Diplomarbeit

Christian Morgenstern

Entwicklung und Evaluierung eines generischen Verfahrens für Malware-Scanner zum Entpacken geschützter Schadprogramme

Christian Morgenstern

Entwicklung und Evaluierung eines generischen Verfahrens für Malware-Scanner zum Entpacken geschützter Schadprogramme

Diplomarbeit eingereicht im Rahmen der Diplomprüfung
im Studiengang Softwaretechnik
am Studiendepartment Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr.-Ing. Martin Hübner
Zweitgutachter : Prof. Dr. Gunter Klemke

Abgegeben am 18. Juli 2007

Christian Morgenstern

Thema der Diplomarbeit

Entwicklung und Evaluierung eines generischen Verfahrens für Malware-Scanner zum Entpacken geschützter Schadprogramme

Stichworte

Laufzeitpacker, Entpacken, Schadprogramme, Malware, Erkennung

Kurzzusammenfassung

Bekannte Schadprogramme können mittels sogenannter Laufzeitpacker mit minimalem Aufwand vor einer Wiedererkennung durch Malware-Scanner geschützt werden. Damit ist eine nahezu unbegrenzt häufige Wiederverwendung einmal entwickelter Schadprogramme möglich. Das im Rahmen dieser Arbeit entwickelte generische Verfahren soll so geschützte Schadprogramme in ihre ursprüngliche und damit erkennbare Form rücktransformieren. Dies ist nicht in jedem Fall möglich, da einige hierbei zu lösende Probleme auf das nicht lösbare Halteproblem reduziert werden können. Um dennoch in vielen Fällen korrekte Ergebnisse erzielen zu können, wurden die in dieser Arbeit entwickelten Heuristiken angewendet. Anhand der Tests, die mit dem hier entwickelten Prototyp durchgeführt wurden, konnte gezeigt werden, dass eine hohe Erfolgsquote bei der Rücktransformation erreicht werden kann.

Christian Morgenstern

Title of the paper

Design and evaluation of a generic procedure usable by malware scanners for unpacking packed malware

Keywords

runtime packers, unpacking, malware, detection

Abstract

Little effort is required to use runtime packers to avoid detection of known malware by malware scanners, thus with different packers existing malware can be re-used almost indefinitely. The procedure developed in this paper can be used to transform packed malware back into its original detectable representation. Success is not guaranteed in all cases as some of the encountered problems can be reduced to the unsolvable halting problem, though in this paper heuristics have been developed and applied to allow correct results to be obtained in most cases. Tests that were performed using the implemented prototype indicate that a high rate of success for transforming malware back into its original representation can be achived.

Inhaltsverzeichnis

1 Einleitung

1.1 Allgemeines

Schadprogramme können auf verschiedenen Wegen auf ein System gelangen und dort ausgeführt werden; mit oder auch ohne Nutzerinteraktion. Dies geschieht beispielsweise in Form eines Trojanischen Pferdes oder Wurms[1]. Das ausgeführte Schadprogramm kann dann, je nach Typ und Funktionalität, unter anderem auf andere Systeme weiterverbreitet werden oder auch einem Angreifer volle Kontrolle über das System ermöglichen.

Aufgrund der Verbreitung von Baukastensystemen[2] zur Erstellung von Schadprogrammen ist es auch Personen ohne Programmierkenntnisse möglich, solche Schadprogramme nach eigenen Vorgaben zu erstellen. Weiterhin können als Quelltext vorliegende Schadprogramme einfach den Anforderungen eines Angreifers angepasst werden.

Hieraus ergibt sich ein hoher Verbreitungsgrad zahlreicher Varianten weniger Grundtypen. Malware-Scanner können diese Varianten zunächst problemlos anhand von Signaturen und Heuristiken erkennen[3].

1.2 Problemstellung

Schadprogramme können vor der Erkennung durch Malware-Scanner geschützt werden, indem sie in ein anderes Format transformiert werden. Wenn die Repräsentation eines Schadprogramms im neuen Format deutlich von der des ursprünglichen Formats abweicht, kann es nicht mehr direkt von Malware-Scannern mittels Signaturvergleichen und Heuristiken erkannt werden.

Ein Beispiel für eine solche Transformation ist das Packen in das verbreitete ZIP-Archivformat. Für dieses Format liegt eine Spezifikation[4] vor. Wenn eine der Spezifikation entsprechende Entpackroutine in Malware-Scannern implementiert ist, können im Archiv enthaltene transformierte Schadprogramme intern entpackt und ggf. erkannt werden.

---

[1] Siehe [Shirey, 2000] sowie [Eckert, 2001, Seite 42ff].

[2] Beispielsweise Optix: http://www.megasecurity.org/trojans/o/optix/Optixlite0.5.html 08.07.2007.

[3] [Muttik, 2000, Seite 3], siehe auch beispielsweise die AgoBot-Varianten: http://www3.ca.com/securityadvisor/virusinfo/virus.aspx?ID=37776 08.07.2007.

[4] http://www.pkware.com/documents/casestudies/APPNOTE.TXT 08.07.2007.

Allerdings können enthaltene Schadprogramme auch erkannt werden, obwohl das Archivformat nicht vom Malware-Scanner unterstützt wird. Denn ein enthaltenes Schadprogramm muss vor der Ausführung zunächst von einer bereits vorhandenen Anwendung entpackt und als Datei im Ursprungsformat zwischengespeichert werden. Somit könnte ein Schadprogramm spätestens dann –, unabhängig von der Unterstützung eines Malware-Scanners für das Archivformat – erkannt werden, wenn der Malware-Scanner Schreibvorgänge überwacht.

Wenn ein rücktransformiertes Schadprogramm vor der Ausführung nicht zwischengespeichert und das verwendete Format vom Malware-Scanner nicht erkannt oder nicht unterstützt wird, kann auch das Schadprogramm vor dessen Ausführung nicht erkannt werden. In so einem Fall wäre es also möglich, dass eigentlich zuverlässig erkannte Schadprogramme ohne vorherige Erkennung ausgeführt werden können. Die Möglichkeit der Rücktransformation und Ausführung ohne Zwischenspeicherung wird in den folgenden Absätzen näher erläutert.

Aufgrund der von heutigen Computern verwendeten Von-Neumann-Architektur erfolgt keine Trennung von Programmcode und Daten [Giloi, 1984, Seite 61]. Es ist also möglich, dass sich Programme zur Laufzeit selbst modifizieren, beispielsweise entschlüsseln, und somit zuvor nicht sichtbaren Code ausführen.

Somit kann ein bestehendes Schadprogramm nachträglich transformiert, beispielsweise verschlüsselt oder gepackt werden. Damit es trotzdem ausführbar bleibt, wird normalerweise eine Routine hinzugefügt, welche die zuvor angewendete Transformation rückgängig macht. Der Programmablauf startet dann bei dieser Routine. Nachdem sie das eigentliche Programm im Arbeitsspeicher in seine Ursprungsform zurücktransformiert hat, kann es normal ausgeführt werden [Christodorescu u. a., 2005b, Seite 10]. Da die Rücktransformation während der Ausführung geschieht, ist dabei keine Zwischenspeicherung notwendig. Dies ist ein typisches Merkmal sogenannter laufzeitgepackter Dateien.

Nur wenn der Malware-Scanner ein gepacktes Schadprogramm in seine ursprüngliche Form transformieren kann, besteht die Möglichkeit, es anhand von Signaturen oder Heuristiken zu erkennen[5]. Es ist aber laut [Natvig, 2002, Seite 2] wichtig, auch transformierte – und somit unbekannte – Schadprogramme erkennen zu können, da diese über das Internet innerhalb von wenigen Sekunden viele Nutzer erreichen können.

Für gepackte und zunächst nicht erkannte Schadprogramme kann schnell und einfach eine Signatur in Form eines Dateihashes erstellt werden, um diese spezifische Version nachträglich erkennen zu können. Allerdings vergehen nach Beobachtungen von [Marx, 2004] in der Regel einige Stunden, bis aktualisierte Signaturen bereitstehen. In der Zwischenzeit kann das gepackte Schadprogramm aber schon auf viele Systeme gelangt sein, da es von den eingesetzten Malware-Scannern nicht erkannt wird.

[5] [Vnuk u. Návrat, 2006, Seite 169], [Norman, 2003, Seite 15], [Szor, 2005, Kapitel 6.2 und 15.4.2].

Im Kontext dieser Diplomarbeit werden, sofern nicht anders angegeben, Laufzeitpacker und Laufzeitverschlüsselungsprogramme als äquivalent behandelt, da beide das Zielprogramm in Dateiform stark verändern und somit das Erkennen mittels einer Byte-Signatur verhindern. Sie werden nur noch allgemein als Laufzeitpacker oder kurz „Packer“ bezeichnet.

1.3 Motivation

Das konkrete Problem wird durch die freie Verfügbarkeit einer großen Anzahl von Laufzeitpackern noch verstärkt. Hiermit können beliebige Programme, also auch Würmer und Trojanische Pferde, mit wenigen Mausklicks gepackt oder verschlüsselt werden. Die ersten verfügbaren Programme dieser Art produzierten noch sehr einfach rücktransformierbare Ergebnisse. Es war also mit wenig Aufwand möglich, die eingebetteten statischen Transformationsroutinen zu analysieren und somit eine Unterstützung für die entsprechenden Formate in einen Malware-Scanner zu integrieren.

Inzwischen werden auch polymorphe Verschlüsselungstechniken verwendet. Hierbei wird die Entschlüsselungsroutine für jedes gepackte Programm verändert. Dies erschwert die Analyse und Erkennung des geschützten Programms [Szor, 2000, Seite 52]. Ohne Kenntnis der Funktionsweise dieser Routine ist ein direktes Entschlüsseln des geschützten Programms nicht möglich. Es können also keine einfachen statischen Routinen mehr verwendet werden, um solche geschützten Programme zu entschlüsseln. Dieses Problem tritt auch auf, wenn ein eigentlich vom Malware-Scanner unterstützter Packer geringfügig modifiziert[6] wird und so die damit gepackten Dateien möglicherweise nicht mehr als solche erkannt werden.

Die Analyse und Identifikation geschützter Schadprogramme ist also ein Problem. Da die Anwendung dieser Art des Schutzes keinerlei spezielle Kenntnisse voraussetzt, ist die Verbreitung nicht mehr direkt erkennbarer Schadprogramme sehr hoch. [Szor, 2000] bestätigt, dass spätestens seit 2000 für diverse Schadprogramme Laufzeitpacker verwendet werden, um eine Erkennung durch Heuristiken und bestehende Signaturen zu verhindern. Aus einer Analyse von [Morgenstern u. Brosch, 2006] geht hervor, dass 92% der in der WildList[7] 03/2006 zu findenden Malware gepackt ist. Im Vergleich dazu sind nur in etwa[8] die Hälfte der vom Autor im Jahr 2004 automatisiert gesammelten[9] Schadprogramme gepackt. Es ist also ein starker Anstieg der Anzahl gepackter Programme zu verzeichnen.

