird dadurch ermöglicht, daß der betreffende Wert für eine konkrete Kombination von Compiler und Speichermodell stets an derselben Stelle des Startup-Codes zu finden ist, so daß er ohne eine Analyse des Maschinencodes extrahiert werden kann.

3.2.3 Prototypen

Die in Kapitel 3.2.1 beschriebene Identifikation der Bibliotheksfunktionen mit Hilfe von Signaturen liefert zunächst lediglich den Namen der jeweiligen Funktion, was bereits eine für die Lesbarkeit des erzeugten Hochsprachenprogramms sehr nützliche Information darstellt. Allerdings ist es für eine sinnvolle Dekompilierung ebenfalls erforderlich, Informationen über die von der Funktion akzeptierten Parameter zu erhalten: In den meisten Hochsprachen wird für die Parameterübergabe kein explizit dargestellter Stack verwendet. Somit muß eindeutig zugeordnet werden, welcher auf dem Stack abgelegte Wert welchem aktuellen Parameter bei einem Funktionsaufruf zuzuordnen ist.

Aus diesem Grund ist es sinnvoll, für die Dekompilierung zusätzlich über *Prototypen* für die in den Signaturen vorkommenden Bibliotheksfunktionen zu verfügen. Die hier verwendete Bedeutung des Begriffs *Prototyp* entstammt in erster Linie der C-Programmierung und umfaßt eine Deklaration einer Funktion, die zusätzlich zu ihrem Namen die Typen des von ihr zurückgegebenen Wertes und ihrer formalen Parameter beschreibt.

Die Zuordnung eines Prototypen zu einer Funktion erfolgt über den Namen der Funktion, setzt also eine erfolgreiche Identifizierung voraus. In Fällen, in denen für

eine erkannte Bibliotheksfunktion kein Prototyp vorliegt, wird die Funktion von dcc trotzdem analysiert, um auf diese Art Aufschluß über die Parameter zu erhalten.

Werkzeug zur Erzeugung von Prototypen für C-Headerdateien: parsehdr

Um die benötigten Prototyp-Informationen nicht erst während der Laufzeit von dcc erzeugen zu müssen, wurde das Tool parsehdr entwickelt.

parsehdr dient dazu, C-Headerdateien zu verarbeiten und die enthaltenen Prototypen so aufzubereiten, daß sie bei der Dekompilierung bereits in einer geeigneten Form, die unabhängig von der C-Repräsentation ist und sich effizient einlesen läßt, vorliegen.

In parsehdr wurde kein vollständiger C-Parser implementiert. Daher werden keine durch den Präprozessor verarbeiteten Makros und keine Typdefinitionen durch typedef unterstützt, sondern es werden lediglich Funktionsdeklarationen ausgewertet.

3.3 Architektur von dcc

Die folgende Darstellung der verschiedenen Dekompilierungsphasen basiert, sofern nicht anders vermerkt, auf [11]; eine graphische Übersicht der einzelnen Phasen ist in Abbildung 3.2 zu finden. Wie im Compilerbau üblich, wird auch bei dem vorliegenden Decompiler dcc 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 Prozessorarchitekturen und Programmiersprachen, wobei gegenüber Compilern Quelle und Ziel vertauscht sind.

3.3.1 Maschinen- und systemspezifische Phasen *(Frontend)*

Syntaktische Analyse

Die Aufgaben dieser Phase decken sich größtenteils mit denen eines Disassemblers. Im einzelnen findet hier das Laden des Binärprogramms, die Gruppierung von Code-Bytes zu Befehlen und die Trennung von Code und Daten statt.

Die Erkennung des Compiler und der Bibliotheks-Funktionen geschieht ebenfalls bereits in dieser Phase, da durch das Überspringen des Startup-Codes die Analyse direkt mit dem Hauptprogramm beginnen kann und Bibliotheksfunktionen ebenfalls von der Analyse ausgenommen werden können.

Falls für eine Funktion zwar eine Signatur, jedoch kein Prototyp vorliegt, so wird sie trotzdem analysiert, damit Informationen über die Parameter der Funktion gesammelt werden können. Dieser Fall kann selbst dann auftreten, wenn die Prototypen

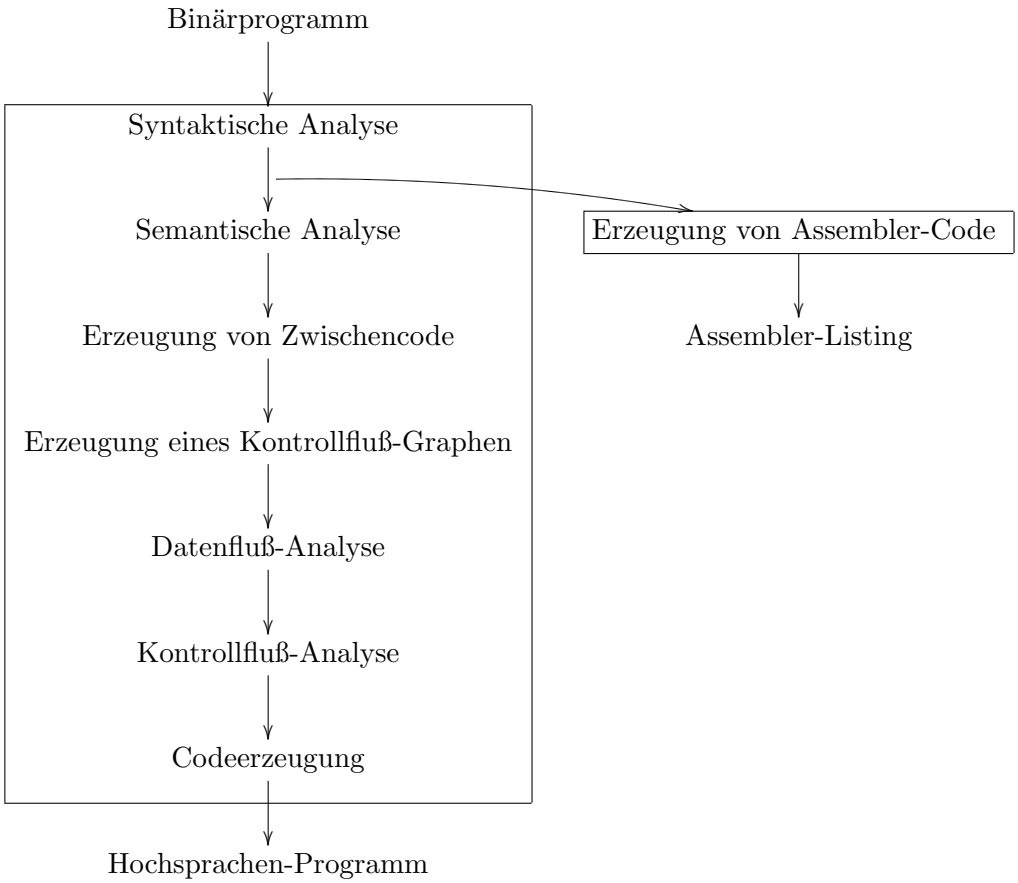

Abbildung 3.2: Architektur von `dcc`

aller *normalen* Bibliotheks-Funktionen bekannt sind, da der vom Compiler erzeugte Code teilweise Hilfsroutinen benötigt, die zwar in der Bibliotheksdatei vorhanden sind, die aber nicht für die explizite Verwendung in Benutzerprogrammen gedacht sind und für die keine Prototypen verfügbar sind.

(Beispielsweise verfügen 16-Bit-C-Compiler für DOS über Hilfsroutinen, die für einige Operationen auf 32-Bit-Zahlen verwendet werden, u. a. `LXMUL@`, `LDIV@`, `LMOD@`, `LXLSH@` und `LXRSH@`. `dcc` dekompiliert diese Routinen und erkennt ebenfalls, daß die Parameter an einige dieser Routinen in Registern übergeben werden.)

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 ebenfalls die Möglichkeit besteht, ein Assemblerlisting zu erzeugen (siehe Kapitel 5.2.1 für eine ausführliche Darstellung). Die 3-Adreß-Notation

dient dazu, auch bei 2-Adreß-Maschinen alle Parameter explizit zu nennen, da dort oft ein einziger Parameter gleichzeitig 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 geschieht mit Hilfe von *Idiomen*, d. h. Befehlsfolgen, die auf einer bestimmten Maschine häufig für bestimmte Aufgaben eingesetzt werden und deren hochsprachliches Äquivalent nicht ohne weiteres ersichtlich ist. Beispiele hierfür wären der Prozedur-Prolog, aus dem Informationen über die Parameter und lokalen Variablen der Funktion abgeleitet werden können, 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 von Zwischencode

Hier werden die gespeicherten Assemblerbefehle in eine Form überführt, die den Hochsprachen näher steht; anstelle der vielen einzelnen Opcodes gibt es jetzt nur noch die Kategorien *Zuweisung* (`HLI_ASSIGN`), *bedingter Sprung* (`HLL_JCOND`), *unbedingter Sprung* (dieser wird nicht als Operation kodiert, sondern implizit über eine Verknüpfung mit dem Folgebefehl dargestellt), *Subroutinenaufruf* (`HLI_CALL`), *Subroutinenrücksprung* (`HLI_RET`), *Push* (`HLI_PUSH`) und *Pop* (`HLI_POP`).

Ein arithmetischer Befehl nimmt also jetzt z. B. statt `add x,y` (in einer 3-Adreß-Notation `add x,x,y`) die Form `HLI_ASSIGN x,x+y` an, was der Operation `x:=x+y` entspricht.

Die Anzahl der Befehle wird dabei – bis auf die unbedingten Sprünge – 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 zunächst in Subroutinen aufgeteilt. Jede Subroutine wird dann weiter in *Basisblöcke* unterteilt, wobei ein Basisblock jeweils die maximale Folge von Instruktionen umfaßt, die genau

einen Eintritts- und einen Endpunkt hat und somit *immer* von Anfang bis Ende durchlaufen wird. Die Menge der Instruktionen eines Programms läßt sich daher eindeutig in eine Menge von zueinander disjunkten Basisblöcken aufteilen.

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

- Basisblock mit einem unbedingten Sprung am Ende: Der Basisblock hat eine ausgehende Kante.
- Basisblock mit einem bedingten Sprung am Ende: Der Basisblock hat zwei ausgehende Kanten.
- Basisblock mit einem indizierten Sprung[2] am Ende: Der Basisblock hat n ausgehende Kanten.
- Basisblock mit einem Subroutinenaufruf am Ende: Der Basisblock hat zwei ausgehende Kanten, nämlich zum Folgebefehl und zur Subroutine. Der Start eines neuen Basisblocks nach einem Subroutinenaufruf ist notwendig, da zwischen dem Subroutinenaufruf und dem nächsten Befehl weitere Befehle ausgeführt werden.
- Basisblock mit einem Subroutinenrücksprung am Ende: Der Basisblock hat keine ausgehenden Kanten. (Da die Funktion von mehreren Stellen aus aufgerufen werden kann, besteht keine eindeutige Beziehung.)
- Basisblock mit einem normalen Befehl am Ende: Die Abtrennung eines solchen Blocks ist notwendig, wenn die Adresse des nächsten Befehls ein Sprungziel darstellt. Die abgehende Kante führt daher zum sequentiell folgenden Basisblock.

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

3.3.2 Universeller Kern (UDM^3)

Datenfluß-Analyse

Durch die Zusammenfassung von mehreren Operationen können hochsprachliche Ausdrücke gebildet werden, wodurch die Darstellung des Programms verbessert

[2] In `C` entspricht dies einer switch()-Anweisung.
[3] UDM = Universal Decompilation Module

wird. In dem resultierenden Code werden – sofern nicht Variablen während ihrer gesamten Lebensdauer in Registern gehalten werden – keine Register mehr verwendet, und bedingte Sprungbefehle hängen nicht mehr von den Status-Flags des Prozessors ab, sondern enthalten den kompletten Vergleichsausdruck.

An dieser Stelle werden auch 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

Die bisherige Repräsentation des Programms benutzt für die Flußkontrolle keine Hochsprachenkonstrukte wie z. B. Schleifen und `if-then-else`-Anweisungen, sondern basiert noch auf den maschinenorientierten Sprunganweisungen. Da dieser `go-to`-Stil zwar in vielen Hochsprachen möglich, aber nicht wünschenswert[4] ist [16], werden in dieser Phase die Graph-Strukturen so weit wie möglich auf Hochsprachenkonstrukte abgebildet. Dabei wurden die folgenden Kontrollkonstrukte ausgewählt, die in den meisten prozeduralen Sprachen zur Verfügung stehen; zur Verdeutlichung ist jeweils das C-Äquivalent angegeben.

- Bedingte Ausführung: `if() ...;`
- Bedingte Verzweigung: `if() ...; else ...;`
- Mehrfachverzweigung: `switch() { case ...: ... }`
- Schleife mit Vorbedingung: `while() ...;`
- Schleife mit Nachbedingung: `do { ... } while();`
- Endlosschleife: `for(;;);`

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 [11] ausführlich dargestellt werden.

3.3.3 Sprachspezifische Endstufe *(Backend)*

Codeerzeugung

Da an diesem Punkt bis auf die Bezeichner alle notwendigen Informationen rekonstruiert worden sind, können die gespeicherten Strukturen ohne weitere Analyse in

[4] „A computer scientist is someone who, when told to “Go to Hell,” sees the “go to” rather than the destination as harmful.“ [21]

ein Hochsprachenprogramm überführt werden. Die Bezeichner werden dabei systematisch generiert, wobei durch ein Benennungsschema nach Funktionen (`procXX`), lokalen (`locXX`) und globalen Variablen (`byteXX` etc.) sowie Parametern (`argXX`) unterschieden wird; die Indizes werden dabei sequentiell hochgezählt. Die Codeerzeugung geschieht dabei rekursiv:

- Der Code für einen Basisblock kann sehr einfach erzeugt werden, da es sich hauptsächlich um Zuweisungsoperationen handelt. Da auch für Subroutinenaufrufe und Subroutinenrücksprünge direkte Hochsprachenäquivalente zur Verfügung stehen, stellen diese ebenfalls kein Problem dar.

- Bedingte oder unbedingte Sprünge werden nach Möglichkeit nicht im Hochsprachencode belassen, sondern es wird die in der Kontrollfluß-Analyse gewonnene Information über das jeweilige Konstrukt betrachtet. Wenn keine bekannte Kontrollstruktur erkannt wurde, muß dennoch ein `goto`-Statement verwendet werden; in diesem Fall wird ein Bezeichner dynamisch generiert und als Argument der `goto`-Anweisung verwendet sowie vor dem entsprechenden Sprungziel als Label eingefügt. Durch dieses Vorgehen wäre eine Dekompilierung in eine `goto`-lose Sprache in vielen Fällen nicht möglich.

  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, es wird sich aber nicht erneut kompilieren lassen.

Eine Lösung stellen Bibliotheks-Bindungen (*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 eine Abbildung der Bezeichner und der Aufruf-Syntax erfolgen kann. Dieses Verfahren ist jedoch in `dcc` noch nicht implementiert.

3.3.4 Nachbearbeitung

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

C durch break möglich ist. Ebenso sind häufig mehrere äquivalente Darstellungsmöglichkeiten vorhanden; so lassen sich etwa in C eine for- und eine while-Schleife mittels weniger zusätzlicher Statements ineinander umformen.

Ob diese Möglichkeiten im Quellcode des zu analysierenden Programms verwendet wurden, läßt sich üblicherweise nicht feststellen, da ein guter Compiler bei semantisch äquivalentem Quelltext identischen Binärcode erzeugen sollte. Zudem ist es gerade beabsichtigt, daß dcc geringe, semantisch nicht relevante Unterschiede im analysierten Objektcode ignoriert, da sich von diesen nicht eindeutig auf Unterschiede im ursprünglichen Quellcode schließen läßt.

Da sich die Lesbarkeit des erzeugten Codes durch die Verwendung weiterer Kontrollstrukturen aber deutlich erhöhen läßt, erscheint es sinnvoll, die vom Decompiler erzeugte Ausgabe in jedem Fall durch sprachspezifische Konstrukte zu verbessern.

Auch für den Fall, daß die ursprüngliche Quellsprache nicht über derartige Konstrukte verfügt, kann eine solche Optimierung vorgenommen werden; es ist lediglich erforderlich, daß diese Möglichkeiten in der Zielsprache des Decompilers zur Verfügung stehen.

Bisher ist in dcc keine Nachbearbeitungsphase implementiert worden.

4 Untersuchungen an ausgewählter Win32-Malware

Die in diesem Kapitel beschriebenen Untersuchungen dienen dazu, generell die Erfolgsaussichten bei einer automatischen Dekompilierung von Win32-Malware zu bestimmen. Die erzielten Ergebnisse wurden teilweise bei der Untersuchung der mit `ndcc` erzielten Resultate in Kapitel 5.2 verwendet, stellen jedoch auch ein eigenständiges Untersuchungsergebnis dar.

4.1 Vorüberlegungen

Die vorliegende Arbeit beschäftigt sich mit der Dekompilierung von Win32-Programmen, die aus reinem x86-Maschinencode bestehen und vorzugsweise von einem Compiler erzeugt wurden (s. a. *Win32-Viren* auf Seite 28). Unter Berücksichtigung der spezifischen Eigenschaften verschiedener Arten von Malware ergeben sich daher folgende Kategorien von auf der Win32-Plattform lauffähigen bösartigen Programmen, bei denen eine Dekompilierung nach der hier verfolgten Methodik aus verschiedenen Gründen nicht möglich bzw. nicht sinnvoll ist:

Skript-Malware: Hierzu zählen unter Win32 hauptsächlich Visual Basic Script (VBS) und JavaScript (JS). Da die Skripte im Klartext vorliegen und kein Objektcode vorhanden ist, ist eine Dekompilierung weder möglich noch notwendig.

Makro-Malware: Dies sind in den Makrosprachen von Office-Paketen oder anderen Programmen geschriebene Viren und andere Malware. Besonders häufig eingesetzt wird hierbei Visual Basic for Applications (VBA), das in den Office-Paketen der Firma Microsoft integriert ist.

Da die Makros nicht als x86-Code kompiliert, sondern lediglich verschleiert, als P-Code oder in Token-kodierter Form gespeichert werden, ist auch hier eine echte Dekompilierung nicht notwendig. Zudem existieren bereits entsprechende Werkzeuge, die den Makro-Code aus Dokumenten extrahieren können.

P-Code-Malware: Von den Sprachen, bei deren Kompilierung P-Code verwendet wird, sind unter Win32 in erster Linie Java, verschiedene der .NET-Architektur

zugehörige Sprachen (u. a. C#, VB.NET, Managed C++) und Visual Basic relevant.

Im Falle von Java-Kompilaten ist kein x86-Objektcode vorhanden. Zudem existieren bereits diverse Java-Decompiler [15, 23, 45, 49, 54].

Visual-Basic-Kompilate hingegen bestehen zwar zu einem kleinen Teil aus x86-Objektcode, jedoch muß der aus P-Code bestehende Hauptteil gesondert behandelt werden. Zu diesem Zweck existieren ebenfalls (zumindest für einige Compiler-Versionen) bereits Decompiler.

Für die in der .NET-Architektur erzeugten Kompilate gilt das gleiche wie für Visual Basic, es existieren verschiedene Decompiler [24, 31, 46].

Win32-Viren: Obwohl diese Viren aus x86-Objektcode bestehen, existieren hier verschiedene Besonderheiten, die einer Dekompilierung im Wege stehen:

- Viren werden häufig in Assembler geschrieben, um den Code kompakt und effizient zu halten und die vollständige Kontrolle über das Speicherlayout der Programmdatei zu haben. Eine Dekompilierung wird also nicht in jedem Fall möglich sein, da sich mit einem Assembler leicht Programme erzeugen lassen, die kein äquivalentes Hochsprachenprogramm besitzen. (Auch in diesen Fällen muß ein *funktional* äquivalentes Programm existieren. Ein solches kann aber nicht mehr automatisch von einem Decompiler gefunden werden.).

  Dieser charakteristische Unterschied zwischen viraler- und nicht-viraler Malware ist ebenfalls bei DOS-Programmen vorhanden, wie bereits in [4] für Trojanische Pferde ausgeführt wird: „The typical Trojan is a compiled program, written in a high-level language (usually C or Pascal, but we have also seen BASIC and even Modula-2) (, ...).“
- Hochsprachenprogramme bestehen außer dem vom Programmierer geschriebenen Code so gut wie immer zum Teil auch aus Bibliotheksroutinen, welche u. U. einen großen Anteil des Binärprogramms ausmachen können. Wenn dies bei der Entwicklung des Decompilers berücksichtigt wird (siehe auch Kapitel 3), kann die Qualität des Dekompilats durch die Verwendung der ursprünglichen Funktionsbezeichner wesentlich verbessert werden.

  Dies ist bei Assemblerprogrammen seltener der Fall, da diese – insbesondere bei der Programmierung von PCs und Workstations – oft ohne Verwendung von zusätzlichen Bibliotheken geschrieben werden. Selbst wenn eine Bibliothek eingebunden wurde, so muß sie nicht unbedingt zu den dem Decompiler bekannten Bibliotheken gehören, welche sinnvollerweise die Laufzeitbibliotheken der gängigsten Compiler umfassen.

- Ein Virus stellt kein isoliertes Programm dar, sondern einen Programmcode, welcher in ein Wirtsprogramm integriert ist. (Dieses Problem kann auch bei Würmern auftreten, siehe hierzu die Erläuterungen zu infizierten DLLs auf den Seiten 34 und 35). Wenn das infizierte Programm durch einen automatischen Decompiler analysiert wird, so wird auch der Code des Wirtsprogramms untersucht, was eher unerwünscht ist und sogar rechtlich unzulässig sein kann [34]. Zudem läßt sich an der hochsprachlichen Darstellung nur schwer erkennen, welche Teile zum Wirtsprogramm und welche zum Virus gehören; bei einer Disassemblierung wäre dies anhand der Struktur der Programmdatei möglich.
- Techniken zur Verhinderung der Erkennung und Analyse werden bei Viren wesentlich häufiger als bei anderer Malware eingesetzt. Einige davon zielen zwar ausschließlich auf Debugging ab [34], Maßnahmen wie Verschlüsselung oder Polymorphie betreffen jedoch auch die Dekompilierung, so daß diese nicht ohne manuelle Vorarbeit durchgeführt werden kann.
  Zudem werden bei Viren meist individuell programmierte Verschlüsselungs-Routinen eingesetzt, während bei nicht-viraler Malware oft bereits vorhandene Verschlüsselungswerkzeuge verwendet werden, wodurch die Verschlüsselung leichter automatisiert zu entfernen ist; siehe hierzu auch Kapitel 4.3.

DOS-Malware: Da der zugrundeliegende Decompiler `dcc` für die Dekompilierung von DOS-Programmen entwickelt wurde, läßt sich mit ihm grundsätzlich auch DOS-Malware analysieren, da, wie bereits dargestellt, auch Trojanische Pferde für DOS meist in Hochsprachen entwickelt werden [4]. Für DOS-Viren gelten zusätzlich die in Bezug auf Win32-Viren gemachten Aussagen (s. o.), so daß eine Dekompilierung für sie weniger in Frage kommt.

4.2 Auswahl und Identifikation der Win32-Malware

Aus der Malware-Datenbank des AVTC[1] wurden 105 Malware-Samples ausgewählt und untersucht, die den folgenden Anforderungen genügten:

- Es sollten ausschließlich Win32-PE-EXE-Dateien untersucht werden.
- Es sollten keine P-Code-Kompilate wie Java in der Stichprobe enthalten sein. (Visual Basic-Kompilate lassen sich nicht sofort erkennen und wurden in einem späteren Schritt ausgesondert. Für die nach dem Untersuchungszeitraum neu hinzugekommene Malware für .NET [43] hätte dasselbe gegolten.)

[1] AVTC = Anti-Virus Test Center, Arbeitsbereich AGN, Fachbereich Informatik, Universität Hamburg

- Dabei sollte ausschließlich nicht-virale Malware, d. h. Würmer und Trojanische Pferde, untersucht werden.

Die ausgewählten Dateien bilden nur einen kleinen Ausschnitt der existierenden Win32-Malware ab. Die Stichprobe ist jedoch mehr als ausreichend um festzustellen, ob der verfolgte Ansatz praktikabel ist, d. h. ob sich nicht-virale Win32-Malware mit den beabsichtigten Verfahren sinnvoll analysieren läßt.

Auf die Abgrenzung der DOS- von den Win32-EXE-Dateien soll an dieser Stelle nicht näher eingegangen werden; sie kann trivialerweise durch ein beliebiges Identifikationswerkzeug, wie z. B. den UNIX-Befehl `file`, erfolgen. Dies gilt ebenfalls für Java-Kompilate.

Die Unterscheidung zwischen viraler und nicht-viraler Malware erfolgte durch die Untersuchung mit mehreren Virenscannern. Es wurden nur Dateien in die Untersuchung aufgenommen, die mit Bezeichnungen wie *worm* oder *trojan* erkannt wurden oder deren fehlende Viralität sich durch weitere Quellen [2, 41] verifizieren ließ. Es wurden dabei die folgenden Virenscanner verwendet:

- AVP (AntiViral Toolkit Pro, Kaspersky) [36] Version 3.5.133.0,
- McAfee VirusScan [42], NAI, Version 5.20.1008,
- F-Prot Antivirus [26], FRISK Software International, Version 3.11a,
- Norton AntiVirus [50], Symantec, Version 7.07.23D.

Die exakten Versionsnummern der Virenscanner spielen dabei keine entscheidende Rolle, da die Virenscanner nur als Werkzeuge dienten und nicht selbst Gegenstand der Untersuchung waren; sie wurden allerdings der Vollständigkeit halber dokumentiert. Ebenfalls werden keine Statistiken über die Erkennung der Malware durch verschiedene Scanner aufgeführt, da die untersuchten Samples nicht repräsentativ sind und jegliche Aussage statistisch irrelevant wäre.

4.3 Analyse und Entfernung von Verschleierungsschichten

Als erster Schritt nach der Identifikation durch die obengenannten Virenscanner wurde, wie bereits beschrieben, die Vermutung überprüft, daß die untersuchten Dateien teilweise mit EXE-Kompressionsprogrammen verschleiert worden waren, um die Entdeckung und Analyse zu erschweren.

Verschleierungen werden bereits seit geraumer Zeit vorgenommen, wie die folgende Aussage belegt:

> „Another kind of packaging that modifies the initial file is to compress this file with one of the existing executable file compressors – LZEXE, PKLite, Diet, etc. ... Such compressed files have to be detected, recognized, and removed. ... Since the compression alters the image of the viruses present in the file, many scanners will not be able to detect the virus, even if it is known to them.“ [4]

Im folgenden soll der Begriff “Verschleierung“ für alle Arten von Kompression, Verschlüsselung und anderen Verfahren gebraucht werden, die den Inhalt einer Datei derart transformieren, daß er nicht mehr ohne weiteres zu interpretieren ist. Auch wenn es sich bei dem verwendeten Verfahren prinzipiell um eine echte Verschlüsselung handelt, spielt dies für die Analyse keine Rolle, da die Entschlüsselungsroutine und ggf. der zugehörige Schlüssel notwendigerweise im Programm enthalten sein müssen. Die tatsächlich geringere Länge der Programmdatei bei einer Kompression spielt im vorliegenden Kontext keine Rolle.

Zur Untersuchung der Verschleierungsschichten wurde das Programm GetTyp 2000 [29] verwendet, das fast alle gängigen EXE-Kompressions- und -Verschlüsselungsprogramme identifizieren kann. Um eine weitere Analyse der vorliegenden Malware-Samples zu ermöglichen, mußten ggf. vorhandene Verschleierungsschichten entfernt werden, da die verfolgte Dekompilierungsmethodik den Programmcode nicht ausführt und der Maschinencode daher unverschleiert vorliegen muß. Diese Entschleierung gelang bei einem Teil der untersuchten Programme durch verschiedene Werkzeuge, die teilweise bereits zusätzlich zu GetTyp bei der Identifizierung der jeweiligen Verschleierung verwendet worden waren. Die Programmdateien, bei denen eine Entschleierung auf diese Weise nicht möglich war, hätten zweifellos manuell entschleiert werden können, bzw. es wäre möglich gewesen, für diese Fälle spezielle Entschleierungswerkzeuge zu entwickeln. Da eine vollständige Entschleierung aller Samples aber nicht das Ziel der vorliegenden Arbeit ist, sondern vielmehr die prinzipielle Anwendbarkeit der Dekompilierungsmethodik gezeigt werden sollte, wird an dieser Stelle darauf verzichtet.

Konkret ergaben sich bei den untersuchten Samples folgende Resultate bezüglich des Einsatzes von EXE-Packern und der Möglichkeit, die Verschleierung zu entfernen:

ASPack: Alle 45 mit ASPack [1] verschleierten Samples konnten ohne weiteres durch UnASPack entpackt werden.

(Für UnASPack konnte keine URL des ursprünglichen Programmierers gefunden werden; dies ist in der Hacker-/Crackerszene nicht selten der Fall. Falls im folgenden auch bei anderen Entpackern keine Quellen angegeben sind, gilt entsprechendes.)

Neolite: Für Neolite [40] konnte kein spezifisches von dritter Seite entwickeltes Entschleierungsprogramm gefunden werden. Es konnte jedoch eines der vier Samples mit NeoLite selbst entpackt werden, welches Programme entpacken kann, die nicht mit der maximalen Kompressionsstufe gepackt wurden.

PECompact: Es existieren zwar verschiedene Entpacker (PEunCompact v0.01, tNO-Peunc v1.5 Beta, UnPECompact 1.31 [51]) für PECompact [13], allerdings konnte keiner von ihnen eines der vorliegenden 10 Samples entpacken.

Petite: Der Petite [37]-spezifische Entpacker Enlarge 1.3 [18] konnte das vorliegende Sample nicht entpacken, da er offenbar nur neuere Versionen von Petite unterstützt.

UPX: Hierbei handelt es sich um ein Open Source-Packprogramm. Da UPX [44] zumindest in neueren Versionen selbst als Entpacker dienen kann, existiert kein unabhängiger Entpacker. Es konnte nur eines der drei Samples entpackt werden, da die beiden anderen mit älteren UPX-Versionen erzeugt worden waren.

Zusätzlich existiert in manchen Fällen das Problem, daß Malware durch modifizierte Versionen von UPX komprimiert wurde, so daß sie nicht durch die normale Version entpackt werden kann [47]. (Die Modifikation von UPX ist aufgrund des vorliegenden Quellcodes leicht möglich.)

WWPack32: Der spezifische Entpacker Un-WWPACK/32 stürzte bei allen fünf mit WWPack32 gepackten [53] Samples ab; mit dem generischen Entpacker Win32Intro [19] gelang es allerdings in vier Fällen, die Dateien zu entpacken.

(Win32Intro hätte laut Spezifikation ebenfalls bei den anderen beschriebenen Packern anwendbar sein sollen, in der Praxis funktionierte dies aber nicht. Dies gilt ebenfalls für den generischen Entpacker **Procdump**.)

Auch ohne daß versucht wurde, den Aufwand bei der Entfernung von Verschleierungsschichten näher abzuschätzen, erscheint dieser doch aufgrund der beschriebenen Erfahrungen nicht unerheblich. Die Entwicklung eines eigenen Entschleierungstools im Rahmen dieser Arbeit wäre somit zwar prinzipiell möglich gewesen, jedoch hätte der damit verbundene Aufwand in keinem Verhältnis zum Nutzen gestanden.

4.4 Identifizierung der verwendeten Compiler

Die interessanteste Information bei der Untersuchung der Malware ist der jeweils zur Erzeugung verwendete Compiler. Hieran läßt sich ablesen, ob wie vermutet Trojaner und Würmer tatsächlich häufig in Hochsprachen programmiert sind, was eine

automatisierte Dekompilierung aussichtsreich erscheinen läßt, oder ob sie durch Assembler oder durch nicht identifizierbare Compiler erzeugt wurden, was eher für eine Disassemblierung sprechen würde. Ebenfalls von Interesse war die Vielfalt der verwendeten Compiler, da für die Berücksichtigung jedes weiteren Compilers bzw. jeder weiteren Version eines Compilers im Decompiler ein (allerdings vertretbarer) Aufwand nötig ist.

Nachdem gegebenenfalls eine vorhandene Verschleierung entfernt worden war (s. o.), welche den Startup-Code des Compilers versteckt und somit die Erkennung des Compilers verhindert hätte, wurde der Compiler mit dem bereits beschrieben Tool "GetTyp" automatisch identifiziert; in Zweifelsfällen wurden weitere Tools eingesetzt bzw. eine manuelle Inspektion durchgeführt.

4.5 Erläuterung der Ergebnistabelle

Die Resultate der bisher beschriebenen Untersuchungen finden sich in der Tabelle im Anhang A, welche im folgenden erläutert werden soll. Eine zusammenfassende Diskussion schließt sich an.

Pfad + Dateiname: Diese Angaben lassen zwar bereits Rückschlüsse auf die Klassifizierung der Malware zu; sie sollen hier aber lediglich der Reidentifizierbarkeit der einzelnen Samples dienen.

CARO-Name: Da zumindest bei der vorliegenden nicht-viralen Malware keiner der verwendeten Virenscanner (s. Kap. 11) in der Lage war, der CARO-Konvention (s. Kap. 2.6) entsprechende Namen auszugeben, mußten die Bezeichnungen aus der Ausgabe der Scanner abgeleitet werden. In den meisten Fällen wurde hierbei die von AVP vergebene, an die CARO-Konvention angepaßte Bezeichnung verwendet, da dieser Scanner als einziger alle Samples erkannte und exakt identifizierte. Von diesem Schema wurde abgewichen, wenn die anderen Scanner mehrheitlich eine andere Bezeichnung verwendeten; außerdem wurden von anderen Scannern gefundene Untertypen in die Tabelle integriert, wenn dadurch ein Sample genauer klassifiziert werden konnte.

AVP, NAI: Dies sind die von AVP und NAI/McAfee VirusScan ausgegebenen Bezeichnungen; die Ergebnisse von F-Prot und Norton AntiVirus wurden nicht wiedergegeben. Ausschlaggebend für diese Auswahl war, daß die beiden letzteren Scanner sehr viele Samples entweder nur generisch als bösartige Software erkannten (F-Prot) oder die Erkennungsrate sehr niedrig lag (Norton); die Ergebnisse flossen aber, wie bereits beschrieben, trotzdem in die Bestimmung der CARO-Namen ein.

Compiler: Bezeichnet den von GetTyp 2000 ausgegebenen Compiler, der zur Übersetzung des Programms eingesetzt wurde, wobei in Zweifelsfällen weitere Tools zur Anwendung kamen. Zusätzlich zu den Compilern verschiedener Hersteller finden sich folgende Einträge:

N/A: [2] Dies bedeutet, daß der Compiler nicht identifizierbar war, da das Sample nicht entpackt werden konnte.

n. i.: [3] In diesen Fällen fand sich selbst nach manueller Analyse kein Hinweis auf die Verwendung eines Compilers, so daß davon auszugehen ist, daß das Programm in Assembler programmiert wurde.

Infizierte DLL: Der Verbreitungsmechanismus einiger Würmer beruht auf der Modifikation von bestimmten System-Bibliotheken; insbesondere die `WINSOCK32.DLL` ist hiervon häufiger betroffen.

Da in diesem Fall der Malware-Code mit dem ursprünglichem Code der DLL vermischt ist, läßt sich der ursprüngliche Compiler nicht mit den normalerweise verwendeten Methoden identifizieren, die hauptsächlich den – vom Wurm nicht modifizierten – Startup-Code betrachten. Daher melden die verwendeten Werkzeuge denjenigen Compiler, der zur Erzeugung der ursprünglichen DLL eingesetzt wurde, was nicht erwünscht ist.

Für diese Fälle wurde daher unter der Rubrik *Compiler* die Bemerkung *infizierte DLL* eingetragen, um anzudeuten, daß nicht festgestellt werden konnte, mit welchem Compiler der in der analysierten DLL enthaltene bösartige Code erzeugt wurde.

Packer: Ebenfalls von GetTyp 2000 wurde der zur Verschleierung des Programms verwendete Packer ausgegeben. Eine nähere Beschreibung der Packer findet sich in Kapitel 4.3. In einem Fall, in dem der Packer als *unbekannt* bezeichnet wurde, ließ sich durch den Einsatz von GetTyp und anderen Werkzeugen kein Packer identifizieren, eine manuelle Inspektion wies aber auf eine Verschleierung hin. Es ist daher von einer individuell programmierten Verschleierungsroutine auszugehen; eine automatische Entschleierung konnte somit nicht stattfinden.

4.6 Zusammenfassung der Resultate

Als Fazit dieser Untersuchungen ergibt sich, daß die Mehrzahl der Samples, wie erwartet, verschleiert vorlag, sich die Verschleierung aber in vielen Fällen entfernen

[2] N/A = *not available*, nicht verfügbar
[3] n. i. = nicht identifizierbar

ließ, so daß der größere Teil der Samples (74 von 105) einer Analyse prinzipiell zugänglich ist. Wenn man berücksichtigt, daß weitere 24 Samples sich wahrscheinlich mit entsprechend hohem Aufwand ebenfalls entpacken lassen würden, erhöht sich diese Zahl sogar auf bis zu 98 Samples, was eine automatisierte Dekompilierung sehr vielversprechend erscheinen läßt.

Kategorie	Anzahl der Samples
Dekompilierung möglich	
Nicht gepackt, Compiler identifiziert	23
Entpackbar, Compiler identifiziert	51
Zwischensumme	74
Dekompilierung potentiell möglich	
Nicht entpackbar	24
Summe potentiell dekompilierbarer Samples	98
Dekompilierung nicht möglich	
Visual Basic-Programme	3
(Mutmaßliche) Assembler-Programme	2
Infizierte DLLs	2
Gesamtsumme: Untersuchte Samples	105

Tabelle 4.1: Statistik über identifizierte Compiler und Packer

Bei den übrigen Samples wäre eine Dekompilierung nach der verfolgten Methodik entweder gar nicht oder nur mit schlechten Resultaten möglich. Die Gründe hierfür wurden teilweise bereits erläutert:

Visual Basic-Programme: Der vom Visual Basic-Compiler eingesetzte P-Code wird vom Decompiler nicht verarbeitet; hierfür wäre eine tiefgreifende Erweiterung des Decompilers notwendig.

Assembler-Programme: Die von einem Assembler erzeugten Programme enthalten üblicherweise keine Bibliotheksroutinen, deren Erkennung zur Dekompilierung beitragen könnte.

Infizierte DLL: Bei den infizierten DLLs würde im erzeugten Dekompilat der vom Wurm integrierte Code mit dem ursprünglichen Bibliothekscode vermischt sein, so daß er sich nicht automatisch isolieren ließe. Im Gegensatz zu einer Infektion durch einen Virus, die üblicherweise am Anfang eines Programms erfolgt, kann der Code hier zudem in eine beliebige Funktion des Programms integriert werden.