---

[6] [Graf, 2005], [Szor, 2005, Kapitel 15.4.2].

[7] http://www.wildlist.org/WildList/ 08.07.2007.

[8] Das Ergebnis beruht auf einem schnellen Test von knapp 1900 ausführbaren Dateien durch das Programm PEiD (http://peid.has.it/ 08.07.2007.) anhand statischer Signaturen. Es ist hierbei möglich, dass einige gepackte Programme nicht als solche erkannt werden.

[9] Die gesammelten Schadprogramme müssen nicht zwangsweise denen im gleichen Zeitraum in der Wildlist befindlichen Schadprogrammen entsprechen.

Es wurden 13 kommerzielle Anti-Virus-Produkte auf die Erkennung gepackter Malware durch [Johansen, 2005] getestet. Hierfür wurde der Nimda.A Wurm, welcher in seiner ursprünglichen Form von allen Produkten erkannt wurde, mit 18 verschiedenen Laufzeitpackern gepackt. Die gepackten Dateien wurden nur noch von 4% bis 81% der Produkte erkannt. Die durchschnittliche Erkennungsrate betrug lediglich 38%.

Eine mögliche Erklärung hierfür ist, dass in der Wildlist vereinzelt vorkommende Laufzeitpacker nur von wenigen Anti-Virus-Produkten unterstützt werden [Morgenstern u. Brosch, 2006]. Aufgrund der großen Anzahl an Laufzeitpackern, die mit Schadprogrammen verwendet werden können, nehmen nur wenige Hersteller von Anti-Virus-Software den Aufwand auf sich, Unterstützung für selten genutzte Laufzeitpacker in ihre Produkte zu integrieren [Schipka, 2006].

Da nicht mehr von einem Malware-Scanner erkennbare Malware schnell und einfach erstellt werden kann, ist diese auch weit verbreitet. Hier besteht also Lösungsbedarf.

Um einen zuverlässigeren Schutz zu gewährleisten, muss ein Malware-Scanner auch mit modifizierten oder unbekannten Laufzeitpackern gepackte Schadprogramme vor deren Ausführung erkennen können. Dieser Schutz ist aber, den Testergebnissen von [Johansen, 2005] zufolge, nicht gegeben. Nach [Morgenstern u. Brosch, 2006] ist der überwiegende Teil der verbreiteten Malware gepackt – eben weil dadurch eine Erkennung verhindert werden soll. Es muss eine Lösung gefunden werden, um eine bessere Erkennung zu ermöglichen. Gepackte Schadprogramme müssen also generisch zurück in ihre Ursprungsform, die oft erkannt werden kann, transformiert werden können.

Abgesehen von der direkten Erkennung durch Malware-Scanner, werden rücktransformierte Schadprogramme auch für andere Analysen benötigt, beispielsweise für die Klassifikation in Familien oder für die Analyse der Schadroutine [siehe Cifuentes u. a., 2001; Schultz u. a., 2001; Carrera u. Erdélyi, 2004; Christodorescu u. a., 2005a; SabreSecurity, 2006].

1.4 Zielsetzung

Ziel dieser Arbeit ist, ein generisches Verfahren zu entwickeln, mit dem laufzeitgepackte Schadprogramme entpackt werden können. Hierzu müssen diese – möglichst unabhängig von unbekannten oder modifizierten Laufzeitpackern – in ihre ursprüngliche Form rücktransformiert werden. Das Verfahren soll so konzipiert werden, dass es in einen Malware-Scanner integriert werden kann. Weiterhin soll anhand von Tests gezeigt werden, inwieweit das entwickelte Verfahren anwendbar ist.

Um laufzeitgepackte Programme generisch entpacken zu können, müssen zunächst Gemeinsamkeiten und Einschränkungen gebräuchlicher Laufzeitpacker analysiert werden. Bestehende Methoden, gepackte Schadprogramme automatisiert in ihre Ursprungsform zu transformieren, sind meistens nicht generisch oder unsicher und nicht für den Einsatz in

Malware-Scannern geeignet. Um ein Programm generisch entpacken zu können, wird dessen eigene Entpackroutine verwendet. Damit dieses Vorgehen keine Sicherheitsrisiken in sich birgt, wird die Ausführung nur emuliert.

Um das entpackte Programm durch die Emulation der Entpackroutine in jedem Fall erhalten zu können, müssen Lösungen für einige teilweise dem Halteproblem entsprechende Problemstellungen gefunden werden. Da das Halteproblem aber nicht lösbar ist, müssen Heuristiken zu den Teilproblemen erstellt werden, um das gewünschte Ergebnis erzielen zu können. Besonders relevant sind die Fragen, wann die Ausführung von der Entpackroutine in das entpackte Programm übergeht und in welchen Bereichen des Speichers das entpackte Programm zu finden ist.

Da die Entpackversuche aufgrund der Verwendung von Heuristiken nicht immer erfolgreich sein müssen, wird zu Testzwecken ein Prototyp implementiert. Mit diesem wird eine Vielzahl gepackter Programme wieder entpackt. Die entpackten Programme werden, im Gegensatz zu ihrer gepackten Form, durch einen Malware-Scanner erkannt. Hierdurch wird die Funktionstüchtigkeit der erarbeiteten Lösung belegt.

1.5 Struktur

Zunächst wird kurz der Begriff Schadprogramm erläutert, da eine Erkennung dieser Programme ermöglicht werden soll. Die hier betrachteten Schadprogramme liegen im Microsoft-PE-Format vor, welches zunächst grob skizziert wird. Mit diesem Grundwissen über das PE-Format kann daraufhin die Funktionsweise gebräuchlicher Laufzeitpacker erklärt werden. Hierzu werden auch kurz häufig verwendete, aber nicht die gegebene Problemstellung lösende Ansätze zum Entpacken von Malware vorgestellt.

Nach der Aufstellung von Anforderungen wird eine geeignete Lösung konzeptioniert. Diese beinhaltet zunächst die emulierte Ausführung des gepackten Programms sowie hierbei relevante Punkte wie Geschwindigkeit und Betriebssystemumgebung. Anschließend werden Möglichkeiten zur Überwachung des Entpackvorgangs aufgezeigt, die dann für eine Unterbrechung des Emulationsvorgangs an geeigneter Stelle und zum Auffinden des entpackten Programms genutzt werden. Das entpackte Programm muss nun nur noch als Datei gespeichert werden, um von gängigen Malware-Scannern geprüft werden zu können.

Entsprechend der beschriebenen Lösung wird ein Prototyp implementiert, und entsprechende Implementierungsdetails werden kurz erläutert. In mit dem Prototyp durchgeführten Tests wird die resultierende Steigerung der Erkennungsleistung von Malware-Scannern dokumentiert.

2 Grundlagen

Zunächst wird kurz auf die im Rahmen dieser Arbeit relevanten Eigenschaften von Schadprogrammen bzw. Malware eingegangen. Durch eine Beschreibung der Struktur des PE-Formats für ausführbare Dateien kann anschließend die Funktionsweise von Laufzeitpackern erklärt werden. Hierauf basierend werden auch verschiedene im weiteren Verlauf der Arbeit wichtige Einschränkungen aufgestellt. Anschließend werden noch einige häufig verwendete, aber nicht die gegebene Problemstellung lösende, Ansätze zum Entpacken von Malware vorgestellt.

2.1 Malware

Zunächst eine Zusammenfassung der geläufigen[1] Malware-Definition nach [Brunnstein, 1999]:

> Unter den Oberbegriff „Malware“ fällt jegliche Art von Software, die für den Verwender dieser Software nicht gewünschte Funktionen intentiell ausführt, beziehungsweise Funktionen ausführt, die das Programm laut seiner Dokumentation nicht vorgibt zu besitzen. Die ausgeführten Funktionen besitzen einen für den Benutzer der Software direkt oder indirekt schädlichen (engl „malicious“) Charakter, woraus sich der Begriff „Malware“ von „malicious software“ ableitet. [Freitag, 2000, Seite 7]

Die weitverbreitete Windows-Betriebssystemfamilie ist das Hauptziel von Malware [Bayer, 2005, Seite 2]. Dies liegt daran, dass dort eine breite Basis an potentiellen Zielen zur Verfügung steht [Lawton, 2002]. Im Rahmen dieser Diplomarbeit ist das Zielbetriebssystem deswegen auf einen mit Windows 2000 kompatiblen Typ beschränkt.

Ein Programm, also auch Malware, kann hier von einem regulären Nutzeraccount aus ausgeführt werden. Einige Arten von Malware erfordern aber auch Administrator-Rechte und führen möglicherweise Code auf Kernel-Ebene aus. [Bayer, 2005, Seite 2] empfiehlt eine Beschränkung der Analyse auf User-Mode-Malware, da die meiste Malware keinen Code auf Kernel-Ebene verwendet.

[1] Siehe auch [Krüger, 2003, Seite 9] und [Eggert u. a., 2003, Seite 2].

Da sich diese Arbeit mit dem Entpackvorgang beschäftigt, und nicht mit der Funktionsweise der Malware selbst, ist es hier nicht weiter relevant, ob die Malware zur Laufzeit Administrator-Rechte benötigen wird.

Auch ist es hier nicht relevant, ob die Malware in Form eines Trojanischen Pferdes oder Wurms verbreitet wird. Ob die Malware Schad- oder Hintertürfunktionalität enthält, spielt auch keine Rolle. Das einzig wichtige Kriterium ist, dass die Malware in Form einer ausführbaren Datei im im nächsten Kapitel erläuterten PE-Format vorliegt. Die Datei enthält hierbei ausschließlich die Malware sowie von der Malware genutzte Daten. Somit fallen Dateiviren, die vorhandene Programme infizieren, nicht unter die hier betrachtete Malware.

Inzwischen wird viel Malware in Hochsprachen geschrieben [Szor, 2005, Kapitel 6.2.6], [Muttik, 2000, Seite 7]. Dadurch besitzt diese meistens Struktureigenschaften, die direkt in Assembler geschriebene Programme in der Regel nicht besitzen. Dies kann für Erkennung und Heuristiken genutzt werden. Hierauf wird in den folgenden Kapiteln, besonders in Erläuterungen, noch näher eingegangen.

2.2 Das PE-Format – Ein Überblick

Nativ ausführbare Dateien liegen unter Microsoft Windows im PE-Format[2] vor. Ein Grundwissen über dieses Format wird benötigt, um später die Funktionsweise von Laufzeitpackern nachvollziehen zu können. Das PE-Format kann auch .NET-Bytecode enthalten. Solche Dateien werden hier aber nicht weiter betrachtet, da diese Arbeit ausschließlich Programme in nativ ausführbarem Maschinencode behandelt.

2.2.1 Dateisicht

Zunächst ein Überblick anhand eines stark vereinfachten Beispiels. Abbildung 2.1 stelle eine typische, von einem Hochsprachen-Compiler generierte Datei dar. Die Größenverhältnisse der einzelnen Elemente zueinander werden aufgrund des Platzbedarfs in allen das PE-Format betreffenden Abbildungen nicht originalgetreu dargestellt.

- Die Datei beginnt an Position 0 mit dem Header. Dieser enthält unter anderem Strukturinformationen über die ausführbare Datei, insbesondere die 1 bis 96 enthaltenen Sektionen, sowie den Einstiegspunkt (EP) zur Programmausführung. Die an der rechten Seite der Grafik in Hexadezimal angegebenen Positionen dienen nur der Illustration und können sich je nach Programm unterscheiden.

  Eine Sektion ist ein logischer Abschnitt der Datei. Es kann diverse Vorteile haben, ein Programm in Sektionen aufzuteilen. Dies betrifft unter anderem die Ladezeit und die

[2] Microsoft Portable Executable, [siehe Microsoft, 2006].

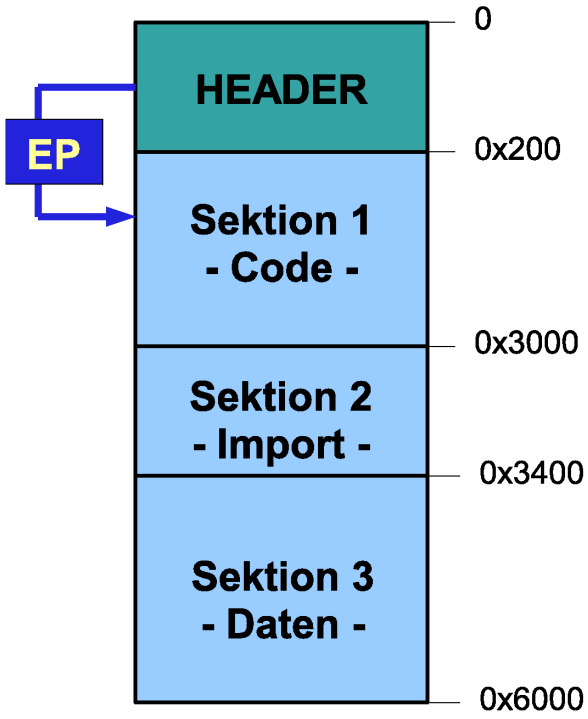

Abbildung 2.1: PE-Format – Dateisicht

Fehlererkennung. Die genannten Vorteile werden nicht weiter erläutert, da sie keine Relevanz für das behandelte Thema haben.

- An Position 0x200 beginnt Sektion 1, die den ausführbaren Code enthält. Der Header verweist auf die Position, die die erste auszuführende Anweisung des Programms enthält.
- Bei 0x3000 ist die Importsektion untergebracht. Sie enthält die Importtabelle. Diese enthält Informationen über die Abhängigkeiten des Programms, also welche externen Funktionen es benötigt. Hierbei handelt es sich in der Regel um Funktionen der Windows-API[3], da ein Programm normalerweise nur über diese mit der Betriebssystemumgebung interagieren kann[4]. Die dort eingetragenen Funktionen werden bei Programmstart in den Arbeitsspeicher geladen und ihre Adressen in vorgesehenen Bereichen des Programms gespeichert. So können sie vom Programm zur Laufzeit aufgerufen werden.
- Schließlich folgt bei 0x3400 die Datensektion. Hier sind feste Daten des Programms, also beispielsweise Texte, sowie globale Variablen untergebracht. Am Ende dieser Sektion endet auch die Datei.

[3] Application Programming Interface.

[4] Die Interaktion ist in der Praxis auch ohne die API-Funktionen mittels direkter Kernel-Aufrufe möglich [siehe Eilam, 2005, Seite 91ff], allerdings schränkt dies die Kompatibilität stark ein und wird hier nicht weiter betrachtet.

In der Praxis können die enthaltenen Elemente auch nahezu beliebig gemischt innerhalb einer einzigen Sektion untergebracht sein. Darauf wird aber zunächst nicht weiter eingegangen, da hier nur ein Überblick gegeben werden soll.

2.2.2 Arbeitsspeichersicht

Bevor ein Programm ausgeführt werden kann, muss es zunächst von dem Datenträger in den Arbeitsspeicher geladen werden. Dies geschieht anhand der im Header vorhandenen Strukturinformationen. Ein Programm erhält normalerweise einen 2 GB großen virtuellen Adressraum[5]. In diesem kann eine hohe Adresse belegt werden, ohne dass die niedrigeren Adressen zwangsweise auch genutzt werden. Daher wird das Programm an eine im Header nahezu beliebig spezifizierbare virtuelle Adresse (VA) in den Arbeitsspeicher geladen. In der folgenden Grafik ist dies 0x400000 – eine typische Adresse für vom Microsoft C-Compiler generierte Programme.

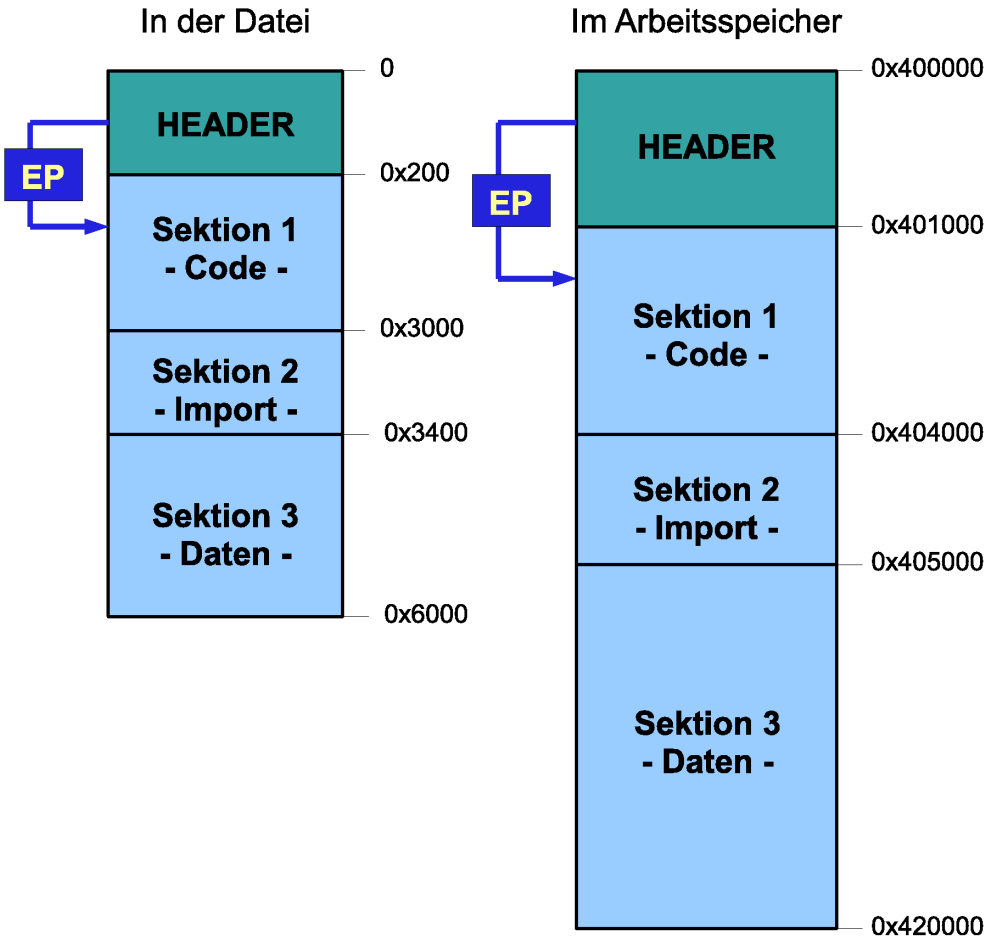

Abbildung 2.2: PE-Format – Arbeitsspeichersicht

[5] [Siehe Russinovich u. Solomon, 2004, Kapitel 7] für eine ausführliche Beschreibung der Speicherstruktur.

Abbildung 2.2 zeigt, wie die komplette Datei in den Arbeitsspeicher geladen wurde. Hier hat sich die Größe aller Sektionen im Vergleich zur Datei erhöht. Auch der Header belegt nun nicht mehr 512 Byte, sondern 4096 Byte. Dies hat den Hintergrund, dass Sektionen in der Dateirepräsentation an der typischen Größe eines Festplattensektors (512 Byte, 0x200 Hex) ausgerichtet sind. Im Arbeitsspeicher orientieren sich Größe und Position jedoch an der Größe einer Speicherseite (4096 Byte, 0x1000 Hex). Die Speicherbereiche, die hierbei nicht mit Daten aus der Datei beschrieben werden, sind normalerweise mit 0 initialisiert.

Auffallend ist, dass sich die Größe der Datensektion sehr stark über die Grenzen einer Speicherseite hinaus erhöht hat. Das liegt daran, dass in diesem Beispiel eine große Anzahl nicht initialisierter globaler Variablen verwendet wird, die in der Datensektion enthalten sind. Da diese nicht initialisiert sind, brauchen sie nicht im Dateiformat gespeichert zu sein – also auch keinen Platz zu belegen. Im Arbeitsspeicher wird die Größe der Datensektion entsprechend den Angaben im Header erhöht, um zusätzlichen Platz für diese Variablen zu schaffen.

Jede Sektion kann also in der Datei und im Arbeitsspeicher unterschiedlich viel Platz belegen. Dies ist eine Voraussetzung für die Funktionsweise vieler Laufzeitpacker.

2.3 Laufzeitpacker

Laufzeitpacker können verwendet werden, um die Größe einer ausführbaren Datei zu reduzieren und ihren Inhalt vor einfacher Ansicht sowie direkter Veränderung zu schützen. Hierzu darf die ursprüngliche ausführbare Datei nicht in ihrer Originalform auf einem Datenträger zwischengespeichert werden.

Die generelle Funktionsweise von Laufzeitpackern zu kennen ist für die Erkennung und die Vorgehensweise beim Entpackvorgang wichtig.