Außerdem ist der zusätzliche Code üblicherweise von einem anderen Compiler erzeugt worden als die modifizierte DLL, so daß er nicht auf dieselben Compiler-Bibliotheken zurückgreifen kann. Abhängig davon, wie der Programmierer des Wurms dieses Problem gelöst hat, treten weitere Schwierigkeiten bei der Dekompilierung auf:

- Falls der integrierte Code vom Programmierer mittels eines Assemblers erzeugt wurde, kann dieser so implementiert worden sein, daß er nicht auf externe Bibliotheksroutinen angewiesen ist. In diesem Fall treten bei der Dekompilierung ähnliche Probleme wie bei einem reinen Assembler-Programm auf.

  Es ist allerdings auch bei der Erzeugung mit einem Compiler mit einigem Aufwand möglich, auf das Einbinden von Compiler-Bibliotheken zu verzichten; in diesem Fall treten keine *zusätzlichen* Probleme bei der Dekompilierung auf.
- Die benötigten Bibliotheksroutinen können auch zusammen mit dem bösartigen Code in die DLL eingefügt worden sein; dies ist allerdings u. a. aufgrund des Relokations-Aufwands und des zusätzlich benötigten Platzes weniger wahrscheinlich. Ist dies jedoch der Fall, so werden die zum Wurm gehörigen Compiler-Bibliotheksroutinen höchstwahrscheinlich vom Decompiler nicht identifiziert und müssen somit ebenfalls dekompiliert werden, was zu einer schlechten Qualität des resultierenden Quellcodes führt.

5 Anpassung von `dcc` an Win32-Programme: `ndcc`

Wie bereits beschrieben, ist der Decompiler `dcc` lediglich in der Lage, unter DOS lauffähige `.com` und `.exe`-Dateien zu dekompilieren. In diesem Kapitel werden die Änderungen beschrieben, die der Verfasser an `dcc` vorgenommenen hat; das resultierende Programm wurde mit ***`ndcc`*** bezeichnet.

Dabei wurde in erster Linie das Ziel verfolgt, `dcc` spezifisch um die für die Analyse von Win32-Programmen notwendigen Eigenschaften zu erweitern. Andere Mängel von `dcc` wurden größtenteils nicht berücksichtigt, so daß `ndcc` erwartungsgemäß wie bereits `dcc` ein prototypischer Decompiler ist, an dem die prinzipielle Machbarkeit demonstriert werden soll.

Auch wenn dies nicht im Quellcode vermerkt ist, ist `dcc` nach Aussage der Autoren Open Source Software und unterliegt der GNU GPL. Nach den Lizenzbestimmungen unterliegt `ndcc` damit ebenfalls der GPL.

Die Entwicklung von `ndcc` fand in erster Linie mit dem GNU-C++-Compiler (*GCC/G++*) unter Linux/x86 statt. Kompatibilitätstests wurden zusätzlich ebenfalls mit GCC unter Win32/x86 und Solaris/SPARC durchgeführt. Bei einer Übersetzung mit anderen C++-Compilern sind keine Schwierigkeiten zu erwarten, da bereits die Entwicklung von `dcc` auf verschiedenen Plattformen – ursprünglich unter DEC Ultrix, später u. a. mit 16-Bit-DOS-Compilern – stattfand.

5.1 Änderungen

5.1.1 Disassemblierung des Intel i80386-Befehlssatzes

Um den in Win32-Programmen verwendeten Maschinencode disassemblieren zu können, müssen einige Unterschiede zu den bisher von `dcc` unterstützten 16-Bit-Befehlen des Intel i80286 berücksichtigt werden, die im folgenden vorgestellt werden. Für eine Einführung in den i80386-Befehlssatz sei an dieser Stelle z. B. auf [32, 38] verwiesen.

- Der i80386-Prozessor verfügt im Vergleich zum i80286 über einige zusätzliche Opcodes, die zum Teil neue Funktionen beinhalten (`LFS`, `BTR`) und zum Teil

erweiterte Möglichkeiten für die Wahl der Operanden bieten (IMUL, bedingter Sprung mit 16/32-Bit-Displacement).

In der aktuellen Version von ndcc werden die meisten der häufiger verwendeten zusätzlichen i80386-Opcodes unterstützt. Dabei war als Besonderheit zu berücksichtigen, daß die neuen Befehle des i80386 fast alle nur durch 2-Byte-Opcodes zu erreichen sind, d. h. ein erstes Byte mit dem Wert 0F hex zeigt an, daß ein zweites Byte mit dem eigentlichen Opcode folgt. Hierdurch wurden gewisse Änderungen in der syntaktischen Analysephase notwendig: Die Disassemblierung von Maschinenbefehlen wurde in dcc in erster Linie durch *eine* Tabelle gesteuert, in der unter dem durch den jeweiligen Opcode adressierten Eintrag die zur Disassemblierung notwendigen Informationen über den betreffenden Maschinenbefehl zu finden sind.

- In dem unter Win32 normalerweise verwendeten 32-Bit-*Protected-Mode* werden ohne besondere Vorkehrungen nur noch 8- und 32-Bit-Daten als Operanden verarbeitet, statt wie bisher 8- und 16-Bit-Daten; eine Verwendung von 16-Bit-Daten ist durch das Voranstellen des *Data-Size*-Präfix-Bytes 66 hex aber weiterhin möglich.

  ndcc unterstützt die Verarbeitung von 32-Bit-Daten als Operanden vollständig. Die gemischte Verwendung von 16- und 32-Bit-Befehlen durch entsprechende Präfixe ist dabei sowohl im 32- wie auch im 16-Bit-Modus möglich; im letzteren Fall kehrt sich dabei die Bedeutung des *Data-Size*-Präfix-Bytes um.

  Entscheidend waren insbesondere diejenigen Änderungen am Disassembler, die *Immediate*-Operanden betrafen, da die Disassemblierung sonst nicht korrekt beim nächsten Befehl fortgesetzt werden würde, sondern ggf. zwei Bytes zu früh. Die Verarbeitung der tatsächlichen 32-Bit-Werte stellte hingegen kein Problem dar, da hierfür lediglich die Typen der verwendeten Variablen angepaßt werden mußten. Zusätzlich waren noch Anpassungen für diejenigen Befehle notwendig, die auf Speicherwörter zugreifen.

- Im 32-Bit-Protected-Mode arbeiten x86-CPUs mit 32-Bit-Adressen. Für die Auswertung der in den Maschinenbefehlen vorkommenden 32-Bit-Offsets gilt ähnliches wie das für 32-Bit-Daten gesagte; allerdings ist hier für die Umschaltung zwischen 16- und 32-Bit-Adressierung das *Address-Size*-Präfix-Byte 67 hex zuständig.

  Gravierender als die bloße Verwendung von 32 Bits für die Adreß-Offsets sind die an die 32-Bit-Adressierung gekoppelten neuen Adressierungsmodi des i80386. Im 16-Bit-Modus sind die für die indizierte Adressierung zur Verfügung stehenden Register sehr eingeschränkt; im einzelnen sind lediglich die

folgenden acht Kombinationen möglich, zu denen jeweils noch ein konstantes Displacement addiert werden kann: `[BX+SI]`, `[BX+DI]`, `[BP+SI]`, `[BP+DI]`, `[SI]`, `[DI]`, `[BP]`, `[BX]`. Die neuen Adressierungsmodi sind dagegen wesentlich flexibler; fast ohne Ausnahme läßt sich eine Adresse durch die Addition zweier beliebiger Register erzeugen, von denen eines zusätzlich noch mit dem Faktor 2, 4 oder 8 multipliziert werden kann.

Die 16-Bit-Adressierungsmodi werden von `dcc` nicht sofort bei der Disassemblierung in ihre Komponenten aufgespalten und somit orthogonal dargestellt; stattdessen bleibt jede der acht zulässigen Kombinationen als *eigene* Adressierungsweise stehen und wird an verschiedenen Stellen im Programm jeweils als Einheit behandelt.

Möglicherweise konnte der Quellcode von `dcc` durch diese Vorgehensweise kompakter oder besser verständlich geschrieben werden. Jedoch führt sie auf jeden Fall dazu, daß sich die Abhängigkeit von der konkreten Darstellung der Maschinenbefehle erhöht und die Implementierung weiterer Adressierungsmodi erschwert wird. Trotzdem wurde es im Rahmen der Diplomarbeit vorgezogen, die 32-Bit-Adressierungsmodi zwar vollständig zu analysieren, sie aber – soweit möglich – auf die vorhandenen acht Kombinationen abzubilden, da nicht genau ersichtlich war, an welchen Stellen in `ndcc` noch Abhängigkeiten bestanden, und dieses Verfahren außerdem für das in Kapitel 5.2.2 untersuchte Programm ausreichend war.

- Da der Adreßraum sich durch die 32-Bit-Adressierung vollständig mittels der Allzweck- und Adreß-Register abdecken läßt, werden die Segmentregister in modernen Betriebssystemen meist nicht mehr bzw. nicht für Anwenderprogramme sichtbar verwendet. Es kann somit aus der Sicht eines Anwenderprogramms davon ausgegangen werden, daß die Segmentregister alle auf denselben Speicherbereich zeigen und sich während der Laufzeit nicht ändern. Dieses Verfahren wird als *flaches* Speichermodell bezeichnet und vereinfacht die Programmierung in vielen Bereichen.

  Hierdurch vereinfacht sich auch die Dekompilierung von Win32-Programmen, da die Inhalte der Segmentregister für die Analyse nicht beachtet werden müssen; bei der Dekompilierung von 16-Bit-Programmen ist es hingegen von entscheidender Bedeutung, stets darüber informiert zu sein, welchen Wert die Segmentregister haben, da sonst nicht festgestellt werden kann, auf welche Speicheradressen tatsächlich zugegriffen wird.

  In `ndcc` ist dies so implementiert worden, daß bei Win32-Programmen davon ausgegangen wird, daß die Segmentregister stets den Wert 0 haben und für die Adreßbildung nicht herangezogen werden. Beachtet werden muß dieses in

erster Linie zu Beginn der Analyse, wenn die Segmentregister beim Laden des Programms initialisiert werden.

Wie oben beschrieben, fanden viele der entscheidenden Änderungen an ndcc im Disassemblierungsteil in der syntaktischen Analysephase statt. Weiterhin waren Änderungen an verschiedenen Teilen von ndcc notwendig; da diese Änderungen aber weit im Programm verstreut liegen und oft nur wenige Zeilen umfassen, lassen sie sich nur schwer zusammenfassend beschreiben.

Im Zuge der vorgenommenen Änderungen stellte sich heraus, daß die bestehende Architektur von dcc grundsätzlich für die Verarbeitung von 32-Bit-Befehlen geeignet war. Allerdings wurden teilweise in späteren Analysephasen implizite Annahmen über Einschränkungen im Befehlssatz des i80286 – z. B. über die für einen Befehl zulässigen Operanden – getroffen, die die Implementierung einiger Befehle des i80386 erschwerten.

5.1.2 Verarbeitung des Win32-PE-Dateiformats

Laden der PE-Datei

Das Laden einer Win32-PE-Datei unterscheidet sich signifikant von dem Laden von DOS-Programmdateien; diese Unterschiede im Ladeverfahren sollen im folgenden erläutert werden.

Da Programme unter DOS an verschiedene Segment-Adressen geladen werden können, wurde dcc so konstruiert, daß sie so weit unten im Speicher wie möglich plaziert werden; die im folgenden verwendeten Adressen sind dabei relativ zu dem von dcc für das Programm reservierten Speicherbereich zu verstehen.

DOS .com-Datei: Der Ladevorgang ist relativ trivial; es muß lediglich berücksichtigt werden, daß das Binärprogramm im Speicher ab der Adresse 0000:0100 hex liegen muß.

DOS .exe-Datei: Hier müssen einige Header-Informationen ausgewertet werden. Dabei sind die Größe des zu ladenden Programms und die Werte einiger Register am Programmstart relativ einfach zu bestimmen.

Da vor dem geladenen Programm Platz für den *PSP*[1] bleiben muß (dies ist bei .com-Dateien von vornherein durch das Laden ab dem Offset 0100 hex garantiert), wird das Programm ab der Adresse 0010:0000 hex geladen. Dabei ist zu berücksichtigen, daß die Programm-Header nicht in den Adreßraum des untersuchten Programms geladen werden, sondern daß an der genannten

[1] PSP = Program Segment Prefix

Adresse lediglich das *Image*, also der eigentliche Objektcode des Programms, liegt.

Da eine `.exe`-Datei ohne weitere Maßnahmen nur dann ausgeführt werden kann, wenn sie ab der Adresse `0000:0000 hex` geladen wird (dieser Fall kann allerdings unter DOS grundsätzlich nicht auftreten), müssen zunächst die im Programm verwendeten Segmentadressen angepaßt werden; dies geschieht mittels der Informationen in der Relokations-Tabelle. Anhand der Relokations-Tabelle kann zusätzlich festgestellt werden, bei welchen Konstanten im Programm es sich um Segmentadressen handelt. Diese Information kann dazu verwendet werden herauszufinden, bei welchen Variablen des Programms es sich um Zeiger handelt.

Win32 `.exe`-Datei: Im Gegensatz zu DOS `.exe`-Dateien liegen die Header-Informationen in PE-Dateien nicht an konstanten Offsets, sondern ihre jeweilige Position muß durch die Verfolgung mehrerer Zeiger bestimmt werden; u. a. hat der DOS-*Stub* eine variable Größe.

Eine Relokation muß nicht vorgenommen werden, da PE-Dateien bereits auf den im Header vorgegebenen Wert von *Image Base* angepaßt sind. Eine Auswertung der Relokations-Informationen zu anderen Zwecken könnte unter Umständen nützlich sein; dies würde jedoch umfangreiche Änderungen an verschiedenen Teilen von `ndcc` erfordern, da Zeiger unter Win32 nicht mehr als 16-Bit-Segment und 16-Bit-Offset, sondern als lineare 32-Bit-Zeiger verwaltet werden.

Ein Problem tritt dadurch auf, daß *Image Base* sehr große Werte annehmen kann, so daß sich der Ansatz verbietet, wie bei DOS-Programmen einen Speicherbereich zu allozieren und das Programm-Image an die entsprechende Startadresse (in diesem Fall *Image Base*) zu laden; in `ndcc` wird das Image daher an den Beginn des allozierten Speicherbereiches geladen. Um die notwendige Verschiebung nicht bei jedem Zugriff auf das Image berücksichtigen zu müssen und die in `dcc` verwendeten Typen beibehalten zu können, erfolgt der Zugriff in `ndcc` über einen Zeiger, der um *Image Base* nach unten verschoben ist, so daß bei einem Zugriff die resultierende Adresse wieder im Adreßbereich des Image liegt.

Im Gegensatz zu DOS-`.exe`-Dateien wird die komplette PE-Datei einschließlich des Headers in den Speicher geladen, was eine gewisse Vereinfachung beim Ladevorgang darstellt. Allerdings muß berücksichtigt werden, daß im PE-Header zwei verschiedene Werte für das *Alignment*, dem die einzelnen Abschnitte des Programms unterliegen, vorliegen können; beispielsweise sind unter Windows 95 ausschließlich Programme lauffähig, deren Abschnitte in der PE-Datei ein Alignment von 512 Bytes und im Hauptspeicher ein Alignment von 4096

Bytes einhalten. Daher muß jeder Abschnitt der PE-Datei einzeln in den Speicher geladen werden; die korrekten Speicheradressen liegen dabei bereits im PE-Header vor, so daß sie nicht erneut aus der Alignment-Information berechnet werden müssen.

Die für das Laden der Win32-PE-Datei zuständige Funktion `LoadPEImage` ist in Listing 5.1 abgebildet; dabei werden aus Gründen der Übersichtlichkeit einige Abfragen zur Fehlerbehandlung an dieser Stelle nicht dargestellt, die in der tatsächlichen Implementation vorhanden sind. Bei Aufruf dieser Funktion ist die Untersuchung des DOS-Stubs bereits abgeschlossen, die Analyse beginnt mit dem PE-Header. Die Funktionen `LH` und `LHD` dienen dazu, 16- bzw. 32-Bit-Werte *Endianess*-unabhängig aus dem Speicher zu lesen, da `ndcc` nicht notwendigerweise auf einer CPU läuft, die wie die x86-CPUs mit einer *Little-Endian*-Darstellung arbeitet; die Wirksamkeit dieser Maßnahme wurde – wie bereits erwähnt – unter Solaris/SPARC getestet.

Listing 5.1: Laden einer Win32-PE-Datei

```
void LoadPEImage(const char *filename, FILE *fp, dword newheader)
{
    fseek(fp, newheader, SEEK_SET);        // offset of new signature
    fread(&pe_header, 1, sizeof(pe_header), fp);      // Read PE header
    word mach = LH(&pe_header.Machine);            // CPU type
    if (mach<0x14c || mach>0x14e) fatalError(UNK_MACH, mach);
    fread(&pe_opt_header, 1, sizeof(pe_opt_header), fp); // Read optional header
    if (LH(&pe_opt_header.Magic) != 0x010b)
        fatalError(CORR_FILE, "pe_opt_header.Magic", filename);
    prog.initIP = LHD(&pe_opt_header.AddressOfEntryPoint)
                + LHD(&pe_opt_header.ImageBase);
    prog.initCS = 0;
    prog.ImageBase = LHD(&pe_opt_header.ImageBase);

    prog.NumberOfSections = LH(&pe_header.NumberOfSections);
    SectionHeaders = (PIMAGE_SECTION_HEADER) allocMem(
        prog.NumberOfSections * sizeof(IMAGE_SECTION_HEADER));
    fread(SectionHeaders, sizeof(IMAGE_SECTION_HEADER),
            prog.NumberOfSections, fp);
    prog.cbImage =
        LHD(&SectionHeaders[prog.NumberOfSections-1].VirtualAddress)
      + LHD(&SectionHeaders[prog.NumberOfSections-1].SizeOfRawData);
    /* Size of file image in memory = RVA of last section
```

```
    (sections are sorted by address) + Size of last section */

    prog.initSP = prog.ImageBase + prog.cbImage
                + LHD(&pe_opt_header.SizeOfStackReserve) - 4;
    prog.initSS = 0;

    prog.origImage = (byte*) memset(allocMem(prog.cbImage), 0, prog.cbImage);
    prog.Image = prog.origImage - prog.ImageBase;
    // Now read prog.origImage
    fseek(fp , 0, SEEK_SET);
    dword soh = LHD(&pe_opt_header.SizeOfHeaders);
    fread(prog.origImage, 1, soh, fp) != soh); // Read all headers again
    for (int i = 0; i<prog.NumberOfSections; i++) { // Read all sections
        dword prd = LHD(&SectionHeaders[i].PointerToRawData);
        if (prd != 0) {
            fseek(fp, prd, SEEK_SET);
            dword sord = LHD(&SectionHeaders[i].SizeOfRawData);
            fread(&prog.origImage[LHD(&SectionHeaders[i].VirtualAddress)],
                    1, sord, fp);
        }
    }
    prog.cReloc = 0; // Relocation info is not used yet

    /* Set up memory map */
    dword cb = (prog.cbImage + 3) / 4;
    prog.map = (byte *)memset(allocMem(cb), BM_UNKNOWN, cb);
}
```

Import von DLL-Funktionen

Der Zugriff auf in DLLs enthaltene Funktionen wird unter Win32 u. a. durch das in PE-Dateien enthaltene *Import Directory*, eine komplexe Datenstruktur, unterstützt.