2.3.1 Funktionsweise

Basierend auf dem zuvor erläuterten PE-Format wird nun die Funktionsweise eines möglichen Laufzeitpackers erklärt. Dies ist für die spätere Erkennung und den Entpackvorgang relevant, zumal dieser nicht durch ein externes Programm, sondern durch die Datei selbst durchgeführt wird. Dies kann für den Nutzer transparent geschehen. Er wird also nicht bemerken, dass die Datei gepackt ist.

Zunächst der stark vereinfachte Packvorgang:

Das in Dateiform vorliegende Programm wird hier, wie in Abbildung 2.3 dargestellt, gepackt. Die Daten aller Sektionen des Programms werden komprimiert und aus den Sektionen entfernt. Zwischen Programmcode und Daten des ursprünglichen Programms wird hierbei nicht unterschieden. Die Originalsektionen haben nun eine Größe von 0 Byte,

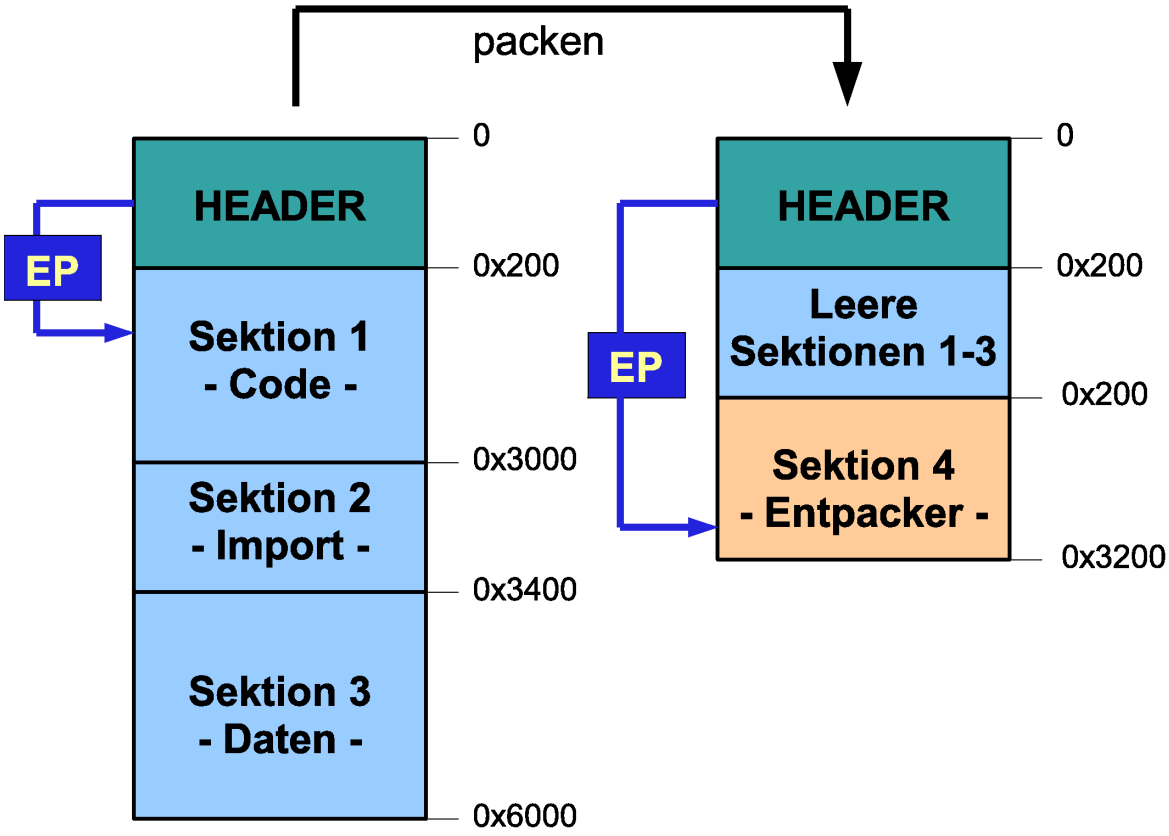

Abbildung 2.3: PE-Format gepackt – Dateisicht

belegen also keinen Platz mehr. Der Datei wird eine neue Sektion hinzugefügt, und die Strukturinformationen im Header werden entsprechend angepasst.

Die gepackten Daten werden zusammen mit der Entpackroutine und der von der Routine benötigten Importtabelle in Sektion 4 geschrieben. Das Programm belegt jetzt deutlich weniger Speicherplatz als zuvor. Durch die Anpassung des Einstiegspunktes wird die Programmausführung bei der Entpackroutine beginnen.

Wird das Programm gestartet, sieht es im Speicher genau wie das ursprüngliche Programm aus (vergleiche Abbildung 2.2), nur dass die ersten 3 Sektionen leer[6] sind, eine vierte Sektion anhängt ist und die Ausführung in dieser Sektion beginnt. Dies wird in Abbildung 2.4 dargestellt.

Die Entpackroutine dekomprimiert zunächst das ursprüngliche Programm und schreibt alle Daten in ihre ursprüngliche Sektion. Hierbei brauchen normalerweise keine zusätzlichen Speicherbereiche vom Betriebssystem angefordert zu werden. Der zum Ladezeitpunkt zugewiesene Speicher reicht also aus.

[6] Mit 0 initialisiert.

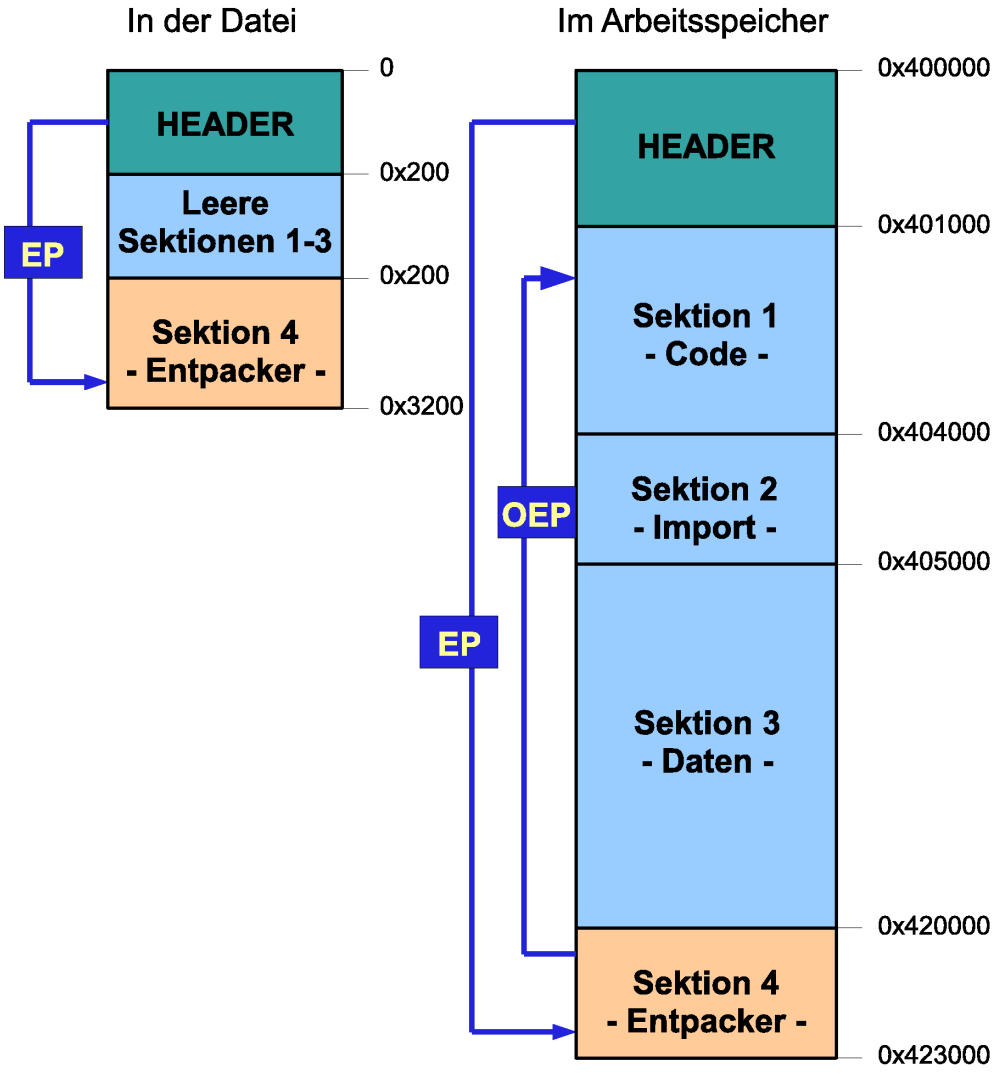

Abbildung 2.4: PE-Format Entpackvorgang – Arbeitsspeichersicht

Da die Importsektion auch komprimiert und somit vom Betriebssystem nicht beachtet wurde, müssen die vom ursprünglichen Programm benötigten externen Funktionen nun von der Entpackroutine geladen werden. Anschließend sieht das Programm im Arbeitsspeicher, wenn man vom modifizierten Header sowie der angehängten vierten Sektion absieht, genauso aus wie das ursprüngliche Programm wenn es, geladen worden wäre, ohne gepackt gewesen zu sein geladen worden wäre. Die Entpackroutine kann nun an den ursprünglichen/originalen Einstiegspunkt (OEP) springen, wodurch das ursprüngliche Programm normal ausgeführt wird, ohne auf einem Datenträger zwischengespeichert worden zu sein.

Analog hierzu wäre im Fall einer regulären Verschlüsselung zunächst jede Sektion verschlüsselt und eine Entschlüsselungsroutine in einer neuen Sektion hinzugefügt worden. Der durch jede Sektion belegte Platz hätte sich gegenüber der Originaldatei nicht verändert. Allerdings würde der von der Datei belegte Platz durch die hinzugefügte Sektion ansteigen. Die Ent-

schlüsselungsroutine würde jede Sektion wieder entschlüsseln, die importierten externen Funktionen laden und zum OEP springen.

2.3.2 Vielfalt

Es existieren, wie bereits in Abschnitt 1.3 angeführt, eine Vielzahl verschiedener Laufzeitpacker. Folgende Packer wurden in den Arbeiten von [Johansen, 2005], [Christodorescu u. a., 2005b] und [Royal u. a., 2006] in Tests verwendet oder bei der Analyse gepackter Malware identifiziert:

- Zip SFX
- RAR SFX
- Armadillo
- ASPack 2.12
- ASProtect 1.24 RC4
- EXE32pack 1.42
- EXECryptor 2.0
- ExeStealth 3.04
- FSG 2.0
- MEW11 SE 1.2
- MoleBox 2.3.3
- Morphine 2.7
- Obsidium
- Packman 0.0.0.1
- PECompact2 2.55
- PeX
- PE-PACK 1.0
- Petite 2.3
- UPack 0.27b
- UPX 1.25
- UPX-Scrambler[7]
- WWPack 1.20
- yoda's Crypter 1.3
- yoda's Protector 1.0b

[7] Kein Packer im eigentlichen Sinne. Allerdings modifiziert dieses Programm mit UPX gepackte Dateien, so dass diese zwar noch durch sich selbst, aber nicht mehr so einfach auf externem Wege erkannt/entpackt werden können.

Die erstgenannten Zip SFX und RAR SFX werden hier nicht weiter betrachtet, da sie entpackte Programme zunächst als Datei zwischenspeichern und somit nicht den Kriterien eines Laufzeitpackers genügen.

Die zuvor aufgeführten Packer werden, je nach Verfügbarkeit auch in unterschiedlichen Versionen, im Rahmen dieser Arbeit für Tests[8] verwendet und teilweise analysiert. Um ein größeres Spektrum abzudecken, werden hierfür noch zusätzlich folgende Packer genutzt:

- ACProtect 2.0
- BeRoEXEPacker 1.0
- EmbedPE 1.13
- eXPressor 1.451
- Hmimys 1.0
- Jdpack 1.01
- KByS Packer 0.28b
- Krypton 0.2
- NeoLite 1.01 und 2.0
- NoodleCrypt 2
- nPack 1.1b
- NSPack 3.7
- ped 0.1
- PelockNT 2.04
- PEncrypt 4
- PESpin 0.3
- PE-crypt 1.02
- polyene 0.01 plus
- RLPack 1.0b und 1.17
- shrinkwrap 1.4
- SimplePack 1.0 und 1.21
- tElock 0.42 und 0.98
- UPXCrypt, UPXRedir, UPXshit und UpolyX 0.5[9]
- Vgcrypt 0.75
- WinKript 1.0
- YZPack 2.0b

Insgesamt werden 51 verschiedene Laufzeitpacker[10] in 67 unterschiedlichen Versionen im Rahmen dieser Arbeit für die zuvor beschriebenen Zwecke genutzt. Eine detaillierte Auflistung hierzu ist im Anhang in Tabelle A.1 zu finden.

---

[8] Siehe Kapitel 6.

[9] Wie auch UPX-Scrambler kein Packer im eigentlichen Sinne.

[10] Inklusive der fünf zuvor genannten Modifikatoren für mit UPX gepackte Dateien.

2.3.3 Allgemeines und Einschränkungen

Dieser Abschnitt beschreibt weitere allgemeine, z. T. auch in anderen Arbeiten beschriebene, Eigenschaften von Laufzeitpackern sowie mögliche Sonderfälle. Hieran werden Einschränkungen festgelegt, auf deren Basis die weitere Arbeit aufbaut. Viele Dateien – besonders Malware enthaltende – sind heutzutage gepackt [Brulez u. Russel, 2005]. Malware ist sogar öfter in gepackter Form anzutreffen als normale Anwendungsprogramme [Gryaznov, 2006]. Dennoch kann nicht jedes gepackte Programm als Malware eingestuft werden, da auch einige normale Programme gepackt vorliegen. Es muss also zunächst der Inhalt überprüft werden.

Im Normalfall müssen alle für den Entpackvorgang notwendigen Daten[11] in der ausführbaren Datei enthalten sein[12]. Dies ist, besonders im Bereich von Malware, eine Voraussetzung für die transparente Ausführung. Laufzeitpacker, welche für den Entpackvorgang Nutzerinteraktion oder externe Informationen (eingelegte Original-CD, Kopierschutzstecker[13], Netzwerkübertragungen etc.) benötigen, werden somit ausgeschlossen.

Ein mit einem Laufzeitpacker gepacktes Programm enthält nach [Christodorescu u. a., 2005b, Seite 6] normalerweise diese Elemente:

1. Die Entpackroutine, also eine Anweisungssequenz, die neue Anweisungen erzeugt.
2. Eine Sprunganweisung, die zu dem von der Entpackroutine generierten Code springt.
3. Ein Speicherbereich, der das entpackte Programm enthält bzw. nach dessen Erzeugung enthalten wird.
4. Ein Speicherbereich, der das gepackte Programm enthält.

Es ist jedoch unklar, ob sich diese Auflistung auf die gepackte Datei, auf das Programm direkt nachdem es in den Speicher geladen worden ist, oder auf das Programm nach beendeter Entpackroutine bezieht. Auch wurde nicht konkretisiert, ob „enthält“ nun „innerhalb der Sektionsgrenzen des Programms“ oder „an einer beliebigen Stelle im virtuellen Adressraum des Programms“ bedeutet.

Ein gepacktes Programm braucht weder in Dateiform noch, sobald es in den Arbeitsspeicher geladen worden ist, einen Speicherbereich zu enthalten, in den das entpackte Programm geschrieben werden kann. Der benötigte Speicherbereich kann bei Bedarf zur Laufzeit vom Betriebssystem angefordert werden. Er befindet sich in diesem Fall nicht innerhalb der Sektionsgrenzen des Programms. Dies trifft unter anderem auf mit „Morphine“ gepackte Dateien zu. Daher wird nur vorausgesetzt, dass sich das entpackte Programm im eigenen[14]

[11] Zwischen Daten und Programmcode wird hier nicht unterschieden.

[12] Siehe [Eilam, 2005, Seite 330], [Christodorescu u. a., 2005b, Seite 6] und [Miller, 2007, Seite 1].

[13] Auch bekannt als „Hardware-Dongle“.

[14] Es ist möglich, ein Programm direkt in den virtuellen Adressraum eines anderen, zur selben Zeit laufenden Programms zu entpacken. Dies kann allerdings zu Problemen führen, wird von regulären Laufzeitpackern nicht gemacht und daher hier auch nicht zugelassen.

virtuellen Adressraum befindet.

Es existiert ein von [Miller, 2005] beschriebener Sonderfall, in dem ein verschlüsseltes Programm keinen Code zur Entschlüsselung und somit auch keine Quelldatensektion im eigentlichen Sinne enthalten muss. Hierbei werden sogenannte „Relocation“-Einträge entgegen ihrer typischen Verwendung – der Anpassung an eine Verschiebung im Speicher – genutzt, um ein Programm direkt vom Betriebssystem entschlüsseln zu lassen. Die Nutzung dieser Technik wurde laut [Szor, 2005, Kapitel 7.2] bei einigen Dateiviren entdeckt. Diese Art der Verschlüsselung ist einfach zu erkennen und zu entschlüsseln [siehe Miller, 2007, Seite 7], deswegen wird sie hier nicht weiter betrachtet. Ein gepacktes Programm muss also die komplette Entpackroutine enthalten oder zur Laufzeit generieren.

Eine Sprunganweisung, die von der Entpackroutine zum generierten Code springt, ist, wie [Royal u. a., 2006, Seite 4] feststellen, nicht zwingend erforderlich. Falls die erste auszuführende Anweisung des entpackten Programms auch die erste Anweisung der Codesektion ist und die Entpackroutine sich direkt vor der Codesektion, also wahrscheinlich am Ende des PE-Headers, befindet, kann die Ausführung direkt von der Entpackroutine zur ersten Anweisung übergehen. Dieses spezifische Verhalten wurde allerdings bei keiner analysierten gepackten Datei entdeckt und wird somit ausgeschlossen. Ein gepacktes Programm muss also eine Sprunganweisung zum entpackten Programm enthalten oder generieren. Hierbei gibt es aber eine erlaubte Ausnahme. „yoda’s crypter 1.2“ und „ExeStealth 2.73“ verwenden SEH, einen Fehlerbehandlungsmechanismus des Betriebssystems, um durch einen provozierten Fehler zum entpackten Code zu springen (siehe Abschnitt A.2). Programme, die Betriebssystemfunktionalität nutzen um zum entpackten Programm zu springen, brauchen also keine entsprechende Sprunganweisung zu enthalten.

Ein Speicherbereich, der das gepackte Programm enthält, braucht nicht enthalten zu sein, wenn das entpackte Programm direkt durch eine Anweisungssequenz generiert werden kann [siehe Miller, 2005, Seite 12]. Die Nutzung dieser Technik würde die Größe des transformierten Programms aber stark erhöhen.

Somit wird, entgegen der Auflistung von [Christodorescu u. a., 2005b, Seite 6] nur noch vorausgesetzt, dass ein gepacktes Programm in Dateiform eine Entpackroutine und die Sprunganweisung zum entpackten Programm enthält oder zur Laufzeit generiert, sofern keine Betriebssystemfunktionalität hierfür genutzt wird.

In der Arbeit von [Christodorescu u. a., 2005b, Seite 6] werden zwei Voraussetzungen angegeben, die ein gepacktes Programm erfüllen muss, damit es im Rahmen der zuvor genannten Arbeit automatisch entpackt werden kann.

1. Kein bereits ausgeführter Code darf mit dynamisch erzeugtem Code überschrieben werden. Dies bedingt auch, dass die Codeerzeugung abgeschlossen ist, bevor Code aus dem Speicherbereich für das entpackte Programm ausgeführt wird.
2. Der Entpackvorgang darf nicht von Eingaben oder der Laufzeitumgebung des Programms abhängen.

Basierend auf eigener Analyse führen mit „UPack 3.99“ gepackte Programme allerdings zunächst Code aus, der später mit dem entpackten Programm überschrieben wird (siehe Abschnitt A.2). Mit „PECompact 2“ gepackte Programme führen sogar Code am späteren Einstiegspunkt des entpackten Programms aus. „AsPack 2.11“ schreibt zur Laufzeit eine Anweisung in den Speicherbereich des später entpackten Programms, führt diese aus und fährt mit dem Entpackvorgang fort, wodurch diese Anweisung mit dem entpackten Programm überschrieben wird (siehe Abschnitt A.2). Um dieses einfach mögliche Verhalten nicht auszuschließen, wird im Rahmen dieser Arbeit nicht vorausgesetzt, dass ausgeführter Code nicht durch anderen Code überschrieben und kein Code vor Ende des Entpackvorgangs im Speicherbereich des entpackten Programms ausgeführt wird.