Alle Verweise oder Zeiger, von denen im folgenden die Rede sein wird, sind *RVA*s[2], d. h. 32-Bit-Werte, die relativ zur Anfangsadresse der PE-Datei im Speicher zu beziehen sind. Je nach den im Header genannten Alignment-Anforderungen stimmen die RVAs nicht zwangsläufig mit der vom Anfang der PE-Datei gezählten Position überein. Da das PE-Dateiformat sehr komplexe Möglichkeiten für den Import von Funktionen erlaubt, werden an dieser Stelle nur die gängigsten davon vereinfacht

[2] RVA = Relative Virtual Address

dargestellt; eine detailliertere Darstellung findet sich z. B. in [39].

Die oberste Ebene des *Import Directory*s besteht dabei aus einem Array von *Image Import Descriptor*s, von denen jeder für die aus einer einzelnen DLL importierten Funktionen zuständig ist. Jeder *Image Import Descriptor* enthält dabei u. a. folgende Einträge:

Name: Ein Verweis auf den Namen der jeweiligen DLL.

OriginalFirstThunk: Ein Verweis auf ein Null-terminiertes Array von *IMAGE_THUNK_DATA*s. Dies sind im Normalfall RVAs, die auf *IMAGE_IMPORT_BY_NAME*-Einträge zeigen; falls allerdings das oberste Bit (*IMAGE_ORDINAL_FLAG*) gesetzt ist, wird die jeweilige Funktion nur über ihren Index importiert, welcher sich in den unteren 16 Bit findet. (Jede DLL exportiert ihre Funktionen zum einen über deren Namen und zum anderen über einen als *Ordinal* bezeichneten Index; die Nutzung der letzteren Möglichkeit zum Import einer Funktion kann allerdings zu Kompatibilitätsproblemen zwischen verschiedenen Versionen einer DLL führen.)

Ein *IMAGE_IMPORT_BY_NAME*-Eintrag besteht aus einem 16-Bit-Wert und dem Null-terminierten Namen der Funktion. Der 16-Bit-Wert sollte dabei, wenn möglich, dem *Ordinal* der jeweiligen Funktion entsprechen; dies ist allerdings nicht unbedingt erforderlich und dient lediglich der Performanz.

FirstThunk: Ebenfalls ein Verweis auf ein Null-terminiertes Array von *IMAGE_THUNK_DATA*s; in vielen Fällen ist dies eine Kopie des von *OriginalFirstThunk* referenzierten Arrays. Während der Laufzeit finden sich hier die tatsächlichen Adressen der importierten Funktionen.

Die beiden Arrays, auf die *OriginalFirstThunk* und *FirstThunk* verweisen, laufen parallel, d. h. beim Laden des Programms ermittelt das Betriebssystem für jeden Eintrag im *OriginalFirstThunk*-Array die Adresse der dort beschriebenen Funktion und schreibt diesen Wert in den entsprechenden Eintrag im *FirstThunk*-Array.

Der Aufruf der importierten Funktion durch das Programm kann somit durch einen indirekten Funktionsaufruf erfolgen. Wenn bei der Entwicklung des Programms nicht explizit spezifiziert wurde, daß es sich um eine aus einer DLL importierte Funktion handelt, und der Compiler daher einen direkten Funktionsaufruf generiert hat, so fügt der Linker beim Erzeugen der PE-Datei eine *Stub*-Funktion in das Programm ein, die lediglich einen indirekten Sprung zu der importierten Funktion enthält, und setzt als Ziel des Funktionsaufrufs die Adresse der Stub-Funktion ein.

Die Verarbeitung der Strukturen des Import Directories findet in der Funktion `ImportFuncs` statt, die in Listing 5.2 abgebildet ist. Die *Import Stubs* lassen sich nicht

auf eine ähnliche Weise direkt behandeln; sie werden vom Linker zwar üblicherweise in einem zusammenhängenden Speicherbereich, der *Transfer Area*, abgelegt, jedoch existieren keine weiteren Strukturen, über die ein Zugriff erfolgen könnte. Aus diesem Grund findet die Erkennung der *Import Stubs* an einer anderen Stelle vor der Erkennung von Bibliotheksfunktionen durch Signaturen statt. Die Implementation der entsprechenden Heuristik ist in Listing 5.3 abgebildet.

Listing 5.2: Auswertung der Importtabellen

```
void ImportFuncs()
{
    if (!prog.f32bit) return;   // Only for Win32 PE−files

    dword imp_desc = LHD(&pe_opt_header.DataDirectory
        [IMAGE_DIRECTORY_ENTRY_IMPORT].VirtualAddress);
    // Address of Win32 PE file import section

    while (LHD(&prog.origImage[imp_desc]))
        // List of import descriptors is zero−terminated in OriginalFirstThunk
    {
        char dllname[24];
        char *cp = (char *) memccpy(dllname,
            prog.origImage + LHD(&prog.origImage[imp_desc+12]),
            '.', sizeof(dllname));
        if (cp) *(cp−1) = 0;    // Ensure dllname is zero terminated
        else    dllname[sizeof(dllname)−1] = 0;

        dword ofthunk = LHD(&prog.origImage[imp_desc]);
        // OriginalFirstThunk;
        dword fthunk = LHD(&prog.origImage[imp_desc+16]);
        // FirstThunk;
        while (dword nameaddr = LHD(&prog.origImage[ofthunk]))
            // List of thunks is zero−terminated
        {
            dword funcaddr = LHD(&prog.origImage[fthunk]);

            // Create and insert new proc entry
            PPROC p = new PROCEDURE;
            p = (PPROC)memset(p, 0, sizeof(PROCEDURE));
            // Initialize , zero name
            pLastProc−>next = p;
```

```
            p->prev = pLastProc;
            p->procEntry = funcaddr;
            pLastProc = p;
            p->flg |= PROC_ISLIB | CALL_PASCAL;
            // Not really PASCAL but STDCALL calling convention

            if (nameaddr & IMAGE_ORDINAL_FLAG) {
                snprintf(p->name, V_SYMLEN,
                        "%s.%04Xh", dllname, nameaddr & 0xffff);
            }   // Ordinal import, name unknown -> "dllname.ordinal"
            else {
                strncpy(p->name, (char *)(prog.origImage+nameaddr+2),
                    sizeof(p->name)-1);
                ProtoCheck(p); // insert prototype information if available
            }

            ofthunk += 4;  fthunk += 4;    // Next thunk
        }
        imp_desc+= sizeof(IMAGE_IMPORT_DESCRIPTOR);
    }   // Next import descriptor
}
```

5.1.3 Weitere Anpassungen, Probleme bei der Implementierung

Unter den an `ndcc` vollzogenen Änderungen waren auch einige, die nicht direkt mit der Anpassung an Win32-Programme zu tun hatten. Dabei wurden kleinere Fehler behoben und generell der Code bereinigt, beispielsweise durch die Verwendung von `const`-Definitionen oder `inline`-Funktionen anstelle von `#define`-Anweisungen, wo immer dies im Programm möglich war. Dies führte u. a. durch die zusätzliche Typsicherheit wiederum zu der Entdeckung einiger weiterer Problemstellen.

Im folgenden sollen einige weitere Maschinencode-spezifische Probleme beschrieben werden, die nicht erst durch die zusätzlichen Befehle des i80386 verursacht wurden, sondern bereits beim i80186 bestanden. (Der i80186 war der Vorläufer des i80286 und wurde kaum in PCs eingesetzt. Er verfügte bereits über alle zusätzlichen Befehle des i80286, abgesehen von denen für die Steuerung des Protected Mode.)

Behandlung von `PUSH` *`immediate`*

Bei der Überprüfung der Funktionalität von `ndcc` anhand des in Kapitel 5.2.1 beschriebenen Testprogramms wurde zunächst festgestellt, daß zwar ein korrektes As-

Listing 5.3: Erkennung von *Import Stubs*

```
if (prog.Image[ fileOffset ] == 0xff && prog.Image[fileOffset+1] == 0x25)
{   // Indirect jump
    Int target , targeti ;
    target = LHD(&prog.Image[fileOffset+2]);
    targeti = LHD(&prog.Image[target]);

    PPROC p, pPrev;
    /* Search procedure list for one with appropriate entry point */
    for (p = pProcList; p && p->procEntry != targeti; p = p->next)
        pPrev = p;
    if (p && p->procEntry == targeti) strcpy(pProc->name, p->name);
    // Trampoline jump to imported function

    if (ProtoCheck(pProc) == NIL)
        /* Have a symbol for it , but does not appear in a header file .
            Treat it as if it is not a library function */
        pProc->flg |= PROC_RUNTIME; /* => is a runtime routine */
}
```

semblerlisting erzeugt wurde, die Dekompilierung jedoch später abbrach. Dies war darauf zurückzuführen, daß `PUSH` *`immediate`*-Befehle (diese dienen dazu, einen konstanten Wert auf den Stack zu legen) zwar in der Disassemblierungsphase erkannt wurden, jedoch waren bei der Entwicklung von `dcc` einige Annahmen getroffen wurden, die dazu führten, daß diese Befehle nicht korrekt in den Zwischencode übertragen werden konnten:

- Bei einem `PUSH`-Befehl mit einem konstanten Operanden wurde vermerkt, daß dieser Befehl *keine* Operanden hätte (dies macht in `dcc` für die Verarbeitung von anderen Befehlen mit ausschließlich konstanten Operanden durchaus Sinn, wie z. B. bei `ENTER`, `RET` oder Sprungbefehlen), worauf bei der Übersetzung in Zwischencode als Operand des `PUSH`-Befehls eine ungültige Referenz eingetragen wurde. Nach einer Korrektur ergaben sich allerdings weitere Schwierigkeiten:

- Bei `PUSH`-Befehlen wird analog zu `POP`-Befehlen der Operand so behandelt, als ob er der Ziel-Operand des Befehls wäre. Diese Betrachtungsweise ist (im Gegensatz zu `POP`-Befehlen) nicht anschaulich, da als eigentliches Ziel der Stack

dient.

- Bei der Umwandlung von Maschinenbefehlen in Zwischencode wird davon ausgegangen, daß, wenn ein Befehl nur einen Operanden hat, dies auf jeden Fall ein Zieloperand ist. Dies wäre normalerweise eine sinnvolle Annahme, da bei einem Befehl mit einem Quelloperanden und ohne Zieloperand nicht klar wäre, welche Wirkungen er haben soll.

- Andererseits wird ebenfalls davon ausgegangen, daß ein konstanter Operand stets nur als Quelloperand vorkommen kann; dies entspricht der üblichen Anschauung.

Da der PUSH-Befehl der einzige ist, bei dem die beschriebenen Punkte auftraten, wurde der Fehler lokal korrigiert und bei der Umwandlung von Maschinenbefehlen in Zwischencode explizit berücksichtigt.

Dieses Problem ist nicht erst durch die zusätzlichen Befehle des i80386 aufgetreten, da der PUSH-*immediate*-Befehl bereits seit dem i80186 existierte. Da dcc laut [11] DOS-Programme, die im Real Mode des i80286 lauffähig sind, unterstützt, stellt sich die Frage, warum dies nicht früher aufgefallen ist. Um dies zu klären, wurden einige Experimente durchgeführt:

Um die Funktionalität von dcc untersuchen zu können, wurden einige Testprogramme mit dem Borland-Turbo-C-Compiler übersetzt [11], wobei es sich laut [17] um Turbo C 2.01 handelt. Die meisten dieser Testprogramme lagen dem dcc-Paket als .exe-Dateien bei; eine Überprüfung mit dem bereits in Kapitel 4.3 beschriebenen Tool GetTyp 2000 bestätigte die Information über die Compiler-Version.

Bei manueller Überprüfung der vorliegenden Testprogramme anhand der durch ndcc erzeugten Assemblerlistings wurde festgestellt, daß keine der zusätzlichen Befehle des i80186 verwendet wurden. Da der Turbo-C-Compiler 2.01 [33] diese Befehle allerdings bereits unterstützte, wurden im folgenden die ebenfalls vorliegenden Quelltexte der Testprogramme erneut mit diesem Compiler übersetzt, um die Ursache für diese Diskrepanz zu bestimmen.

Eine Kompilierung mit den vom Compiler vorgegebenen Parametern erzeugte zwar keine mit den ursprünglichen Testprogrammen identischen Dateien, was wahrscheinlich auf Unterschiede in den Laufzeitbibliotheken oder den verwendeten Parametern zurückzuführen ist; der eigentliche Maschinencode der übersetzten Programmroutinen stimmte jedoch mit Ausnahme der Adressen von Funktionen und globalen Variablen überein.

Als nächster Schritt wurde die Kompilierung wiederholt, wobei der Compiler angewiesen wurde, für das erzeugte Programm die zusätzlichen i80186-Opcodes zu verwenden. Ein Vergleich der beiden Programmversionen – hierfür wurde Turbo C angewiesen, zu jedem Quellprogramm ein Assemblerprogramm zu erzeugen – ergab,

daß insgesamt die folgenden zusätzlichen Befehle genutzt wurden, unter denen sich die meistgenutzten zusätzlichen Befehle des i80186 befinden:

- `SHL` *`reg, imm`*: Schieben eines Registers um eine konstante Anzahl von Bits nach links.
- `PUSH` *`imm`*: Einen konstanten Wert auf den Stack legen.
- `IMUL` *`reg, r/m, imm`*: Multiplikation eines beliebigen Operanden mit einem konstanten Wert; Ergebnis wird in ein Register gespeichert.
- `ENTER` *`imm, imm`*: Funktions-Prolog; wird von Hochsprachen-Compilern am Anfang einer Funktion verwendet.
- `LEAVE`: Funktions-Epilog; wird von Hochsprachen-Compilern am Ende einer Funktion verwendet.

Beim Versuch, diejenigen Binärprogramme, die die zusätzlichen Opcodes enthielten, von `dcc` dekompilieren zu lassen, stellte sich heraus, daß die Dekompilierung in allen Fällen durchlief. Die zusätzlichen Befehle waren aber offenbar größtenteils nicht korrekt verarbeitet wurden, was punktuell zu der Erzeugung von eindeutig falschem C-Code führte. Lediglich `SHL` und `LEAVE` wurden korrekt umgesetzt, wobei der letztere Befehl ohnehin bei der Analyse ignoriert wird.

Zusammenfassend läßt sich daher sagen, daß die Unterstützung der i80186-Befehle bei der Entwicklung von `dcc` sicherlich beabsichtigt war. Allerdings wurde sie offenbar nur ungenügend bzw. gar nicht getestet, so daß sie eher als fragmentarisch zu bezeichnen ist. Im folgenden soll daher noch auf die bei den noch nicht diskutierten Befehlen auftretenden Probleme eingegangen werden.

Behandlung von `ENTER` *`imm, imm`*

Der `ENTER`-Befehl wird, ebenso wie der alternative Prolog `PUSH BP / MOV BP,SP / SUB SP,` *`imm`*, ausschließlich während der semantischen Analyse (s. Kap. 3.3.1) berücksichtigt und spielt in späteren Analysephasen keine Rolle mehr. (Der `ENTER`-Befehl entspricht nur dann der genannten Befehlsfolge, wenn sein zweiter Parameter den Wert 0 hat.)

Dieser Befehl wurde von `dcc` zwar ansatzweise richtig verarbeitet, jedoch führte die gleichzeitige Verwendung von Register-Variablen (d. h. Variablen, die während ihrer Lebensdauer in Prozessor-Registern gehalten werden) zu verschiedenen subtilen Fehlern. Die Ursache hierfür war, daß eine Funktion nach Auftreten des `ENTER`-Befehls als aus einer Hochsprache stammend gekennzeichnet wird und der Prolog somit als abgeschlossen betrachtet wird; die Erkennung der Verwendung von Register-

Variablen, die durch das Sichern der Register SI und DI durch PUSH-Befehle erfolgt, findet anschließend nicht mehr statt.

Eine Korrektur erfolgte durch eine entsprechende Behandlung des ENTER-Befehls bei der Erkennung von Idiomen. Für den komplementären LEAVE-Befehl, der der Folge MOV SP, BP / POP BP entspricht, war keine weitere Behandlung notwendig.

Behandlung von IMUL *reg, r/m, imm*

Diese spezielle Form eines vorzeichenbehafteten Multiplikationsbefehls nimmt im Befehlssatz des i80186 eine Sonderstellung ein, da er zwei *explizite* Quelloperanden und einen weiteren unabhängigen, ebenfalls expliziten Zieloperanden besitzt. (Eine ähnliche Möglichkeit gibt es zwar für die Addition mittels des LEA-Befehls; diese wird aber in der dcc-internen Repräsentation völlig anders dargestellt, so daß sich kein Problem ergibt.)

Im einzelnen waren dafür Änderungen an mehreren Stellen notwendig:

- Bei der Umwandlung in Zwischencode war der eben beschriebene Fall noch gar nicht berücksichtigt worden. Die Implementation wurde daher an der entsprechenden Stelle erweitert.
- Nach dieser Korrektur wurde immer noch falscher C-Code erzeugt, obwohl dieser Fall in der syntaktischen Analysephase grundsätzlich korrekt implementiert worden war. Allerdings führten verschiedene andere nicht berücksichtigte Effekte dazu, daß die von der Datenflußanalyse benötigten Definitions/Benutzt-Information für den vorliegenden Fall nicht richtig gesetzt wird.

5.1.4 Regressionstests

Bei den Änderungen an ndcc wurde darauf geachtet, die Analyse von Win32-Programmen so in den Dekompilierungsprozeß zu integrieren, daß die existierende Funktionalität für DOS-Programme erhalten wurde. Hierdurch können etwaige Verbesserungen am Kern von ndcc auch für DOS-Programme nutzbar gemacht werden.

Zudem erlaubte es dieses Vorgehen, während der Entwicklung bereits vor der vollständigen Implementation der Unterstützung von Win32-Programmen zu erkennen, ob durch die vollzogenen Änderungen Fehler in das Programm eingefügt worden waren. Dies geschah, indem während der Entwicklung laufend überprüft wurde, ob ndcc für die dem dcc-Paket beiliegenden und teilweise in [11] beschriebenen DOS-Programme dieselben oder bessere Resultate wie dcc lieferte.

5.1.5 Verarbeitung der Header-Dateien

Auch das in Kapitel 3.2.3 beschriebene Tool `parsehdr` mußte modifiziert werden, da es in der vorliegenden Form nicht an die Win32-Header-Dateien und die an `ndcc` vorgenommenen Änderungen angepaßt war.

Im einzelnen betraf dies folgende Punkte:

- Die verschiedenen ganzzahligen Datentypen werden von einem 32-Bit C-Compiler anders interpretiert als von einem 16-Bit C-Compiler. Zwar entspricht ein `long int` in beiden Fällen einer 32-Bit-Zahl und ein `short int` einer 16-Bit-Zahl, jedoch wird ein einfacher `int` der nativen Wortlänge angepaßt.

  Bei einer Dekompilierung eines Binärprogramms läßt sich nicht mehr feststellen, ob eine Variable explizit als `short` bzw. `long` deklariert wurde oder ob der Standard-Typ `int` verwendet wurde; daher wird in `ndcc` kein generischer `int`-Typ verwendet. Da es hierdurch notwendig wird festzulegen, welchem internen Typ `int` ohne qualifizierende Schlüsselwörter zuzuordnen ist, muß `parsehdr` beim Aufruf durch den Parameter `-w` mitgeteilt werden, daß es sich bei den zu parsenden Header-Dateien um Win32-Header-Dateien handelt.

  Zusätzlich führt dieser Parameter dazu, daß die erzeugte Prototypen-Datei nicht `dcclibs.dat`, sondern `dcclibsw32.dat` genannt wird.

- Da Bezeichner von Win32-API-Funktionen recht lang werden können, wurde die zulässige Länge von ursprünglich 15 Zeichen auf vorerst 39 Zeichen geändert. Gleichzeitig wurde das Dateiformat der Prototypen-Datei dahingehend angepaßt, daß dieser Wert dort gespeichert und flexibel geändert werden kann, ohne `ndcc` erneut anpassen zu müssen.