Die Unabhängigkeit von Eingaben und der Laufzeitumgebung ist eine sinnvolle Voraussetzung. Hierdurch wird sichergestellt, dass prinzipiell die Möglichkeit besteht, ein laufzeitgepacktes Programm automatisch zu entpacken. Allerdings kann eben genau diese Umgebungsabhängigkeit von Packern verwendet werden, um einen automatischen Entpackvorgang zu verhindern. Deswegen gilt auch dieser Punkt, der in späteren Abschnitten noch diskutiert wird, als keine unabdingbare Voraussetzung. Dennoch muss die gepackte Datei, wie am Anfang dieses Abschnitts festgelegt, alle Daten enthalten, die für den Entpackvorgang notwendig sind. Dies deckt den möglichen Fall ab, dass die Entpackroutine zwar das enthaltene Programm entpacken könnte, dies aber nach einer Prüfung von Umgebungsparametern nicht tut. Dieses Verhalten kann nach [Stepan, 2005, Seite 45] als Schutz gegen Entpackversuche eingesetzt werden.

Die Entpackroutine arbeitet in der Regel auf den bei Programmstart in den Arbeitsspeicher geladenen Daten. Sonderfälle, die noch die eigene Datei öffnen und Daten nachladen oder eine temporäre Datei anlegen müssen, werden hier nicht weiter betrachtet.

Sobald die erste Anweisung des entpackten Programms ausgeführt wird, muss das Speicherabbild des normal gestarteten Programms in dem Speicherabbild des entpackten Programms enthalten sein. „Enthalten sein“ heißt, dass es nicht von Bedeutung ist, ob vor oder nach dem Programm noch andere Daten im Speicher vorhanden sind. Das Programm muss nur vollständig und unverändert[15] entpackt worden sein. Hierbei gibt es allerdings drei Ausnahmen:

- Der PE-Header braucht nicht dem des Originalprogramms zu entsprechen. Einige compilergenerierte Programme können, nachdem sie in den Arbeitsspeicher geladen worden sind, auch ohne den Header funktionieren. Daher speichern viele Packer den Originalheader nicht. Allerdings speichern einige Packer Daten oder Programmcode im Header. Wird ein Programm zweimal mit einem entsprechenden Laufzeitpacker gepackt, und der Header hierbei nicht gesichert, ist der zweite Entpackvorgang aufgrund der fehlenden Daten nicht erfolgreich.

[15] Dies ist eine Voraussetzung dafür, dass das entpackte Programm von einem Malware-Scanner genauso gut wie das ungepackte Originalprogramm erkannt werden kann. Daher werden auch keine Fälle betrachtet, in denen ein Zielprogramm manuell an einen Laufzeitpacker angepasst wurde.

- Die Importtabelle braucht nicht vorhanden zu sein. Sie wird nur für Programme benötigt, deren Abhängigkeiten durch Windows geladen werden. Meistens erledigt dies aber der Laufzeitpacker, um den Platz der unkomprimierten Importtabelle einzusparen. Auch kann die Importtabelle explizit gelöscht werden, um eine Weiterverbreitung des entpackten Programms zu verhindern, da dies ohne Importtabelle in der Regel nicht lauffähig ist. Trotzdem müssen aber alle Abhängigkeiten geladen und für das entpackte Programm nutzbar sein, so als wäre dies direkt durch das Betriebssystem geschehen.
- Lücken zwischen den Sektionen brauchen nicht mit 0 initialisiert zu sein. Programme könnten sich jedoch auf diese Initialisierung verlassen und fehlerhaft arbeiten, wenn dies nicht der Fall ist. Lücken können durch die Ausrichtung an bestimmten Bytegrößen entstehen. Angenommen, der Programmcode belege 1700 Byte. Da die Größe der Codesektion ein Vielfaches von 512 ist, wird die Codesektion 2048 Byte umfassen. Somit sind am Ende der Sektion 348 ungenutzte Bytes vorhanden. Analog zu einem Cavity Virus[16] könnte diese Lücke zur Unterbringung einer Entschlüsselungsroutine genutzt werden. Dies hätte den Vorteil, dass die Dateigröße beibehalten wird und keine weitere Sektion hinzugefügt werden muss.

Da der PE-Header nicht dem des Originalprogramms entsprechen muss, könnte sich die logische Struktur des entpackten Programms von der des Originalprogramms unterscheiden. Der Packer könnte alle Sektionen des Originalprogramms zu einer einzigen Sektion kombiniert haben, um weiteren Platz einzusparen. Solange hierbei dennoch alle Programmteile an ihre ursprüngliche virtuelle Adresse geladen werden, ist dies zulässig.

2.4 Gebräuchliche Entpackmethoden

Seit es Laufzeitpacker gibt, besteht auch Bedarf, die gepackten Programme wieder zu entpacken und zu speichern, damit der Inhalt analysiert oder verändert werden kann. Da einige Programme auch mit Packern geschützt werden sollen, wurden einige Packer so erstellt, dass sie bestimmte Entpackmethoden erschweren. Im Folgenden werden kurz einige Methoden, mit denen Programme entpackt werden können, sowie markante Vor- und Nachteile aufgezählt.

- Das Programm starten und anschließend ein Speicherabbild erstellen.

  Hierbei wird ausgenutzt, dass das Programm sich bei der Ausführung selbst entpackt. Daher ist diese Methode sehr schnell, generisch, schwer durch das Programm erkennbar und mit wenig Aufwand anwendbar. Allerdings ist dies auch sehr unsicher, da das Programm normal ausgeführt wird. In Malware enthaltene Schadroutinen könnten also auch ausgeführt werden.

[16] Siehe [Szor, 2005, Kapitel 4.2.6].

Je nach Programm und Packer kann diese Methode auch möglicherweise nicht erfolgreich sein. Ein sehr kurzes Programm könnte bereits beendet sein, bevor versucht wird, das Speicherabbild zu erstellen. Auch könnte das Speicherabbild zu früh erstellt werden, wenn das Programm noch nicht komplett entpackt ist. Weiterhin könnte sich das Programm zur Laufzeit selbst modifizieren, beispielsweise indem nicht mehr benötigte Codeabschnitte nach der Verwendung überschrieben werden, um eine Erkennung anhand dieser Codeabschnitte zu verhindern.

Mit dieser Methode hat man also keine Kontrolle über das Programm. Da das Abbild nicht exakt zu dem Zeitpunkt, an dem die erste Anweisung des entpackten Programms ausgeführt wird, erstellt wird und auch nicht immer bekannt ist, wo sich das entpackte Programm im Speicher befindet, erhält man möglicherweise unbrauchbare Resultate.

- Das Programm starten, direkt vor Ausführung des entpackten Programms anhalten und ein Speicherabbild erstellen.

  Indem das Programm rechtzeitig angehalten wird, kann ein unverändertes Speicherabbild erstellt werden. Hierzu müsste man aber zunächst die genaue Anweisung bestimmen, an der die Entpackroutine beendet ist beziehungweise das entpackte Programm beginnt. Diese zu bestimmen kann, je nach Packer und Programm, schwierig sein. Es besteht prinzipiell das Risiko, dass die vermeintliche Entpackroutine Schadcode enthält, welcher zusammen mit ihr ausgeführt wird.

  Um das Programm an der gewünschten Stelle anzuhalten, kann man die Debugfunktionalität des Prozessors nutzen oder das Programm entsprechend verändern, beispielsweise an der entsprechenden Stelle einen Fehler provozieren oder eine Endlosschleife einbauen. Führt das Programm einen Selbsttest durch, bevor der Haltepunkt erreicht wird, kann es die Modifikation erkennen und sich, je nach Programmierung, selbst beenden oder sogar die Modifikation rückgängig machen. Die Verwendung der Debugfunktionalität des Prozessors, also Einzelschrittausführung oder gesetzte Haltepunkte, kann laut [Bayer u. a., 2006, Seite 3] vom Programm erkannt werden. Hier besteht auch das Risiko, dass das Programm aus dieser kontrollierten Ausführung ausbricht [Szor, 2005, Kapitel 11.4].

  Diese Methode bietet eine begrenzte Kontrolle über den Programmablauf. Wenn alles wie beabsichtigt gelingt, erhält man ein vollständiges und unverändertes Speicherabbild. Allerdings besteht die Möglichkeit, dass hierbei Schadcode zur Ausführung kommt. Enthält die Entpackroutine entsprechenden Code, könnte sie das rechtzeitige Anhalten des Programms verhindern.

- Anwendung der zuvor genannten Verfahren innerhalb einer virtuellen Maschine.

  Durch die Verwendung einer virtuellen Maschine wird eine Abkapselung vom eigenen System erreicht. Falls bei einem Entpackversuch Schadcode ausgeführt wird, so kann

er sich nur[17] auf die virtuelle Maschine, nicht aber auf das eigene System auswirken.

Die virtuelle Maschine könnte allerdings durch entsprechenden Code erkannt[18] werden. In diesem Fall kann das Programm sein Verhalten ändern und den Entpackvorgang abbrechen.

Mit diesem Verfahren wird die Sicherheit des eigenen Systems erhöht. Gleichzeitig wird aber riskiert, dass einige Entpackroutinen nicht mehr wie gewünscht arbeiten könnten.

- Die vorhandene Entpackroutine analysieren und eine eigene Entpackroutine schreiben.

  Um eine eigene Entpackroutine für einen spezifischen Laufzeitpacker schreiben zu können, muss zunächst dessen Entpackroutine analysiert werden. Die hierfür benötigte Zeit hängt stark von der Komplexität der Entpackroutine ab. Mit dieser Routine können dann alle mit dem spezifischen Laufzeitpacker gepackten Programme schnell und sicher[19] entpackt werden, da das gepackte Programm nicht ausgeführt wird.

  Wird die Entpackroutine vom Laufzeitpacker polymorph generiert – sie besitzt also bei jedem gepackten Programm ein anderes Aussehen und leicht veränderte Funktionalität – ist es sehr schwer[20], eine eigene statische Entpackroutine hierfür zu erstellen.

  Falls ein Packer geringfügig modifiziert wird, so dass die hiermit gepackten Dateien nicht mehr als solche erkannt werden oder die Entpackroutine sich geringfügig anders verhält, können die hiermit gepackten Dateien zunächst nicht mehr von der eigenen Routine entpackt werden.

  Dieses Verfahren ist zwar in der Anwendung schnell und sicher, erfordert aber bei der Erstellung einigen Zeitaufwand und ist nicht generisch. [Vnuk u. Návrat, 2006] beschreiben eine Methode, mit der der benötigte Zeitaufwand reduziert werden kann. Diese ändert aber nichts daran, dass für jede Laufzeitpackervariante eine neue eigene Entpackroutine erstellt werden muss. Das Verfahren ist somit nicht für unbekannte oder modifizierte Laufzeitpacker geeignet.

Die aufgeführten generischen Methoden sind unzuverlässig, unsicher und schwer in einen Malware-Scanner zu integrieren. Die sichere, gut in einen Malware-Scanner integrierbare Methode hingegen ist nicht generisch genug, so dass sie leicht umgangen werden kann. Es wird also eine Methode benötigt, die sowohl generisch als auch sicher ist.