- In den Win32-Header-Dateien kommen sehr viele Bezeichner vor, die für `parsehdr` irrelevant sind (`WINBASEAPI`, `CALLBACK`, ...) und daher ignoriert werden sollten. Die Liste dieser Bezeichner wurde durch manuelle Inspektion einiger Win32-Header-Dateien ermittelt; viele dieser Bezeichner werden normalerweise durch den C-Präprozessor entfernt, so daß ein weiterentwickelter Parser ebenfalls dazu in der Lage wäre. (Eine vorherige Filterung durch einen C-Präprozessor wurde untersucht, hat sich aber als nicht praktikabel herausgestellt.)

- Ebenfalls werden für fast alle Basistypen alternative Bezeichner definiert, z. B. `ULONG` für `unsigned long int`. Diese wurden zu den Begriffen der bereits in `parsehdr` vorhandenen Heuristik hinzugefügt, was zufriedenstellende Resultate lieferte.

5.2 Resultate

5.2.1 Dekompilierung eines Testprogrammes

Um die grundsätzliche Funktionalität der vollzogenen Änderungen zu untersuchen, wurde zunächst ein Win32-`hello-world`-Programm dekompiliert. Die Erzeugung der verschiedenen Versionen (Listings 5.4 und 5.5, Abb. 5.1 auf Seite 53) dieses Programms wurde in [39] ausführlich dargestellt; Listing 5.4 zeigt die ursprüngliche C-Version dieses Programms.

Listing 5.4: hello.c: Ein kleines Programm in C

```
#include <stdio.h>
void main(void)
{
    puts("hello, world");
}
```

Da die Verwendung von Compiler- und Funktions-Signaturen für Win32-Programme noch nicht in `ndcc` implementiert wurde, die im Programm vorkommende Funktion `puts()` aber auf die Laufzeit-Bibliothek eines C-Compilers angewiesen ist, mußte das Programm manuell umgeschrieben werden, damit auf die Einbindung von Bibliotheken verzichtet werden konnte. Anstatt Bibliotheksfunktionen zu verwenden, greift die in Listing 5.5 dargestellte Version des Programms daher direkt auf Win32-API-Funktionen zurück.

Diese angepaßte Version des Programms wurde im folgenden manuell in Assemblercode übersetzt und eine Win32-PE-`.exe`-Datei vollständig manuell konstruiert. Eine Erzeugung der PE-Datei durch einen Compiler wäre prinzipiell ebenfalls möglich gewesen, die resultierende Datei wäre allerdings wesentlich größer gewesen, was die Darstellung an dieser Stelle erschwert und keine Vorteile für die Demonstration der Dekompilierung gebracht hätte; die Funktionalität wäre ebenfalls identisch gewesen. Ein Hex-Dump der resultierenden `.exe`-Datei findet sich in Abb. 5.1; für die vollständige Dokumentation des Erzeugungsprozesses sei auf [39] verwiesen. (Das Programm ist aufgrund von Alignment-Anforderungen in der dargestellten Binärform nur auf der Windows NT-Plattform, nicht aber unter Windows 9x lauffähig. Dies ist aber für die Dekompilierung unerheblich, und ein zu Windows 9x kompatibles Alignment hätte lediglich die `.exe`-Datei unnötig vergrößert und die Darstellung an dieser Stelle erschwert.)

Zuerst soll die in `ndcc` integrierte Disassemblierungsfunktion demonstriert werden. Hierzu wird `ndcc` mit den Parametern `-a hello.exe` aufgerufen, wodurch die Datei

```
0000  4d 5a 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |MZ..............|
0010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0030  00 00 00 00 00 00 00 00  00 00 00 00 40 00 00 00  |............@...|
0040  50 45 00 00 4c 01 02 00  00 00 00 00 00 00 00 00  |PE..L...........|
0050  00 00 00 00 e0 00 02 01  0b 01 00 00 20 00 00 00  |............ ...|
0060  a0 00 00 00 00 00 00 00  a0 01 00 00 a0 01 00 00  |................|
0070  c0 01 00 00 00 00 10 00  20 00 00 00 20 00 00 00  |........ ... ...|
0080  04 00 00 00 00 00 00 00  04 00 00 00 00 00 00 00  |................|
0090  c0 00 00 00 a0 01 00 00  00 00 00 00 03 00 00 00  |................|
00a0  00 00 10 00 00 10 00 00  00 00 10 00 00 10 00 00  |................|
00b0  00 00 00 00 10 00 00 00  00 00 00 00 00 00 00 00  |................|
00c0  e0 01 00 00 6f 00 00 00  00 00 00 00 00 00 00 00  |....o...........|
00d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0120  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
0130  00 00 00 00 00 00 00 00  2e 63 6f 64 65 00 00 00  |.........code...|
0140  00 00 00 00 a0 01 00 00  20 00 00 00 a0 01 00 00  |........ .......|
0150  00 00 00 00 00 00 00 00  00 00 00 00 20 00 00 60  |............ ..`|
0160  2e 64 61 74 61 00 00 00  00 00 00 00 c0 01 00 00  |.data...........|
0170  a0 00 00 00 c0 01 00 00  00 00 00 00 00 00 00 00  |................|
0180  00 00 00 00 40 00 00 c0  00 00 00 00 00 00 00 00  |....@...........|
0190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
01a0  6a 00 68 d0 01 10 00 6a  0d 68 c0 01 10 00 6a f5  |j.h....j.h....j.|
01b0  2e ff 15 28 02 10 00 50  2e ff 15 24 02 10 00 c3  |...(...P...$....|
01c0  68 65 6c 6c 6f 2c 20 77  6f 72 6c 64 0a 00 00 00  |hello, world....|
01d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
01e0  18 02 00 00 00 00 00 00  ff ff ff ff 08 02 00 00  |................|
01f0  24 02 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |$...............|
0200  00 00 00 00 00 00 00 00  6b 65 72 6e 65 6c 33 32  |........kernel32|
0210  2e 64 6c 6c 00 00 00 00  30 02 00 00 40 02 00 00  |.dll....0...@...|
0220  00 00 00 00 30 02 00 00  40 02 00 00 00 00 00 00  |....0...@.......|
0230  01 00 57 72 69 74 65 43  6f 6e 73 6f 6c 65 41 00  |..WriteConsoleA.|
0240  02 00 47 65 74 53 74 64  48 61 6e 64 6c 65 00 00  |..GetStdHandle..|
0250  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Abbildung 5.1: hello.exe: Ein kleines Programm, Hex-Dump

Listing 5.5: hello.c: Ein kleines Programm in C, angepaßte Version

```
#define STD_OUTPUT_HANDLE -11UL
#define hello "hello, world\n"

__declspec(dllimport) void * __stdcall
GetStdHandle(unsigned long nStdHandle);

__declspec(dllimport) int __stdcall
WriteConsoleA(void *hConsoleOutput,
              const void *lpBuffer,
              unsigned long nNumberOfCharsToWrite,
              unsigned long *lpNumberOfCharsWritten,
              void *lpReserved
              );

unsigned long written;

void main(void)
{
    WriteConsoleA(GetStdHandle(STD_OUTPUT_HANDLE),
                  hello,sizeof(hello)-1,&written,0);
    return;
}
```

`hello.a1` in Abb. 5.2 erzeugt wird. Das Assemblerlisting ist folgendermaßen zu lesen:

- Die Zeilen `main PROC NEAR` und `main ENDP` rahmen die Funktion mit dem Namen `main` ein.
- In der ersten Spalte werden Indizes für die Maschinenbefehle der Prozedur geführt. Diese entsprechen nicht unbedingt der Reihenfolge, in der die Maschinenbefehle im Programm vorliegen; vielmehr fährt `ndcc` nach einem `JMP`-Befehl mit der Ausgabe des adressierten Folgeblocks fort, sofern dieser nicht bereits ausgegeben worden war. Fehlende Blöcke werden am Schluß der Prozedur ausgegeben, und es wird durch einen synthetischen `JMP`-Befehl (durch den Kommentar `Synthetic inst` gekennzeichnet) die Semantik erhalten.

  Dieses Verhalten ist eine Folge davon, daß `ndcc` primär kein Disassembler, sondern vielmehr ein Decompiler ist und für die Dekompilierung lediglich die

```
                main  PROC  NEAR
000 001001A0 6A00                PUSH           0
001 001001A2 68D0011000          PUSH           1001D0h
002 001001A7 6A0D                PUSH           0Dh
003 001001A9 68C0011000          PUSH           1001C0h
004 001001AE 6AF5                PUSH           0FFFFFFF5h
005 001001B0 2EFF1528021000      CALL  near ptr GetStdHandle
006 001001B7 50                  PUSH           eax
007 001001B8 2EFF1524021000      CALL  near ptr WriteConsoleA
008 001001BF C3                  RET

                 main  ENDP
```

Abbildung 5.2: hello.a1: Ein kleines Programm, Assemblerlisting

Ausführungsreihenfolge, nicht aber die tatsächliche Plazierung der Befehle relevant ist. Abgesehen davon werden in einer späteren Phase alle `JMP`-Befehle nur noch implizit als Kanten eines Graphen dargestellt, so daß die Offsets nicht mehr relevant sind.

- In der zweiten Spalte findet sich in hexadekadischer Notation der Offset des jeweiligen Maschinenbefehls. Dieser entspricht normalerweise nicht dem tatsächlichen Offset in der Datei, sondern ergibt sich durch den dort angegebenen Wert *Image Base* und durch Alignment-Bedingungen.
- In der dritten Spalte sind – ebenfalls in hexadekadischer Notation – die tatsächlichen Objektcode-Bytes des Maschinenbefehls aufgeführt.
- Der Rest der Zeile besteht aus einem optionalen Label (`L`*`XX`*`:`), dem Assembleräquivalent des Maschinenbefehls und einem durch ein Semikolon abgetrennten optionalen Kommentar.

Während des Disassemblierungsvorgangs werden von `ndcc` einige Meldungen ausgegeben, die hier kurz erläutert werden sollen:

```
Main could not be located!
Model: x
Warning - compiler not recognised
Signature file: .../dcc/dccxxx.sig
```

Zwischen diesen Meldungen besteht ein Zusammenhang: Da kein Startup-Code eines Compilers vorhanden ist, kann `ndcc` keinen Compiler anhand seiner Signatur erkennen, wovon auch die Erkennung der Position der `main()`-Funktion abhängt. Daher beginnt die Dekompilierung an der Adresse `001001A0 hex`, was in diesem Fall auch korrekt ist, da kein Startup-Code vorhanden ist, der ignoriert werden müßte. Die an dieser Stelle beginnende Funktion wird trotzdem mit `main` bezeichnet, damit klar wird, daß der Programmfluß dort beginnt.

```
Reading library prototype data file .../dcclibsw32.dat
```

Diese Meldung dient lediglich der Information und besagt, daß korrekterweise auf die Datei zugegriffen wird, die die Prototypen der Windows-API-Funktionen enthält.

```
Warning: cannot open signature file .../dcc/dccxxx.sig
```

Auch diese Meldung ist eine Folge davon, daß kein Compiler erkannt wurde. Es kann dadurch nicht ermittelt werden, welche Datei die passenden Signaturen für die Bibliotheksfunktionen enthält, so daß `ndcc` auf die (nicht vorhandene) Datei `dccxxx.sig` zurückfällt. Im vorliegenden Fall stellt dies keinen Fehler dar, da das untersuchte Programm keine Bibliotheksfunktionen enthält. Zudem sind noch keine Bibliotheks- oder Compilersignaturen für Win32-Programme erzeugt worden.

```
dcc: writing assembler file hello.a1
dcc: Writing C beta file hello.b
dcc: Finished writing C beta file
```

Diese Informationsmeldungen besagen, daß die disassemblierte und die dekompilierte Darstellungsform des Programms erzeugt wurden.

```
Call Graph:
main
   GetStdHandle
   WriteConsoleA
```

An der ausgegebenen Struktur des Aufruf-Graphen läßt sich erkennen, daß von der Funktion `main` aus die beiden Funktionen `GetStdHandle` und `WriteConsoleA` aufgerufen werden. Da keine Funktionssignaturen verwendet wurden, stammen die Namen der Funktionen aus der Importtabelle des Win32-PE-Dateiformates.

Das erzeugte C-Programm ist in Listing 5.6 abgebildet. Hierbei sind in erster Linie die an `WriteConsoleA` übergebenen Parameter von Interesse:

hConsoleOutput: Der Rückgabewert der Funktion `GetStdHandle()` ist korrekt eingesetzt worden.

lpBuffer: Da es sich bei diesem Parameter laut Prototyp um einen untypisierten Zeiger (`void *`) handelt, kann `ndcc` vorerst keine weiteren Schlüsse ziehen und setzt den Zahlenwert ein. Ändert man den Prototypen in der Header-Datei zu Testzwecken, so daß `lpBuffer` den passenden Typen `char *`, also Zeiger auf eine Zeichenkette erhält, so fügt `ndcc` korrekt die Zeile

```
WriteConsoleA (GetStdHandle (-11),"hello, world\n", 13, 0x1001D0, 0);
```

ein. Ein entsprechendes Resultat ließe sich prinzipiell auch durch eine Heuristik im Decompiler erreichen, die allerdings in anderen Fällen auch zu Fehlinterpretationen führen könnte.

nNumberOfCharsToWrite: Die Zahl ist mit 13 korrekt wiedergegeben; daß es sich um die Länge der Zeichenkette `"hello, world\n"` handelt, kann `ndcc` nicht ermitteln, ohne die Semantik der Funktion `WriteConsoleA` zu kennen.

lpNumberOfCharsWritten: Hier bestehen noch Verbesserungsmöglichkeiten, da `ndcc` lediglich den Zahlenwert des Zeigers einsetzt. Aus dem Prototypen geht zwar hervor, daß es sich um einen Zeiger auf eine 32-Bit-Zahl (`unsigned long *`) handelt; intern wird jedoch lediglich zwischen Zeigern auf Zeichenketten und anderen Zeigern unterschieden.

lpReserved: Hier wird wieder der Zahlenwert des Zeigers wiedergegeben, was im Falle der Null auch korrekt ist.

Listing 5.6: hello.b: Ein kleines Programm, dekompilierter C-Code

```
/*
 * Input file  : hello.exe
 * File type   : Win32 PE-EXE
 */

#include "dcc.h"


void main ()
/* Takes no parameters.
 */
{
    WriteConsoleA (GetStdHandle (-11), 0x1001C0, 13, 0x1001D0, 0);
}
```

5.2.2 Dekompilierung eines Win32-Trojaners

Um die Anwendbarkeit von ndcc auf reale Malware zu testen, wurde seine Funktion an denjenigen Samples untersucht, die, wie in Kapitel 4.6 beschrieben, prinzipiell für eine Dekompilierung geeignet erschienen. Dabei stellte sich heraus, daß die aktuelle Version von ndcc bei keinem der Samples in der Lage war, eine erfolgreiche Dekompilierung durchzuführen. Es gelang allerdings bei einer Reihe von Samples, ein Assemblerlisting zu erzeugen.

Im folgenden sollen die bei der Analyse des in der Datei DUNPWS\TESTSP-C.EE vorliegenden Trojaners trojan://W32/PSW.TestSpy.c erzielten Ergebnisse diskutiert werden. Für die Wahl dieses Samples sprachen folgende Gründe:

- Eine Disassemblierung mit einem anderem Disassembler war möglich, so daß das Ergebnis auf seine Korrektheit hin überprüft werden konnte.
- Die Datei lag bereits unverschleiert vor und mußte somit nicht erst entschleiert werden, was die Struktur des Programms u. U. hätte beschädigen können.
- Das Sample ist zwar mit 47.634 Bytes relativ groß, davon beträgt der eigentliche Programmcode allerdings lediglich 1.101 Bytes; der Rest besteht aus vom Programm verwendeten Daten, diversen Strukturinformationen, Windows-Ressourcen und Alignment-bedingtem Verschnitt. Durch den geringen Codeumfang ist das Programm somit für eine manuelle Analyse relativ übersichtlich, und es besteht eine geringere Wahrscheinlichkeit, daß für ndcc problematische Befehle vorkommen.
- Soweit es bei einer manuellen Inspektion des Assemblerlistings erkennbar war, ist der Code offenbar mit einem Compiler erzeugt worden. (Durch Assembler-Programmierung kann nicht nur, wie in Kapitel 4.1 beschrieben, eine Dekompilierung verhindert, sondern auch eine Disassemblierung erschwert werden.)
- Obwohl dieses Sample vermutlich mit einem Compiler erzeugt worden war, enthielt es weder Bibliotheksroutinen noch Startup-Code. In Anbetracht der Tatsache, daß ndcc in der aktuellen Version noch keine Bibliothekssignaturen für Win32-Programme verwenden kann, ist dies durchaus vorteilhaft.

Ergebnis der Disassemblierung

Aufgrund der potentiell schädlichen Natur des Programmcodes werden an dieser Stelle nur Auszüge des erzeugten Assemblerlistings abgedruckt. Allerdings lassen sich bereits anhand des in Abbildung 5.3 gezeigten Ausschnitts vom Anfang des Programms wesentliche Aussagen über das Disassemblierungsergebnis nachvollziehen:

```
        main  PROC  NEAR
000 00401000 55                          PUSH          ebp
001 00401001 8BEC                        MOV           ebp, esp
002 00401003 81EC58010000                SUB           esp, 158h
003 00401009 56                          PUSH          esi
004 0040100A 8D85A8FEFFFF                LEA           eax, [ebp-158h]
005 00401010 57                          PUSH          edi
006 00401011 33F6                        XOR           esi, esi
007 00401013 33FF                        XOR           edi, edi
008 00401015 50                          PUSH          eax
009 00401016 6804010000                  PUSH          104h
010 0040101B FF15EC304000                CALL  near ptr GetTempPathA
```

Abbildung 5.3: Beginn des Assemblerlisting von `trojan://W32/PSW.TestSpy.c`

- Die Disassemblierung von 32-Bit-x86-Maschinencode erfolgt (für die in diesem Programm vorhandenen Maschinenbefehle) korrekt, 32-Bit-Adreß- und -Datenoperanden werden richtig eingelesen.
- An Befehl `010` läßt sich wie in Kapitel 5.2.1 erkennen, daß die Namen von importierten Funktionen aus den entsprechenden Strukturen der PE-Datei übernommen wurden. Importierte Funktionen können auch auf eine andere Art aufgerufen werden, die z. B. in der Funktion `proc_1` in der Zeile

  ```
  101 00401347 E80E010000              CALL     near ptr LZOpenFileA
  ```

  verwendet wurde, wo eine vom Linker generierte Stub-Funktion (s. S. 44) aufgerufen wird.

  Korrekterweise müßte ein Funktionsaufruf nach der erstgenannten Methode als

  ’`010 0040101B FF15EC304000          CALL dword ptr [GetTempPathA]`’,

  also als indirekter Funktionsaufruf, disassembliert werden. Da sich dies allerdings – abgesehen von der höheren Effizienz – nicht von der zweiten Methode unterscheidet, werden beide Möglichkeiten als direkter Funktionsaufruf der entsprechenden importierten Funktion erkannt und identisch disassembliert. Dies ist u. a. auch deshalb nötig, weil der entsprechende Funktionsprototyp anhand des Funktionsbezeichners identifiziert wird.

Probleme bei der Dekompilierung

Nach der vollständigen Disassemblierung des Programm durch ndcc stellte sich die Frage, warum die Dekompilierung nicht ebenfalls erfolgreich war, sondern mit der Fehlermeldung

```
dcc: Definition not found for condition code usage at opcode 68
```

abbrach. Um eine aussagekräftigere Fehlerdiagnose zu bekommen, wurde ndcc modifiziert, was die Fehlermeldung

```
ndcc: Definition not found for condition code usage at opcode 68, proc
proc_2, addr 0x40113B
```

lieferte, mit Hilfe derer die genaue Position des Abbruchs bestimmt werden konnte.

Dies entspricht der in Abbildung 5.4 gezeigten Zeile 009 in der Funktion proc_2, was zu der Vermutung führte, daß ndcc nicht korrekt mit String-Operationen umgehen kann. Durch eine Inspektion des Quellcodes von ndcc wurde festgestellt, daß String-Operationen zwar zu Beginn der Analyse berücksichtigt, aber nicht in Zwischencode überführt werden, was zu einem späteren Zeitpunkt zu einem Programmabbruch führt. Ein Vergleich mit dem ursprünglichen Quellcode von dcc bestätigte, daß dieser Fehler bereits dort vorhanden war.

```
009 0040113B A4                         MOVSB
010 0040113C 8DBDF5FEFFFF               LEA             edi, [ebp-10Bh]
011 00401142 B940000000                 MOV             ecx, 40h
012 00401147 F3AB                       REP STOSD
013 00401149 66AB                       STOSW
014 0040114B AA                         STOSB
```

Abbildung 5.4: Ursache des Abbruchs der Dekompilierung

Im Falle eines einzelnen String-Befehls ließe sich die Umsetzung in Zwischencode wahrscheinlich mit relativ geringem Aufwand implementieren; beispielsweise könnte der MOVSB-Befehl im vorliegenden Fall als HLI_ASSIGN (char *)*edi++, (char *)*esi++ dargestellt werden. Hierbei muß allerdings zusätzlich der Zustand des Direction-Flags (*DF*) der CPU berücksichtigt werden, da dieses angibt, ob die Register inkrementiert (DF=0) oder dekrementiert (DF=1) werden sollen, so daß die Darstellung ggf. auch HLI_ASSIGN (char *)*edi--, (char *)*esi-- lauten könnte.

Das Direction Flag wurde grundsätzlich bereits in dcc berücksichtigt. Problema-

tisch ist allerdings, daß dieses Flag von Hochsprachen-Compilern meist nur einmalig initialisiert und dann nicht mehr verändert bzw. immer auf denselben Wert (meist 0) zurückgesetzt wird, da die Analyse von Prozessor-Flags in `dcc` lediglich innerhalb eines Basisblocks durchgeführt wird. Daher wäre es sinnvoll, in Fällen, in denen innerhalb eines Basisblocks kein Befehl gefunden wird, der das Direction Flag setzt (dies sind `CLD` und `STD`), von einem vorgegebenen Wert auszugehen.

Schwieriger ist dies bei den mit einem `REP`-Präfix versehenen String-Befehlen, da sich diese in der Architektur von `dcc` offenbar nicht ohne weiteres auf den Zwischencode abbilden lassen. Eine Behandlung während der semantischen Analyse, indem der zusammengesetzte String- durch eine entsprechende Schleife ersetzt wird, scheidet ebenfalls aus, da sich dort keine zusätzlichen Befehle einfügen lassen.

Eine vollständige Behandlung der String-Befehle wurde daher vorerst zurückgestellt. Vermutlich waren diese bei der Implementation von `dcc` als für die Analyse unerheblich eingestuft worden, da DOS-Compiler meist nur wenig optimierten Code erzeugen konnten; String-Befehle kamen daher nur in Bibliotheksroutinen vor, welche durch die Verwendung von Bibliothekssignaturen ohnehin nicht weiter analysiert wurden.

5.3 Weitere durchzuführende Änderungen

Im Rahmen einer Diplomarbeit war es erwartungsgemäß nicht möglich, `ndcc` zu einem voll funktionsfähigen Win32-Decompiler weiterzuentwickeln. Dies gilt insbesondere, da die Grundlage `dcc` lediglich als prototypischer Decompiler entwickelt worden war und selbst unter Berücksichtigung dieser Voraussetzung nicht alle Spezifikationen erfüllte, wie z. B. an den in Kapitel 5.1.3 behandelten zusätzlichen Befehlen des i80186 und den in Kapitel 5.2.2 erörterten String-Operationen ersichtlich ist. Im folgenden soll daher ein zusammenfassender Überblick über mögliche Erweiterungen erfolgen.

5.3.1 Unterstützung weiterer Maschinenbefehle

Die Unterstützung der bereits diskutierten String-Operationen ist auf jeden Fall wünschenswert, da diese von modernen 32-Bit-Compilern als Resultat einer Schleifenoptimierung oder durch das Inlining von Bibliotheksfunktionen durchaus verwendet werden.

Ebenfalls wäre eine vollständige Unterstützung der restlichen zusätzlichen i80386-Befehle sinnvoll. Außerdem müßte die korrekte Verarbeitung der bereits unterstützten Befehle anhand von weiteren Testprogrammen überprüft werden.

5.3.2 Vollständige Unterstützung der 32-Bit-Adressierungsmodi

Wie bereits in Kapitel 5.1.1 erläutert wurde, werden die erweiterten Adressierungsmodi des i80386 zur Zeit noch nicht von `ndcc` unterstützt. Für nicht-triviale Programme ist dies allerdings je nach den bei der Programmierung des konkret vorliegenden Programms verwendeten Sprachelementen erforderlich: Wenn im ursprünglichen Quellprogramm keine Zeiger oder Array-Zugriffe benutzt wurden, wird das resultierende Binärprogramm oft ohne die erweiterten Adressierungsmodi auskommen, da die für den Zugriff auf den Stack mittels des Registers `EBP` benötigten Adressierungsmodi bereits genügen.

Es ist aber zu berücksichtigen, daß moderne Compiler die erweiterten Adressierungsmodi ebenfalls nutzen bzw. zweckentfremden, um mittels des `LEA`-Befehls Rechnungen durchzuführen, da hiermit zwei Register und eine Konstante addiert und das Ergebnis in ein drittes Register geschrieben werden können. Diese Verwendungsweise wurde noch nicht mit `ndcc` getestet; es ist also möglich, daß dabei Schwierigkeiten auftreten, weil die so berechneten Werte immer als Adressen interpretiert werden.

5.3.3 Startup-Signaturen für Win32-Compiler

Für die Dekompilierung von realistischen Win32-Programmen ist es auf jeden Fall wichtig, die Startup-Signaturen der gängigsten Win32-Compiler zu erkennen, um die Analyse am Hauptprogramm beginnen und die richtigen Bibliothekssignaturen auswählen zu können.

Zur Zeit erfolgt die Erkennung des Compilers und die Bestimmung der Position des Hauptprogramms in `ndcc` durch manuell implementierte Signaturen. Es ist daher wünschenswert, diese Signaturen in eine externe Datei auszulagern, um `ndcc` leichter erweitern zu können.

Eine automatische Generierung dieser Signaturen ist hingegen weniger entscheidend, da nur eine Signatur je Compiler und Speichermodell (wobei Win32 nur ein Speichermodell besitzt) benötigt wird. Außerdem wäre dies vermutlich unverhältnismäßig schwierig, da der vom Compiler verwendete Startup-Code nicht wie die Bibliotheksroutinen in einem standardisierten Dateiformat vorliegt bzw. überhaupt nicht in einer vom Compiler getrennten Datei verfügbar ist; dies wird von in [28] beschriebenen Erfahrungen bestätigt.

5.3.4 Verbesserter Algorithmus für Bibliothekssignaturen

Wie teilweise bereits in Kapitel 3.2.1 erläutert wurde, besitzt das von `dcc` verwendete Verfahren zur Erzeugung von Bibliothekssignaturen verschiedene Nachteile, die unter anderem auch die Erweiterung auf 32-Bit-Befehle erschweren:

1. Das Verfahren ist von der binären Repräsentation der Maschinenbefehle abhängig, so daß es bei einer Erweiterung des von `dcc` unterstützten Befehlssatzes nicht ausreicht, nur die syntaktische Analyse anzupassen, sondern dies ebenfalls bei der Erzeugung der Signaturen berücksichtigt werden muß.

   Aus diesem Grund wird die Implementation des Verfahrens durch die Berücksichtigung sehr vieler Eigenschaften des Maschinencodes relativ aufwendig; in `dcc` werden hierfür ca. 500 Zeilen Quellcode benötigt, was eine evtl. fehlerträchtige Replikation von Funktionalität darstellt. Dies gilt insbesondere, da das Verfahren sowohl von den Tools zur Erzeugung der Signatur-Dateien als auch von `dcc` selbst während der Erkennung der Bibliotheksroutinen eingesetzt wird, wobei allerdings in beiden Fällen derselbe Quellcode eingesetzt wird.

   Tatsächlich behandelt das Verfahren zwar bereits die zusätzlichen i80386-Opcodes, jedoch werden weder 32 Bit breite Adressen und Operanden, die Umschaltung der Wortbreite durch Präfix-Bytes (s. Kap. 5.1.1) noch die geänderte Interpretation des Objektcodes bei Verwendung der 32-Bit-Adressierungsmodi berücksichtigt.

2. Es werden mehr Bytes als variant markiert als notwendig, da ausschließlich von den im Maschinencode selbst vorhandenen Informationen ausgegangen wird. Konkret werden fast alle 16-Bit-Immediate-Operanden durch Wildcards ersetzt, obwohl diese nicht alle variant sein müssen.

3. Unter Umständen werden Signaturen früher abgeschnitten als notwendig, da das Verfahren jeden unbedingten Kontrollflußtransfer als das Ende einer Funktion betrachtet.

4. Zudem werden Signaturen immer spätestens nach einer bestimmten Anzahl von Bytes abgeschnitten, was wie die beiden vorherigen Punkte dazu führen kann, daß für eigentlich unterschiedliche Funktionen trotzdem identische Signaturen erzeugt werden.

Aufgrund dieser Beobachtungen bietet sich eine andere Vorgehensweise für die Erzeugung der Bibliothekssignaturen an, indem Informationen darüber, welche Bytes variant sind und wann eine Funktion beendet ist, ausschließlich den Strukturinformationen der Bibliotheksdatei entnommen werden. Dies behebt zunächst die meisten Auswirkungen der drei erstgenannten Nachteile:

1. Es besteht keine Abhängigkeit von der Interpretation des Objektcodes mehr, alle nötigen Informationen sind in den Verwaltungsstrukturen der Bibliotheksdatei enthalten.

2. Es werden genau diejenigen Bytes als variant markiert, die tatsächlich beim Binden der Bibliotheksroutinen verändert werden können.

3. Die Länge jeder Bibliotheksfunktion ist in der Bibliotheksdatei abgelegt, so daß ein unbedingter Kontrollflußtransfer nur dann zu einem Abschneiden der Signatur führt, wenn die Funktion an der betreffenden Stelle tatsächlich zu Ende ist. Es können mehrere Funktionen in einem zusammenhängenden Block in der Bibliotheksdatei abgelegt sein (dies kann lediglich dazu führen, daß beim Binden mehr Code als nötig eingebunden wird). Da diese Funktionen in diesem Fall aber auch im Binärprogramm zusammenhängend abgelegt sind, schadet es nicht, wenn sich dadurch eine Signatur über das eigentliche Ende der Funktion hinaus erstreckt.

Da die so erzeugbaren Signaturen *echte* Wildcards enthalten, können sie weder mittels eines Hash-Verfahrens in der Signatur-Datei gesucht noch direkt Byte für Byte mit dem Code im Binärprogramm verglichen werden. Daher bietet es sich beispielsweise an, in der Signatur-Datei einen Entscheidungsbaum abzulegen. Dies spart gleichzeitig Speicherplatz, da diejenigen Bytes, die mehreren Funktionen gemeinsam sind, nur einmal abgelegt werden müssen [28]. Ein solches Verfahren dürfte kaum langsamer als das bisher verwendete Hash-Verfahren sein, da die Anzahl der notwendigen Vergleiche logarithmisch zur Anzahl der Funktionen wächst; zudem entfällt die Notwendigkeit, das Verfahren zur Signaturerzeugung ebenfalls auf jede Funktion im Binärprogramm anzuwenden.

Es besteht weiterhin das Problem, daß verschiedene Funktionen zwar in den ersten n Bytes identisch sein können, sich aber später unterscheiden. In [28] wurde hierfür folgender Ansatz gewählt:

1. Grundsätzlich werden für die Erstellung der Signatur die ersten 32 Bytes verwendet.

2. Unterscheiden sich verschiedene Funktionen in den ersten 32 Bytes nicht in den konstanten (d. h. nicht-varianten) Bytes, so werden diese zunächst in demselben Endknoten des Baumes gespeichert. Da es relativ sicher ist, daß eine Bibliotheksfunktion erkannt wurde, wenn eine Funktion im Binärprogramm innerhalb der ersten 32 Bytes mit einer Bibliotheksfunktion übereinstimmt, ist es nun gerechtfertigt, für den restlichen Vergleich aufwendigere Verfahren einzusetzen, die zu einer möglichst eindeutigen Identifizierung führen.

3. Falls die Funktionen in den ersten 32 Bytes übereinstimmen, wird zusätzlich die CRC16-Prüfsumme vom 33. Byte bis zum nächsten varianten Byte der Funktion berechnet und ebenfalls in der Signatur-Datei abgelegt; es wird ebenfalls

die Anzahl der hierfür berücksichtigten Bytes abgespeichert, da sich diese unterscheiden kann. Der Fall, daß das 33. Byte variant ist, wird akzeptiert, da dieser Fall anscheinend in der Praxis selten auftritt.

4. Falls die Funktionen bis hierher nicht unterschieden werden konnten, wird versucht, eine Position zu finden, an der sich alle Funktionen in einem Endknoten unterscheiden.

5. Wenn sich Funktionen tatsächlich nicht in den konstanten Bytes unterscheiden, wird versucht, die von dieser Funktion referenzierten Namen – diese Information ist in der Bibliotheksdatei vorhanden – zu berücksichtigen. Dies hat allerdings den Nachteil, daß die Erkennung einer Bibliotheksfunktion von der vorherigen Erkennung einer anderen Bibliotheksfunktion abhängig wird, was entsprechend in der Architektur des Disassemblers oder Decompilers berücksichtigt werden muß. Ggf. müssen bei der Erkennung mehrere Durchgänge durchgeführt werden; andererseits gibt es im genannten Fall keine Alternative zu diesem Verfahren.

   In `ndcc` ist eine Durchführung von mehreren Durchgängen zur Zeit nicht ohne weiteres möglich, da die Erkennung nur einmalig aufgerufen wird, sobald die Analyse erstmalig auf die Funktion stößt. Es ist jedoch eine rekursive Lösung denkbar, indem bei der Untersuchung einer Bibliotheksfunktion zunächst weitere, von dieser referenzierte Bibliotheksfunktionen analysiert werden.

Die Praxistauglichkeit des beschriebenen Verfahrens ist durch die erfolgreiche kommerzielle Implementation in [14] erwiesen. Dennoch sind verschiedene Verbesserungen denkbar:

- Die CRC16-Prüfsumme (oder ein ähnliches Verfahren) wird über alle restlichen nicht-varianten Bytes der Funktion gebildet; ggf. kann die berücksichtigte Länge auch auf eine bestimmte Obergrenze beschränkt werden. Dabei genügt offensichtlich nicht mehr die Kenntnis der Anzahl der Bytes; vielmehr muß – beispielsweise mittels einer Bitmaske – für jedes einzelne Byte abgespeichert werden, ob dieses bei der Berechnung der Prüfsumme berücksichtigt wurde.

- Da das Verfahren durch die Verwendung von Suchbäumen anstelle von Hash-Funktionen nicht mehr von vornherein auf Signaturen fester Länge beschränkt ist, können auch solange zusätzliche Bytes für die einzelnen Signaturen herangezogen werden, bis ein Unterschied festgestellt wird; kürzere Signaturen bei frühzeitig erzielter Eindeutigkeit sind hingegen vermutlich weniger sinnvoll.

  Durch diese Maßnahme wird gewährleistet, daß für Funktionen, die nicht in allen konstanten Bytes übereinstimmen, auf Anhieb unterschiedliche Signaturen gefunden werden können. Dies vereinfacht das Verfahren, da die Punkte

Listing 5.7: Darstellung eines Operanden eines Maschinenbefehls