---

[17] Vorausgesetzt, die virtuelle Maschine enthält keine Sicherheitslücken, die der Schadcode nutzen könnte um auf dem eigentlichen System ausgeführt zu werden.

[18] Siehe [Quist u. Smith, 2006], [Bayer u. a., 2006, Seite 3] und [Brulez, 2006, Folie 8].

[19] Vorausgesetzt die Entpackroutine ist fehlerfrei implementiert, so dass keine Sicherheitslücken durch präparierte Malware ausgenutzt werden können.

[20] [Eilam, 2005, Seite 330] formuliert hierfür ein Konzept.

Dazu bietet sich Emulation an. Hierbei werden Anweisungen nicht direkt ausgeführt, sondern nur von einem Programm interpretiert und ihre Auswirkungen simuliert. Das gepackte Schadprogramm kann somit keine Schäden anrichten, da es zu keinem Zeitpunkt regulär ausgeführt wird.

Emulation wird bereits eingesetzt, um polymorphe Viren zu entschlüsseln. Diese enthalten als Reaktion darauf teilweise Routinen, die eine Emulation verhinden sollen [Szor, 2001].

[Natvig, 2002] beschreibt in „Sandbox II: Internet“ eine Sandbox, also eine Umgebung, in der Malware keine schädliche Auswirkung auf das eigentliche System haben kann, die in Form eines Emulators realisiert wurde. Durch Ausführen von Programmen in der Sandbox soll erkannt werden, ob diese eine Schadroutine besitzen. Natvig weist darauf hin, dass gepackte Programme kein Problem für die Erkennung darstellen sollen. Diese Sandbox-Technologie wurde in den Malware-Scanner NVC5[21] integriert. Auf der zugehörigen Produkt-Webseite wird die Sandbox-Technologie erst im August 2003 erwähnt.

Dieser Emulator war laut einem Test[22] von Roman Kopecny im Januar 2003, sofern die Sandbox-Technik zu diesem Zeitpunkt bereits in NVC5 integriert war und der Test korrekt durchgeführt wurde, anscheinend noch nicht praxistauglich. Die Erkennung gepackter Schadprogramme durch Malware-Scanner, darunter auch NVC5, wurde anhand von 6 verschiedenen Schadprogrammen getestet. Diese wurden von allen getesteten Malware-Scannern erkannt. Zum Packen wurden 18 verschiedene Laufzeitpacker verwendet. Keine der gepackten Dateien wurde von NVC5 erkannt.

Zwei Jahre später wurden 13 von 18 gepackten Nimda.A-Dateien in einem Test von [Johansen, 2005] in der Sandbox korrekt entpackt und als Wurm identifiziert. Es ist unbekannt, ob der Wurm ausschließlich am Verhalten oder auch direkt nach abgeschlossenem Entpackvorgang anhand von Signaturen erkannt wurde.

Da es sich bei kommerzieller AV-Software um geschlossene Produkte handelt, ist es schwer möglich, an Details zu deren Funktionsweise zu gelangen [Royal u. a., 2006, Seite 3].

[21] Norman Virus Control 5 http://www.norman.com/Product/Home_Home_office/NVC/758/en/ 08.07.2007.

[22] http://www.rokop-security.de/main/article.php?sid=473 09.08.2004. Der Testbericht ist nicht mehr online verfügbar. Eine lokale Kopie ist jedoch vorhanden.

3 Anforderungen an eine Lösung

Um praxistauglich zu sein, muss eine Lösung für das Entpacken geschützter Schadprogramme eine Reihe von Anforderungen erfüllen. Denn das Liefern guter Entpack-Ergebnisse allein ist nicht mit Praxistauglichkeit gleichzusetzen, wenn andere Punkte den effizienten Einsatz in einem Malware-Scanner verhindern. Abgesehen von den regulären Qualitätsanforderungen für Software, müssen folgende lösungsspezifische Anforderungen erfüllt werden.

- Benutzerinteraktion:

  Das Entpacken einer Zieldatei muss automatisch erfolgen können. Es darf hierbei keine steuernde Benutzerinteraktion notwendig sein. Dies sichert die Praxistauglichkeit, da ein durchschnittlicher Endanwender nicht die erforderlichen Kenntnisse für die Steuerung des Entpackvorgangs besitzt. Weiterhin ist ein Verzicht auf Benutzerinteraktion eine Voraussetzung für den serverseitigen Einsatz.

- Sicherheit:

  Der Entpackvorgang muss kontrolliert erfolgen. Das Zielprogramm muss hierbei so isoliert sein, dass es keine Auswirkung auf das den Entpackvorgang durchführende System oder angeschlossene Netzwerke haben kann. Hierdurch wird insbesondere verhindert, dass eine in der Entpackroutine des Zielprogramms möglicherweise enthaltene Schadroutine eine negative Auswirkung haben kann.

- Generische Anwendbarkeit:

  Die Lösung muss so ausgelegt sein, dass nicht nur mit bekannten Laufzeitpackern gepackte Malware entpackt werden kann. Es muss vom Design her die Möglichkeit gegeben sein, auch mit modifizierten oder unbekannten Laufzeitpackern[1] gepackte Malware zu entpacken. Ansonsten würde die Lösung keinen Vorteil gegenüber bestehenden statischen Entpackroutinen bieten. Dennoch kann – in Hinblick auf den Implementierungsaufwand – nicht gefordert werden, dass jede beliebige Datei ohne vorherige Verbesserung der Lösung entpackt werden kann.

- Plattformunabhängigkeit:

  Die erstellte Lösung muss unter Windows 2000-kompatiblen Betriebssystemen, dem Hauptziel von Malware, sowie UNIX-kompatiblen Betriebssystemen lauffähig sein. Prozessortypen, die der weitverbreiteten x86-32- oder x86-64-Archtitektur entsprechen,

[1] Entsprechend den in Abschnitt 2.3.3 genannten Einschränkungen.

müssen unterstützt werden. Hierdurch wird eine Nutzung in Malware-Scannern sowohl im Desktop- als auch im Serverbereich ermöglicht, was eine wesentliche Voraussetzung für die Praxistauglichkeit ist.

Um diese Plattformunabhängigkeit und konsistente Ergebnisse zu ermöglichen, dürfen keine Änderungen in der Logik oder Funktionsweise vorgenommen werden, um die Lösung von einer auf eine andere Plattform zu portieren. Ergänzende plattformspezifische Optimierungen zur Geschwindigkeitssteigerung sind jedoch zulässig.

- Funktionssicherheit:

  Die erzielten Ergebnisse müssen auf jedem anderen Zielsystem reproduzierbar sein. Diese Anforderung leitet sich aus den Punkten Plattformunabhängigkeit und Sicherheit ab. Das Betriebssystem, der Prozessor oder sonstige Parameter des Zielsystems dürfen somit keinen Einfluss auf das Ergebnis haben. Abweichende Ergebnisse, die auf möglichen Konfigurationsunterschieden bei der konkreten Installation beruhen, sind zulässig, da sie reproduzierbar sind. Durch diese Anforderung werden inkonsistente Ergebnisse, die beispielsweise durch Timing-Probleme oder Hardware-Unterschiede entstehen könnten, verhindert.

- Geschwindigkeit:

  Die Arbeitsgeschwindigkeit eines Malware-Scanners hat einen starken Einfluss auf seine Praxistauglichkeit. Sie ist nach [Muttik, 2000, Seite 8] fast so wichtig wie die Eigenschaft, Malware zu erkennen. Hier ist es schwer, eine absolute Obergrenze für die Entpackzeit pro Datei festzulegen, zumal sich die Dateigrößen und Entpackroutinen stark unterscheiden können. Falls ein Schlüssel für die Entschlüsselung des ursprünglichen Programms von der Entpackroutine durch Ausprobieren gesucht wird, kann der Entpackvorgang sehr lange dauern [Szor, 2000, Seite 57].

  Es sollte höchstens ein Verhältnis von 100 zu 1 für die Entpackzeit, zuzüglich einer angemessenen Fixzeit[2], durch die Lösung gegenüber der Entpackzeit bei nativer Ausführung des gepackten Programms benötigt werden. Für ein laufzeitgepacktes Programm, welches sich innerhalb einer vom Nutzer kaum wahrnehmbaren Zehntelsekunde entpackt, dürften also höchstens 10 Sekunden Zeit – zuzüglich Fixzeit – zum Entpacken benötigt werden. Dies hat den Hintergrund, dass es mit Wissen über die konkrete Implementierung der Lösung möglich sein kann, eine Entpackroutine zu konstruieren, die schnell nativ ausgeführt wird, aber sehr viel Zeit beim Entpackvorgang durch die Lösung benötigt.

[2] Hierdurch soll verhindert werden, dass ein Verhältnis von 100 zu 1 bei sehr kurzen oder schleifenlosen Programmen ggf. nicht erreicht wird.

4 Konzeption einer eigenen Lösung

Nicht jede ausführbare Datei ist gepackt. Daher sollte zunächst festgestellt werden, ob eine Zieldatei gepackt ist. Wenn sie es nicht ist, kann die für den Entpackversuch notwendige Zeit eingespart werden. Ist die Zieldatei gepackt und keine schnelle statische Entpackroutine für sie vorhanden, muss sie nun generisch entpackt werden.

[Muttik, 2000, Seite 60], [Szor, 2000, Seite 49] und [Stepan, 2005, Seite 40] nennen die X-Ray-Methode[1] und Emulation[2] als Möglichkeit, polymorph verschlüsselte Viren zu entschlüsseln.

Die X-Ray-Methode wird aber nur in Bezug auf Entschlüsselung, nicht auf Dekomprimierung genannt. Zudem ist diese Methode nur für einfache Verschlüsselungen geeignet [siehe Stepan, 2005, Seite 40].

Der Umstand, dass sich gepackte Programme bei der Ausführung selbst entpacken, kann ausgenutzt werden, um das entpackte Programm zu erhalten. Da die direkte Ausführung zu viele Nachteile hat, die den Vorteil der hohen Geschwindigkeit überwiegen, bleibt nur die simulierte bzw. emulierte Ausführung. Diese kann zudem auch deutlich besser überwacht werden.

Während der simulierten Ausführung muss erkannt werden, wann die erste Anweisung des entpackten Programms ausgeführt wird, es also vollständig entpackt ist. In diesem Fall kann die simulierte Ausführung angehalten werden. Das entpackte Programm liegt nun als Speicherabbild vor.