```
// LOW_LEVEL icode operand record
typedef struct ICODEMEM {
    Int     off;          // memory address offset
    int     segValue;     // Value of segment seg during analysis
    bool    segdef;       // Segment defined
    byte    seg;          // CS, DS, ES, SS
    byte    segOver;      // CS, DS, ES, SS if segment override
    byte    regi;         // 0 < regs < INDEXBASE <= index modes
};
```

2.-4. entfallen können, und führt dadurch möglicherweise zu einer höheren Geschwindigkeit. Es muß allerdings überprüft werden, ob hierdurch der Platzbedarf signifikant erhöht wird; andererseits entfallen die genannten Zusatzinformationen, und laut [28] besteht die resultierende Signatur-Datei typischerweise ohnehin zu ca. 95% aus den Bezeichnern der Bibliotheksfunktionen.

5.3.5 Umfangreiche Neustrukturierung des Codes

Bei der Erweiterung von `ndcc` stellte sich heraus, daß viele Änderungen durch einige stark an den Befehlssatz des i8086 angepaßte Details der Implementation von `dcc` erschwert wurden.

Zum einen fiel dies bei der Behandlung der in Kapitel 5.1.3 beschriebenen zusätzlichen i80186-Befehle auf; problematisch war hierbei, daß die Typen der Operanden (Register, Konstante, Speicherzelle) zunächst nicht explizit dargestellt werden, sondern sich lediglich implizit aus der Belegung mehrerer Variablen und Flags ergab. Die Interpretation dieser impliziten Darstellung findet an sehr vielen Stellen in `dcc` statt und ist nicht immer einheitlich gelöst bzw. führte stellenweise zu falschen Ergebnissen, die in der Folge korrigiert werden mußten.

Ein anderes Beispiel hierfür ist die Darstellung der Adressierungsmodi, die in der bisherigen Form für die Unterstützung der zusätzlichen 32-Bit-Adressierungsmodi nicht geeignet ist. Zur Zeit geschieht die Repräsentation eines Operanden (bis auf konstante *Immediate*-Operanden) durch die in Listing 5.7 abgebildete Struktur, wobei der Typ des Operanden dem Element `regi` zu entnehmen ist:

- Der Wert 0 steht für einen Speicherzugriff auf eine direkt angegebene Adresse; die Adresse befindet sich in den übrigen Einträgen, wobei hiervon für Win32 aufgrund der nicht mehr verwendeten Register lediglich der Wert von `off`

relevant ist.

- Die folgenden Werte 1 bis 22 werden für die Prozessor-Register verwendet.
- Die Werte 24 bis 31 stehen für die acht verschiedenen Adressierungsmodi; ein ggf. vorhandenes Displacement befindet sich wiederum im Eintrag `off`.

Wie sich erkennen läßt, ist für die Berücksichtigung der erweiterten 32-Bit-Adressierungsmodi eine völlig andere Repräsentation notwendig, die weitreichende Änderungen an `ndcc` mit sich brächte. (Der Sinn der vorliegenden Darstellungsweise besteht offenbar hauptsächlich darin, die für die Datenflußanalyse benötigten Definitions/Benutzt-Informationen möglichst effizient gewinnen zu können.)

Es erscheint daher sinnvoll, weite Teile des Codes von *ndcc* neu zu implementieren, um die beschriebenen Probleme nicht nur zu umgehen, sondern eine stabile Basis für Erweiterungen zu erhalten. Geeignet wäre insbesondere ein objektorientierter Ansatz, um eine Kapselung der Daten zu erreichen; dies könnte die Abhängigkeit von der konkret gewählten Repräsentation verringern, die, wie beschrieben, in der jetzigen Implementation für viele Probleme verantwortlich ist.

6 Zusammenfassung und Ausblick

Aufgabe dieser Arbeit war es, die grundsätzliche Dekompilierbarkeit von Win32-Malware zu untersuchen sowie einen existierenden Decompiler so zu erweitern, daß die für eine Dekompilierung in Frage kommende Malware damit analysiert werden kann.

Bei der Untersuchung einer Auswahl von Win32-Malware stellte sich heraus, daß der Maschinencode erwartungsgemäß bei vielen der analysierten Samples verschleiert worden war, was eine direkte Disassemblierung oder Dekompilierung zunächst verhindert hätte. Allerdings war für die Verschleierung nur in einem Fall eine individuell programmierte Routine benutzt worden. Im übrigen waren ausschließlich bereits vorhandene Packer und Verschlüsselungswerkzeuge zum Einsatz gebracht worden, so daß insgesamt nur eine begrenzte Auswahl verschiedener Entschleierungsroutinen vorgefunden wurden. Grundsätzlich bestanden daher gute Aussichten, die Verschleierungsschichten zu entfernen.

Ein im folgenden unternommener Versuch, die verschleierten Samples zu entschleiern, war überwiegend erfolgreich. In einigen Fällen konnten keine spezifischen Tools für das bei der Verschleierung der betreffenden Samples verwendete Verfahren gefunden werden. In anderen Fällen existierten zwar spezifische Tools, die aber entgegen ihrer Spezifikation die vorliegenden Samples nicht entschleiern konnten. Mit den darüber hinaus zur Verfügung stehenden generischen Entschleierungstools konnten zusätzlich einige weitere Samples entschleiert werden, die allerdings jeweils mit demselben Packer erzeugt worden waren. Trotz dieser praktischen Widrigkeiten erscheinen die Aussichten für die automatisierte Entschleierung der übrigen Samples prinzipiell günstig; eine Entwicklung entsprechender Tools hätte jedoch den Rahmen dieser Arbeit gesprengt.

Alle entschleierten sowie alle originär unverschleierten Samples wurden daraufhin untersucht, von welchem Compiler sie erzeugt worden waren. Auf eine Analyse derjenigen Samples, die nicht mit den oben genannten Tools hatten entschleiert werden können, wurde in diesem Arbeitsgang verzichtet, da der zusätzliche Erkenntniswert gering gewesen wäre und in keinem Verhältnis zum Aufwand einer manuellen Entschleierung gestanden hätte.

Die Analyse ergab, daß mehr als 90% der untersuchten Samples von nur wenigen verschiedenen Win32-Compilern erzeugt worden und damit grundsätzlich der

Dekompilierung zugänglich waren. Lediglich zwei Samples waren vermutlich in Assembler programmiert worden und damit ebenso ungeeignet wie weitere fünf, bei denen dies aus verschiedenen anderen Gründen der Fall war (vgl. Kap. 4.6).

Dieses positive Ergebnis gibt Grund zu der Annahme, daß die meiste nicht-virale Win32-Malware grundsätzlich für eine maschinelle Dekompilierung geeignet ist: Einerseits können etwaige Verschleierungsschichten prinzipiell maschinell entfernt werden. Andererseits wurde die betreffende Malware zum überwiegenden Teil mit Hochsprachencompilern übersetzt, wodurch problematische Befehlsfolgen und andere ungünstige Eigenschaften, wie sie z. B. in Assemblerprogrammen vorkommen, vermutlich generell nur selten auftreten.

Der als Grundlage dienende prototypische Decompiler `dcc`, der lediglich MS-DOS-Programme unterstützt, wurde zu dem prototypischen Decompiler `ndcc` weiterentwickelt.

`ndcc` ist zusätzlich auch für die Dekompilierung von Win32-Programmen geeignet. Um dies zu ermöglichen, war es unter anderem notwendig, `dcc` um die zusätzlichen Maschinenbefehle des i80386-Prozessors sowie die von diesem unterstützte 32-bittige Arbeitsweise zu erweitern. Zusätzlich mußte das Laden von Programmen implementiert werden, die im Win32-spezifischen PE-Format vorliegen; dies lieferte durch die Auswertung der im PE-Format vorliegenden Verwaltungsinformationen zusätzlich die Namen der aus DLLs importierten Funktionen.

Bei den an `ndcc` vorzunehmenden Erweiterungen ergaben sich verschiedene Probleme, die auf Beschränkungen in `dcc` zurückzuführen waren. Diese bestanden teilweise darin, daß `dcc` entgegen den Angaben der Entwickler nicht den vollständigen Befehlssatz des i80286-Prozessors unterstützte: Die beim i80286 hinzugekommenen Maschinenbefehle waren zwar rudimentär implementiert worden und wurden somit korrekt disassembliert, jedoch traten in späteren Analysephasen Fehler auf, die zur Erzeugung von inkorrektem Code oder zu Programmabbrüchen führten.

Zudem stellte sich bei der Implementierung von zusätzlichen Maschinenbefehlen – u. a. auch bei den fehlenden i80286-spezifischen Befehlen – heraus, daß die `dcc`-interne Repräsentation der für die Dekompilierung benötigten Daten stark auf die spezifischen Eigenschaften des i8086 und insbesondere auf die von diesem unterstützten Befehle ausgerichtet war, was die Erweiterung schwieriger gestaltete. Aus diesem Grund wurde unter anderem auf die vollständige Implementierung aller 32-Bit-Adressierungsmodi verzichtet.

Bei der Dekompilierung eines zu Testzwecken verwendeten Programms zeigte sich, daß `ndcc` in der Lage war, das PE-Dateiformat korrekt einzulesen und zu verarbeiten; insbesondere wurden die Namen der verwendeten DLL-Funktionen richtig wiedergegeben. Die in diesem Programm verwendeten Maschinenbefehle wurden korrekt disassembliert, und der erzeugte Quelltext entsprach bis auf den für einen Parameter verwendeten Typ in den wesentlichen Punkten dem Original.

Im folgenden wurde `ndcc` anhand der bereits für die grundsätzlichen Untersuchungen verwendeten Malware-Samples auf die erzielte Funktionalität hin untersucht. Es stellte sich heraus, daß die Verarbeitung des PE-Dateiformats und der Maschinenbefehle offenbar auch bei komplexeren Programmen im Rahmen der Implementation korrekt erfolgten, allerdings brach die Dekompilierung in allen Fällen vorzeitig ab, was möglicherweise auf Beschränkungen des in `dcc` implementierten Dekompilierungsverfahrens zurückzuführen ist. Die Ursache für einen derartigen Abbruch wurde detailliert bei einem der vorliegenden Samples untersucht. Dabei wurde festgestellt, daß der Abbruch auf die fehlerhafte Verarbeitung eines Maschinenbefehls zurückzuführen war, der bereits im i8086-Befehlssatz vorhanden war. Der Fehler ließ sich somit auf eine bereits in `dcc` bestehende Lücke zurückführen.

Um eine erfolgreiche Dekompilierung bei komplexeren Programmen zu erzielen, wäre es notwendig, verschiedene Verbesserungen und Erweiterungen an `ndcc` vorzunehmen. Hierunter fiele zunächst die Vervollständigung des unterstützten Befehlssatzes; dabei stünden zunächst die Unterstützung der String-Befehle und aller 32-Bit-Adressierungsmodi im Vordergrund. Des weiteren wäre die Implementierung von Compiler- und Bibliothekssignaturen für die Dekompilierung von Hochsprachenprogrammen von entscheidender Bedeutung.

Eine Weiterentwicklung von `ndcc` erscheint allerdings nur dann sinnvoll, wenn zunächst einige grundsätzliche Änderungen an der Architektur und an der verwendeten Datenrepräsentation vorgenommen werden, da diesbezügliche Mängel allgemein die Implementierung erschwerten und zu einigen Fehlern führten, die nur mit erhöhtem Aufwand zurückverfolgt werden konnten.

Wie gezeigt wurde, ist die Dekompilierung von Win32-Malware grundsätzlich möglich. Um zu einer auch in der täglichen Praxis einsetzbaren Lösung zu kommen, ist es allerdings notwendig, der Dekompilierungsthematik in der universitären und kommerziellen Forschung mehr Aufmerksamkeit zu widmen, da die inhärente Komplexität eines Decompilers die anderer Reverse-Engineering-Werkzeuge deutlich übersteigt. In Anbetracht der Möglichkeiten, die ein weiterentwickelter Decompiler eröffnen könnte, erscheint dieser Aufwand jedoch lohnenswert.

7 Glossar

Alignment: Ausrichtung von Daten an Adressen, die an bestimmten Vielfachen eines Bytes liegen. Beispielsweise muß ein Objekt mit einem 512-Byte-Alignment stets an einer durch 512 teilbaren Speicheradresse beginnen.

DOS-Stub: DOS-EXE-kompatibler Abschnitt am Anfang einer PE-Datei.

Header: *Hier:* Abschnitt am Anfang einer Programmdatei, der Verwaltungs-und Struktur-Informationen enthält.

Image: Binäres Abbild einer Programmdatei, insbesondere im Hauptspeicher; umfaßt bei DOS-`.exe`-Dateien nur den eigentlichen Programmcode, bei Win32-`.exe`-Dateien auch die Header.

Image Base: Bevorzugte Basis-Adresse für das Image einer PE-Datei im Hauptspeicher, wird im PE-Header angegeben.

Immediate: Operand eines Maschinenbefehls, der im Anschluß an diesen unmittelbar als Konstante vorliegt, im Gegensatz zu Speicher- oder Register-Operanden.

Opcode: Kurz für *Operation Code*. Der Opcode ist der Teil eines Maschinenbefehls, der die eigentliche Operation beschreibt. Die Operanden gehören nicht zum Opcode im eigentlichen Sinne.

Protected Mode: Betriebsmodus, über den CPUs ab dem i80286 verfügen; wird von allen modernen Betriebssystemen genutzt.

Prototyp: Beschreibung der Typen des Rückgabewerts sowie der Parameter einer Funktion.

Program Segment Prefix: 256 Bytes langer Verwaltungsabschnitt eines DOS-Programms, der beim Laden erzeugt wird und üblicherweise direkt vor diesem im Speicher liegt.

Relokation: Anpassung der in einem Programm vorkommenden Adressen an den beim Laden tatsächlich verwendete Adreßbereich.

Sample: *Hier:* Exemplar eines bösartigen Programms, z. B. eines Virus, in Form einer Datei.

viral: Virus-Eigenschaften besitzend.

Wildcard: Joker-Zeichen, das für jedes andere Zeichen stehen kann.

Win32: 32-Bit Windows-Plattform (z. Zt. Windows 95, 98, ME, NT, 2000 und XP).

8 Abkürzungsverzeichnis

API	Application Program(ming) Interface
AVP	AntiViral Toolkit Pro
AVTC	Anti-Virus Test Center, Arbeitsbereich AGN, Fachbereich Informatik, Universität Hamburg
CARO	Computer Antivirus Researchers' Organization
DLL	Dynamic Link Library (während der Laufzeit dynamisch eingebundene Bibliothek unter Win32)
ELF	Enhanced Link Format (Dateiformat für UNIX-Programme)
JNE	Jump if Not Equal (Assemblerbefehl)
JNZ	Jump if Not Zero (Assemblerbefehl)
MS-DOS	Microsoft Disk Operating System
MSIL	Microsoft Intermediate Language (in der .NET-Architektur von Microsoft verwendeter P-Code)
N/A	Not Available
NAI	Network Associates, Inc.
NE	New Executable (Dateiformat für Win16-Programme)
PE	Portable Executable (Dateiformat für Win32-Programme)
PSP	Program Segment Prefix (von DOS-Programmen benötigte Verwaltungsstruktur)
PSW	Password-Stealing-Ware
RVA	Relative Virtual Address
UDM	Universal Decompilation Module
URL	Uniform Resource Locator
VBA	Visual Basic for Applications (Makrosprache der Microsoft Office Anwendungen)

9 Literaturverzeichnis

[1] ASPack. *ASPack – Powerful executable compressor,* `http://www.aspack.com/aspack.htm`

[2] *AVP Virus Encyclopedia,* `http://www.avp.ch/avpve`

[3] Bontchev, Vesselin; Skulason, Fridrik; Solomon, Dr. Alan. *A New Virus Naming Convention,* CARO (Computer Antivirus Researchers' Organization), 1991 `ftp://ftp.elf.stuba.sk/pub/pc/text/naming.zip`

[4] Bontchev, Vesselin. *Analysis and Maintenance of a Clean Virus Library,* Proceedings of the third International Virus Bulletin Conference, 1993 `http://www.virusbtn.com/OtherPapers/VirLib/`

[5] Bontchev, Vesselin; Brunnstein, Klaus; Dierks, Jörn. *Macro Virus List,* 2001 `ftp://agn-www.informatik.uni-hamburg.de/pub/texts/macro/macrol_s.00c`

[6] Brinkley, D. *Intercomputer transportation of assembly language software through decompilation,* Technical report, Naval Underwater Systems Center, 1981

[7] Brunnstein, Klaus. *Computer-Viren-Report,* 2. Auflage 1991

[8] Brunnstein, Klaus. *Neue Informationstechnologien – Neue Kriminalität, Neue Bekämpfungsmethoden: Herausforderungen an die Forensische Informatik,* BKA-Arbeitstagung 1996

[9] Brunnstein, Klaus. *From AntiVirus to AntiMalware Software and Beyond: Another Approach to the Protection of Customers from Dysfuncional System Behaviour,* 22nd National Information Systems Security Conference 1999 `http://csrc.nist.gov/nissc/1999/proceeding/papers/p12.pdf`

[10] Chikofsky, Elliot J.; Cross, James H. II. *Reverse Engineering and Design Recovery: A Taxonomy,* IEEE Software 1/1990

[11] Cifuentes, Cristina. *Reverse Compilation Techniques,* PhD dissertation, Queensland University of Technology, School of Computing Science, 1994 `ftp://ftp.it.uq.edu.au/pub/CSM/dcc/decompilation_thesis.ps.gz`

[12] Cohen, Fred. *Computer Viruses – Theory and Experiments,* 1984 http://all.net/books/virus/

[13] Collake Software. *PECompact: An advanced Win32 executable compressor,* http://www.collakesoftware.com/PECompact/CSPECompact.htm

[14] DataRescue. *IDA Pro – The Interactive Disassembler,* http://www.datarescue.com/idabase

[15] *Decafe Pro – The Java decompiler,* http://decafe.hypermart.net

[16] Dijkstra, Edsger W. *Go To Statement Considered Harmful,* Communications of the ACM 11(3)/1968 http://www.acm.org/classics/oct95/

[17] Emmerik, Mike Van. *Signatures for Library Functions in Executable Files,* Queensland University of Technology, Faculty of Information Technology, 1994 http://www.itee.uq.edu.au/~cristina/tr-2-94-qut.ps

[18] *Enlarge – Unpacker for Petite v2.1, v2.2,* http://www.absolutelock.de/construction/releases.html

[19] Evseenko, Vitaly. *Win32 Intro – Intelligent Executable Unpacker/Dumper,* http://madmat.hypermart.net/w32intro.htm

[20] Farmer, Mick. *Network Security* http://penguin.dcs.bbk.ac.uk/~mick/academic/networks/bsc/security/index.shtml

[21] Firestone, Roger M. http://web.archive.org/web/20000605230314/www.cacr.caltech.edu/~rfire/humor.html#compsci

[22] Fleisch, Brett D. *Malicious code (Part 2) and Java Decafeinated*, Department of Computer Science, University of California, Riverside, 2001 http://www.cs.ucr.edu/~brett/cs165_s01/LECTURE8/lecture8and9-4up.pdf

[23] Ford, Daniel. *Jive : A Java decompiler,* IBM San Jose Research Lab., 1996

[24] Freeman, Jay. *Anakrino MSIL -> C# Decompiler,* http://www.saurik.com/net/exemplar

[25] Friedman, Frank L. *Decompilation and the Transfer of Mini-Computer operating systems : Portability of systems oriented assembly language programs,* PhD dissertation, Dept of Computer Science, Purdue, 1974

[26] Frisk Software International. *F-Prot Antivirus,* http://www.f-prot.com

[27] Graham, Paul. *Hijacking is Buffer Overflow,* 2001.
`http://www.paulgraham.com/paulgraham/hijack.html`

[28] Guilfanov, Ilfak. *Fast Library Identification and Recognition Technology*
`http://www.datarescue.com/idabase/flirt.htm`

[29] Helger, Philip. *GetTyp 2000 – File format analyzer,*
`http://www.unet.univie.ac.at/~a9606653/gettyp/gettyp.htm`

[30] Hollander, Clifford R. *Decompilation of Object Programs,* PhD Thesis, Stanford University, 1973

[31] i+ Software. *DCI+: Decompiler for .NET* `http://www.iplussoftware.com/default.asp?Doc=20`

[32] Intel Corporation. *Intel 80386 Programmer's Reference Manual 1986,* 1987
`http://webster.cs.ucr.edu/Page_TechDocs/Doc386/0_toc.html`

[33] Intersimone, David. *Antique Software: Turbo C version 2.01,* Borland Software Corporation 2000
`http://community.borland.com/article/0,1410,20841,00.html`

[34] Janz, André. *Reverse Engineering: Rechtliche Rahmenbedingungen und praktische Möglichkeiten,* Studienarbeit, Universität Hamburg, Fachbereich Informatik, 2000
`http://agn-www.informatik.uni-hamburg.de/papers/stud2000.htm`

[35] Kallnik, Stephan; Pape, Daniel; Schröter, Daniel; Strobel, Stefan. *Sicherheit: Buffer-Overflows,* c't 23/2001, `http://www.heise.de/ct/01/23/216/`

[36] Kaspersky Lab Int. *Kaspersky Anti-Virus (AVP),* `http://www.kaspersky.com`

[37] Luck, Ian. *PEtite Win32 Executable Compressor,*
`http://www.un4seen.com/petite/`

[38] Ludloff, Christian. *sandpile.org – IA-32 architecture,*
`http://www.sandpile.org/ia32/index.htm`

[39] Lüvelsmeyer, Bernd. *The PE file format,*
`http://webster.cs.ucr.edu/Page_TechDocs/pe.txt`

[40] NeoWorks. *NeoLite: Program Encryption & Compression for Developers,*
`http://www.neoworx.com/products/neolite/`

[41] Network Associates. *McAfee AVERT Virus Information Library,*
`http://vil.nai.com`

[42] Network Associates. *McAfee VirusScan,* `http://www.mcafee.com`

[43] Gold, Steve. *First 'Proof Of Concept' .Net Virus Appears,* 2002 `http://www.newsbytes.com/news/02/173540.html`

[44] Oberhumer, Markus F.X.J.; Molnár, Lázló. *UPX – the Ultimate Packer for eXecutables,* `http://upx.sourceforge.net`

[45] Proebsting, T.A.; Watterson, S.A. *Krakatoa: decompilation in Java (does bytecode reveal source?,* Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS), 1997

[46] Remotesoft Inc. *Salamander .NET Decompiler,* `http://www.remotesoft.com/salamander`

[47] Shipp, Alex. *Forumsbeitrag,* 16.12.2001. `http://upx.sourceforge.net/phpBB/viewtopic.php?topic=6&forum=3&11`

[48] Simon, Istvan. *A Comparative Analysis of Methods of Defense against Buffer Overflow Attacks,* California State University, Hayward, 2001 `http://www.mcs.csuhayward.edu/~simon/security/boflo.html`

[49] SourceTec Software Co., Ltd. *SourceTec Java Decompiler,* `http://www.srctec.com/decompiler/`

[50] Symantec Corporation. *Norton AntiVirus,* `http://www.symantec.com/nav/nav_9xnt/`

[51] *UnPECompact: Unpacker for exe files being compressed with any PECompact version,* 2000. `http://mitglied.lycos.de/yoda2k/Proggies.htm`, `http://y0da.cjb.net/`

[52] Virus Bulletin Ltd. *Project VGrep,* `http://www.virusbtn.com/VGrep/`

[53] Warezak, Piotr; Wierzbicki, Rafal. *WWPack32 advanced EXE/DLL compressor for Windows,* `http://www.wwpack32.venti.pl`

[54] WingSoft. *WingDis – A Java Decompiler,* `http://www.wingsoft.com/wingdis.html`

[55] Wreski, Dave. *Linux Security Interview with David A. Wheeler,* 2000. `http://search.linuxsecurity.com/feature_stories/feature_story-6.html`

[56] Yoo, C. *A study of the program disassembly using flow analysis techniques,* Seoul, Korea, Advanced Institute of Science and Technology, Computer Science Dept, 1985

[57] Zeltser, Lenny. *Reverse Engineering Malware,* 2001
`http://www.zeltser.com/sans/gcih-practical/`

A Untersuchte Malware-Samples

Pfad + Dateiname	CARO (abgeleitet)	AVP	NAI	Compiler	Packer
DUNPWS\MSDUN-C.EX	trojan://W32/PSW.Msdun.c	Trojan.PSW.Msdun.c	DUNpws.bi	Borland C++	-
PLAGE2K\INETD.VXE	worm://W32/Plage@M	I-Worm.Plage	W32/Plage.gen@M	Borland C++	-
DUNPWS\HACKOF.EX	trojan://W32/PSW.Hackof	Trojan.PSW.Hackof	DUNpws.bh	Delphi 3/4	-
ICQPWS\ICUP.EXE	trojan://W32/PSW.Icup	Trojan.PSW.Icup	ICQpws	Delphi 3/4	-
W32SKA2K\SKA.EXE	worm://W32/Ska@M	I-Worm.Happy	W32/Ska@M	Delphi 3/4	-
W32SKAC\HAPPY99.EXE	worm://W32/Ska@M	I-Worm.Happy	W32/Ska@M	Delphi 3/4	-
ZIPPEDFI\ZIPPEDFI.NE2	worm://W32/ExploreZip.b@M	I-Worm.ZippedFiles	W32/ExploreZip.worm.pak.b@M	Delphi 3/4	Neolite
W32SKA@M\HAPPY99.EXE	worm://W32/Ska@M	I-Worm.Happy	-	Delphi 3/4	UPX
PRETTYPA\PRETTY-G.EX	worm://W32/PrettyPark.G@MM	I-Worm.PrettyPark	W32/Pretty.gen@MM	Delphi 3/4	WWPack32
PRETTYPA\PRETTY-A.EX4	worm://W32/PrettyPark.H@MM	I-Worm.PrettyPark	W32/Pretty.gen@MM	Delphi 3/4	WWPack32
PRETTYPA\PRETTY-L.EX	worm://W32/PrettyPark.L@MM	I-Worm.PrettyPark.wwpack	W32/Pretty.gen@MM	Delphi 3/4	WWPack32
PRETTYPA\PRETTY-A.WWP	worm://W32/PrettyPark@MM	I-Worm.PrettyPark.wwpack	W32/Pretty.gen@MM	Delphi 3/4	WWPack32
ICQ2K\ICQ2K-C.EXE	trojan://W32/Icq2k	Trojan.Win32.Icq2k	ICQ2K	Delphi 4	-
DUNPWS\TESTSP-C.EE	trojan://W32/PSW.TestSpy.c	Trojan.PSW.TestSpy.c	DUNpws.bk	MSVC++ 4.2	-
DUNPWS\TESTSP-C.EX	trojan://W32/PSW.TestSpy.c	Trojan.PSW.TestSpy.c	DUNpws.bk	MSVC++ 4.2	-
DUNPWS\COCED\COCE235.EX	trojan://W32/PSW.Coced.235.a	Trojan.PSW.Coced.235.a	PWS.gen	MSVC++ 5.0	-
DUNPWS\CONF.EXE	trojan://W32/PSW.Coced.236	Trojan.PSW.Coced.236	PWS.gen	MSVC++ 5.0	-
DUNPWS\CONFGUIN.EXE	trojan://W32/PSW.Coced.236.b	Trojan.PSW.Coced.236.b	PWS.gen	MSVC++ 5.0	-
DUNPWS.CK\111\GIPWIZAR.EXE	trojan://W32/PSW.Gip.111	Trojan.PSW.Gip.111	DUNpws.ck.cfg	MSVC++ 5.0	-
DUNPWS\WINPIC32.EXE	trojan://W32/PSW.Hooker.a	Trojan.PSW.Hooker.a	DUNpws.av	MSVC++ 5.0	-
DUNPWS\WINKEY.DLL	trojan://W32/PSW.Hooker.b	Trojan.PSW.Hooker.b	DUNpws.av.dll	MSVC++ 5.0	-
DUNPWS.CK\BETA\MTUSPEED.EXE	trojan://W32/PSW.ICQ.Spaels	Trojan.PSW.ICQ.Spaels	DUNpws.ck	MSVC++ 5.0	-
QAZ-ABCD\PS000810.ZIP/QAZ-A.WEE	worm://W32/Qaz.a	Worm.Qaz	W32/QAZ.worm.gen	MSVC++ 5.0	-
QAZ-ABCD\PS000810.ZIP/QAZ-B.WEE	worm://W32/Qaz.b	Worm.Qaz	W32/QAZ.worm.gen	MSVC++ 5.0	-
QAZ-ABCD\PS000810.ZIP/QAZ-C.WEE	worm://W32/Qaz.c	Worm.Qaz	W32/QAZ.worm.gen	MSVC++ 5.0	-
QAZ-ABCD\PS000810.ZIP/QAZ-D.WEE	worm://W32/Qaz.d	Worm.Qaz	W32/QAZ.worm.gen	MSVC++ 5.0	-
ICQPWS\TEMP$01.EXE	trojan://W32/PSW.Coced.233	Trojan.PSW.Coced.233	ICQpws.gen	MSVC++ 5.0	ASPack
DUNPWS\COCED\COCE235.EDR	trojan://W32/PSW.Coced.235.a	Trojan.PSW.Coced.235.a	DUNpws.az.dr	MSVC++ 5.0	ASPack
DUNPWS\COCED\COCE235.EX1	trojan://W32/PSW.Coced.235.a	Trojan.PSW.Coced.235.a	DUNpws.az.dr	MSVC++ 5.0	ASPack
DUNPWS\COCED\COCE235.EX2	trojan://W32/PSW.Coced.235.a	Trojan.PSW.Coced.235.a	DUNpws.az.dr	MSVC++ 5.0	ASPack
DUNPWS\PRICE.EXE	trojan://W32/PSW.Coced.236	Trojan.PSW.Coced.236	DUNpws.bm.gen	MSVC++ 5.0	ASPack

Pfad + Dateiname	CARO (abgeleitet)	AVP	NAI	Compiler	Packer
DUNPWS\NS236C0.EXE	trojan://W32/PSW.Coced.236.b	Trojan.PSW.Coced.236.b	DUNpws.bm.gen	MSVC++ 5.0	ASPack
DUNPWS\NS236CI.EXE	trojan://W32/PSW.Coced.236.b	Trojan.PSW.Coced.236.b	DUNpws.bm.gen	MSVC++ 5.0	ASPack
DUNPWS\NS236CJ.EXE	trojan://W32/PSW.Coced.236.b	Trojan.PSW.Coced.236.b	DUNpws.bm.gen	MSVC++ 5.0	ASPack
DUNPWS\NS236CQ.EXE	trojan://W32/PSW.Coced.236.b	Trojan.PSW.Coced.236.b	DUNpws.bm.gen	MSVC++ 5.0	ASPack
DUNPWS\NS236CW.EXE	trojan://W32/PSW.Coced.236.b	Trojan.PSW.Coced.236.b	DUNpws.bm.gen	MSVC++ 5.0	ASPack
DUNPWS\NS236CZ.EXE	trojan://W32/PSW.Coced.236.b	Trojan.PSW.Coced.236.b	DUNpws.bm.gen	MSVC++ 5.0	ASPack
DUNPWS\NS2363.EXE	trojan://W32/PSW.Coced.236.d	Trojan.PSW.Coced.236.d	DUNpws.bm.gen	MSVC++ 5.0	ASPack
DUNPWS\NS237.EXE	trojan://W32/PSW.Coced.236.e	Trojan.PSW.Coced.236.e	DUNpws.bm.gen	MSVC++ 5.0	ASPack
DUNPWS\NS237DIR.EXE	trojan://W32/PSW.Coced.236.e	Trojan.PSW.Coced.236.e	DUNpws.bm.gen	MSVC++ 5.0	ASPack
DUNPWS\NS237ICQ.EXE	trojan://W32/PSW.Coced.236.e	Trojan.PSW.Coced.236.e	DUNpws.bm.gen	MSVC++ 5.0	ASPack
DUNPWS\NS237SET.EXE	trojan://W32/PSW.Coced.236.e	Trojan.PSW.Coced.236.e	DUNpws.bm.gen	MSVC++ 5.0	ASPack
DUNPWS\NS237WRD.EXE	trojan://W32/PSW.Coced.236.e	Trojan.PSW.Coced.236.e	DUNpws.bm.gen	MSVC++ 5.0	ASPack
DUNPWS\NS237ZIP.EXE	trojan://W32/PSW.Coced.236.e	Trojan.PSW.Coced.236.e	DUNpws.bm.gen	MSVC++ 5.0	ASPack
DUNPWS.CK\GIP.EXE	trojan://W32/PSW.Coced.240	Trojan.PSW.Coced.240	PWS.gen	MSVC++ 5.0	ASPack
DUNPWS.CK\KERN98.EXE	trojan://W32/PSW.Coced.240	Trojan.PSW.Coced.240	PWS.gen	MSVC++ 5.0	ASPack
DUNPWS.CK\SLSHOW2.EXE	trojan://W32/PSW.Coced.240	Trojan.PSW.Coced.240	PWS.gen	MSVC++ 5.0	ASPack
DUNPWS.CK\SPEDIA.EXE	trojan://W32/PSW.Coced.240	Trojan.PSW.Coced.240	-	MSVC++ 5.0	ASPack
DUNPWS.CK\1.EXE	trojan://W32/PSW.Coced.241	Trojan.PSW.Coced.241	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\2.EXE	trojan://W32/PSW.Coced.241	Trojan.PSW.Coced.241	-	MSVC++ 5.0	ASPack
DUNPWS.CK\GIP110.EXE	trojan://W32/PSW.Gip.110.a	Trojan.PSW.Gip.110.a	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\111\GIP111EX.EXE	trojan://W32/PSW.Gip.111	Trojan.PSW.Gip.111	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\111\GIP111JP.EXE	trojan://W32/PSW.Gip.111	Trojan.PSW.Gip.111	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\113\CONFIG.EXE	trojan://W32/PSW.Gip.113	Trojan.PSW.Gip.113	DUNpws.ck.cfg	MSVC++ 5.0	ASPack
DUNPWS.CK\113\GIP113DO.EXE	trojan://W32/PSW.Gip.113	Trojan.PSW.Gip.113	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\113\GIP113JP.EXE	trojan://W32/PSW.Gip.113	Trojan.PSW.Gip.113	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\113\GIPWIZAR.EXE	trojan://W32/PSW.Gip.113	Trojan.PSW.Gip.113	DUNpws.ck.cfg	MSVC++ 5.0	ASPack
DUNPWS.CK\KERNEL32.EXE	trojan://W32/PSW.Gip.113.b	Trojan.PSW.Gip.113.b	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\SCAN.EXE	trojan://W32/PSW.Gip.113.b	Trojan.PSW.Gip.113.b	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\111\CALC.EXE	trojan://W32/PSW.Gip.based	Trojan.PSW.Gip.based	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\111\WINUPDT2.EXE	trojan://W32/PSW.Gip.based	Trojan.PSW.Gip.based	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\ISOAQ.EXE	trojan://W32/PSW.Gip.based	Trojan.PSW.Gip.based	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\SLSHOW.EXE	trojan://W32/PSW.Gip.based	Trojan.PSW.Gip.based	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\110\GIP110~1.EXE	trojan://W32/PSW.Gip.MrNop	Trojan.PSW.MrNop	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\110\GIP110~2.EXE	trojan://W32/PSW.Gip.MrNop	Trojan.PSW.MrNop	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\110\GIP110~3.EXE	trojan://W32/PSW.Gip.MrNop	Trojan.PSW.MrNop	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\110\GIP110~4.EXE	trojan://W32/PSW.Gip.MrNop	Trojan.PSW.MrNop	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\BETA\MTUSPDDR.EXE	trojan://W32/PSW.ICQ.Spaels	Trojan.PSW.ICQ.Spaels	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS.CK\BETA\WINUPDT1.EXE	trojan://W32/PSW.ICQ.Spaels	Trojan.PSW.ICQ.Spaels	DUNpws.ck	MSVC++ 5.0	ASPack
DUNPWS\AJAN\AJANCONF.EX	trojan://W32/PSW.Ajan.10	Trojan.PSW.Ajan.10	DUNpws.ax.cfg2	MSVC++ 5/6.0	-
DUNPWS\AJAN\AJANSERV.EX	trojan://W32/PSW.Ajan.10	Trojan.PSW.Ajan.10	DUNpws.ax	MSVC++ 5/6.0	-
DUNPWS\TEMP#01.EXE	trojan://W32/PSW.Stealth.a	Trojan.PSW.Stealth.a	DUNpws.at	MSVC++ 6.0	ASPack

Pfad + Dateiname	CARO (abgeleitet)	AVP	NAI	Compiler	Packer
DUNPWS\SPOOLSRV.EXE	trojan://W32/PSW.Stealth.d	Trojan.PSW.Stealth.d	DUNpws.au	MSVC++ 6.0	ASPack
DUNPWS\MSEXT.EXE	trojan://W32/PSW.Ext	Trojan.PSW.Ext	DUNpws.aw	MSVC++ 6.0	-
WINSOUND\WINSOUND.EXE	worm://W32/Winsound	Win32.HLLP.Winfig	Winsound	VB 5.0	-
DUNPWS\AC\REGEDIT.EXE	trojan://W32/AOL.Stealth	Trojan.AOL.Stealth	DUNpws.ac.gen	VB 6.0	PECompact
DUNPWS\EA\BRAIN.EXE	trojan://W32/PSW.Brain	Trojan.PSW.Brain	DUNpws.ea	VB 6.0	PECompact
W32FIX\FIX2001.EXE	worm://W32/Fix2001@M	I-Worm.Fix2001	W32/Fix.12288@M	n.i.	-
MANDRAGORE\mandragore.VXE	worm://W32/Gnuman	Gnutella-Worm.Mandragore	W32/Gnuman.worm	n.i.	-
W32HYBRI\WSOCK32.DLL	worm://W32/Hybris.b@MM	I-Worm.Hybris.b	W32/Hybris.dll@MM	infizierte DLL	-
W32SKA2K\WSOCK32.DLL	worm://W32/Ska@M	I-Worm.Happy	W32/Ska.dll@M	infizierte DLL	-
DUNPWS\AJAN\AJANBASE.EX	trojan://W32/PSW.Ajan.10	Trojan.PSW.Ajan.10	DUNpws.ax.dr	N/A	Armadillo
DUNPWS\AJAN\AJANBIND.EX	trojan://W32/PSW.Ajan.10	Trojan.PSW.Ajan.10	DUNpws.ax.cfg	N/A	Armadillo
DUNPWS.CK\110\CONFIG.EXE	trojan://W32/PSW.Gip.110.a	Trojan.PSW.Gip.110.a	DUNpws.ck.cfg	N/A	Armadillo
DUNPWS.CK\111\CONFIG.EXE	trojan://W32/PSW.Gip.111	Trojan.PSW.Gip.111	DUNpws.ck.cfg	N/A	Armadillo
DUNPWS\AC\SDE16.EXE	trojan://W32/PSW.Stealth.d	Trojan.PSW.Stealth.d	DUNpws.ac	N/A	Armadillo
Navidad\NAVIDAD.EX	worm://W32/Navidad@M	I-Worm.Navidad.a	W32/Navidad.gen@M	N/A	Armadillo
ZIPPEDFI\ZIPPEDFI.NEO	worm://W32/ExploreZip.a@M	I-Worm.ZippedFiles	W32/ExploreZip.worm.pak.a@M	N/A	NeoLite
W32EXZIP\FRANCH.EXE	worm://W32/ExploreZip.e@M	I-Worm.ZippedFiles	W32/ExploreZip.worm.pak.e@M	N/A	NeoLite
ZIPPEDFI\ZIPPEDFI.NE3	worm://W32/ExploreZip.e@M	I-Worm.ZippedFiles	W32/ExploreZip.worm.pak.e@M	N/A	NeoLite
DUNPWS\BADBOY.PAC	trojan://W32/PSW.BadBoy	Trojan.PSW.BadBoy	DUNpws.bf	N/A	PECompact
DUNPWS\PLATAN\PLATAN-E.EX	trojan://W32/PSW.Mail777	Trojan.PSW.Mail777	DUNpws.be	N/A	PECompact
DUNPWS\PLATAN\PLATAN-B.EX	trojan://W32/PSW.Pec.a	Trojan.PSW.Pec.a	DUNpws.bb	N/A	PECompact
DUNPWS\PLATAN\PLATAN-C.EX	trojan://W32/PSW.Pec.d	Trojan.PSW.Pec.d	DUNpws.bc	N/A	PECompact
DUNPWS\PLATAN\A\PHOTO.EX	trojan://W32/PSW.Pec.e	Trojan.PSW.Pec.e	DUNpws.ba	N/A	PECompact
DUNPWS\PEC-F.EX	trojan://W32/PSW.Pec.f	Trojan.PSW.Pec.f	DUNpws.bj	N/A	PECompact
DUNPWS\PLATAN\PLATAN-C.EX1	trojan://W32/PSW.Platan.c	Trojan.PSW.Platan.c	DUNpws.bc	N/A	PECompact
DUNPWS\PLATAN\PLATAN-C.EX2	trojan://W32/PSW.Platan.c	Trojan.PSW.Platan.c	DUNpws.bc	N/A	PECompact
DUNPWS\PLATAN\PLATAN-D.EX	trojan://W32/PSW.Platan.d	Trojan.PSW.Platan.d	DUNpws.bd	N/A	PECompact
ZIPPEDFI\ZIPPEDFI.PEC	worm://W32/ExploreZip.c@M	I-Worm.ZippedFiles	W32/ExploreZip.worm.pak.c@M	N/A	PECompact
DUNPWS\AC\SDEP1.EXE	trojan://W32/PSW.Stealth.d	Trojan.PSW.Stealth.d	DUNpws.ac.gen	N/A	Petite
DUNPWS.W\SAVER.EXE	trojan://W32/PSW.Kuang.c	Trojan.PSW.Kuang.c	DUNpws.w	N/A	UPX
DUNPWS\AC\SDEP5.EXE	trojan://W32/PSW.Stealth.d	Trojan.PSW.Stealth.d	DUNpws.ac.gen	N/A	UPX
W32HYBRI\DWARF4YO.EXE	worm://W32/Hybris.b@MM	I-Worm.Hybris.b	W32/Hybris.gen@MM	N/A	unbekannt
W32EXZIP\EXPLZIWW.EXE	worm://W32/ExploreZip.d@M	I-Worm.ZippedFiles	W32/ExploreZip.worm.pak.d@M	N/A	WWPack32

Tabelle A.1: Detaillierte Liste der Malware-Samples