Für die Überprüfung durch einen Malware-Scanner wird aus dem Speicherabbild die Dateiform des Originalprogramms so gut wie möglich rekonstruiert. Die rekonstruierte Datei ist ausführbar und wird sich, sofern sie ausgeführt wird, exakt wie das ungepackte Originalprogramm verhalten. Sie wird also keinen[3] Code dynamisch erzeugen und ausführen. Der Malware-Scanner kann nun also eine ausführbare Datei, die fast dem Originalprogramm entspricht, überprüfen.

---

[1] Entschlüsselung durch gezieltes Ausprobieren oder auch automatische Kryptoanalyse.

[2] Simulierte Ausführung von Anweisungen.

[3] Vorausgesetzt, das Originalprogramm enthält keine Routinen zur dynamischen Codeerzeugung und der Entpackversuch war erfolgreich.

4.1 Erkennung gepackter Dateien

Ein Programm, welches sich zur Laufzeit entpackt, muss neuen Code generieren. Wird zur Laufzeit eine Codesequenz ausgeführt, die vor dem Programmstart noch nicht vorhanden war, so handelt es sich wahrscheinlich[4] um ein gepacktes Programm.

Die Frage, ob ein Programm zur Laufzeit neuen Code erzeugen wird oder nicht, ist äquivalent zum Halteproblem und somit unentscheidbar [Royal u. a., 2006, Seite 4, sowie Anhang A].

Es ist also aufgrund des Halteproblems nicht generell möglich vorherzusagen, was ein Programm machen wird. Jedoch ist normalerweise eine Abschätzung möglich [Schultz u. a., 2001, Seite 3]. Für Ansätze zur Erkennung gepackter Dateien wird auf folgende Arbeiten verwiesen: [Christodorescu u. a., 2005b], [Christodorescu u. a., 2005a], [Szor, 2005], [Brulez, 2006] und [Santamarta, 2006].

Da die Entscheidung, ob ein unbekanntes Programm gepackt ist oder nicht, nur auf Heuristiken beruht, sind hier auch Fehler möglich. Besonders mit genauer Kenntnis der verwendeten Heuristiken wäre es möglich, ein selbstentpackendes Programm zu konstruieren, welches nicht als solches erkannt wird. Dies würde bedeuten, dass kein Entpackversuch unternommen wird – unabhängig davon, ob das Programm entpackt werden könnte. Möglicherweise enthaltene Malware wird also nicht erkannt werden.

Die einzige Schutzmöglichkeit hiergegen besteht darin, einen Entpackversuch für jedes zu prüfende Programm zu starten.

4.2 Realisierung des Entpackvorgangs

Das zu entpackende Programm darf aus Sicherheitsgründen nicht direkt ausgeführt werden. Zudem könnte es auch nicht immer ausgeführt werden, da auf dem Zielsystem kein geeignetes Betriebssystem zur Verfügung zu stehen braucht.

Ziel ist es also, für die Entpackroutine eine Umgebung zu simulieren, die sich in allen relevanten Aspekten exakt wie die erwartete Umgebung verhält. Der Entpackvorgang muss überwacht werden können, so dass er abgebrochen werden kann, sobald das Programm vollständig entpackt wurde.

Da das Zielbetriebssystem nicht dem benötigten Betriebssystem entsprechen muss, wird untersucht, wie Programme unter fremden Betriebssystemen ausgeführt werden können. [Lawton, 1999] und [Surauer, 2002, Seite 3] geben für die Ausführung von Programmen unter anderen Betriebssystemen drei Möglichkeiten an: API-Emulation, Virtualisierung und vollständige Emulation. Diese Techniken werden in den folgenden Abschnitten erläutert und auf ihre Anwendbarkeit im Rahmen der Zielsetzung untersucht. Hierbei wird ersichtlich,

[4] Es wäre auch möglich, dass es sich um ein selbstoptimierendes Programm handelt, welches beispielsweise mittels Just-In-Time-Compilation Code erzeugt.

dass die gesetzten Anforderungen ausschließlich durch Verwendung vollständiger Emulation erfüllt werden können.

4.2.1 API-Emulation

Bei der API-Emulation werden die entsprechenden Funktionen eines Betriebssystems nachgebildet. Sie werden einem Programm, welches unter einem fremden Betriebssystem ausgeführt werden soll, zur Verfügung gestellt. Dieser Ansatz wird vom Wine-Projekt[5] verwendet, um unvorbereitete Windows-Anwendungen unter Linux auszuführen.

Eine komplette Nachbildung ist allerdings schwierig[6], da API-Funktionen teilweise sehr umfangreiche oder auch schlechte Dokumentationen besitzen, sofern sie überhaupt dokumentiert sind.

Durch die API-Emulation agiert das entsprechende Programm nicht direkt mit dem Betriebssystem, sondern mit einer Zwischenschicht – den emulierten API-Funktionen. Das Programm wird dabei direkt auf dem Prozessor ausgeführt, weswegen die resultierende Ausführungsgeschwindigkeit bei diesem Verfahren sehr hoch ist.

Bei einer korrekten Implementierung kann das Betriebssystem hierüber nicht negativ beeinflusst werden. Es besteht allerdings, wie bereits kurz in Abschnitt 2.2.1 erwähnt, die Möglichkeit, auch ohne Verwendung von API-Funktionen mit dem Betriebssystem zu interagieren [siehe Eilam, 2005, Seite 91ff]. Somit bietet dieses Verfahren nur unzureichende Sicherheit, zumal direkte Systemaufrufe nach [Kaspersky u. a., 2003] auch in einigen Fällen als Anti-Debugger-Maßnahme eingesetzt werden können.

Da das Programm direkt auf dem Prozessor ausgeführt wird, ist keine Kontrolle über den Ablauf gegeben. Auch wird die Anforderung der Funktionssicherheit nicht erfüllt, da das Ergebnis vom verwendeten Prozessor und der aktuellen Prozessorlast abhängen kann. Dieses Verfahren ist somit nicht für eine Realisierung des Entpackvorgangs geeignet.

4.2.2 Virtualisierung

Durch Virtualisierung des Prozessors können Programme – oder sogar ganze Betriebssysteme – auf einem Prozessor ausgeführt werden, ohne ein darauf bereits laufendes Betriebssystem zu beeinflussen. In der Regel werden hierbei auch Hardwarekomponenten virtualisiert. Da Malware jedoch nicht auf diese zugreifen darf, reicht es aus, wenn diese nur emuliert werden.

Die Haupteigenschaft der Virtualisierung ist es, einen Großteil der Anweisungen direkt auf dem Prozessor ausführen zu können [Bayer, 2005, Seite 7]. Anweisungen, die nicht direkt

[5] „Wine Is Not an Emulator“ http://www.winehq.org 08.07.2007.
[6] Siehe [Nerche, 2000, Seite 8], [Eilam, 2005, Seite 142] und [Szor, 2005, Kapitel 3.6.5.2].

ausgeführt werden können oder sollen, müssen emuliert werden. Die Anzahl an Anweisungen, die direkt auf dem Prozessor ausgeführt werden können, bestimmt maßgeblich die erreichte Geschwindigkeit [Robin u. Irvine, 2000, Seite 3].

Dies bedeutet aber auch, dass Virtualisierung nicht architekturübergreifend einsetzbar ist. Ein der x86-32-Architektur entsprechendes System kann also auch nur eine x86-32-Architektur virtuell zur Verfügung stellen [Ohl, 2002, Seite 4]. Ansonsten müssten zu viele Anweisungen emuliert werden, wodurch der durch die Virtualisierung erlangte Geschwindigkeitsvorteil hinfällig wird. Die Anforderung der Plattformunabhängigkeit wird somit nicht erfüllt.

Abgesehen davon sind Prozessoren der x86-32- und x86-64-Architektur „nicht oder nur partiell für die Virtualisierung entworfen worden“ [Ohl, 2002, Seite 5].

> Die größten Schwierigkeiten bereiten folgende Eigenheiten der IA32[7]: Systemregister sind auch für nichtprivilegierte Programme lesbar, das komplizierte Befehlsformat erlaubt überlappende und geschachtelte Befehle, und ein Kontextwechsel zwischen Kern und Nutzermodus dauert relativ lange.
> [Nerche, 2000, Seite 51]

Durch ein Auslesen der Systemregister könnte ein in einer virtualisierten Umgebung ablaufendes Programm einfach feststellen, dass es sich in einer virtualisierten Umgebung befindet. Entsprechende Zugriffe müssen daher abgefangen und emuliert werden. Dies ist aber nicht einfach und kostet zusätzliche Zeit [Ohl, 2002, Seite 13].

Nach [Nerche, 2000, Seite 23] verlangsamt Code, der viele Selbstmodifikationen oder Selbsttests durchführt, eine virtualisierte Ausführung zusätzlich aufgrund der bei der Virtualisierung notwendigen speziellen Behandlung von Lese-/Schreib- und Ausführungszugriffen. Hierdurch wird auch eine Überwachung des Entpackvorgangs starke zeitliche Nachteile mit sich bringen.

Eine virtualisierte Umgebung kann anhand einiger weiterer Eigenheiten möglicherweise generisch erkannt[8] werden. Die Anforderung der Plattformunabhängigkeit und der Funktionssicherheit kann mit diesem Ansatz also nicht erfüllt werden.

Die Virtualisierung ist dennoch ein interessanter und performanter Ansatz für den Umgang mit gepackten Dateien, deren Entpackroutine keinen Code zur Erkennung der virtualisierten Umgebung enthält. Dies gilt insbesondere dann, wenn der OEP manuell bestimmt oder abgeschätzt werden kann, so dass der Entpackvorgang nicht intensiv überwacht zu werden braucht.

Für weitere Informationen zur Virtualisierung sowie den damit verbundenen Problemen und Möglichkeiten wird auf folgende Arbeiten verwiesen: [Popek, 1974], [Lawton, 1999], [Nerche, 2000], [Robin u. Irvine, 2000], [Ohl, 2002], [Bayer, 2005] und [Ferrie, 2006].

---

[7] x86-32 ist eine generellere Bezeichnung für IA32.

[8] Siehe [Ferrie, 2006], [Quist u. Smith, 2006], [Brulez, 2006, Slide 8] und [Nerche, 2000, Seite 50].

4.2.3 Vollständige Emulation

Bei der vollständigen Emulation wird das Zielprogramm nicht direkt auf der Hardware ausgeführt, sondern alle Anweisungen und Interaktionen mit der Umgebung werden durch ein Programm emuliert.

> That all computers can simulate each other is an immediate consequence of the theoretical work of Alan Turing and Alonzo Church.
> [Magnusson u. a., 2002, Seite 50]

Das eingesetzte Verfahren kann also komplett plattformunabhängig und zudem auch sicher sein, da keine Anweisung des Zielprogramms direkt ausgeführt wird.

Abgesehen von einigen generellen Aspekten, werden in den folgenden Abschnitten noch potentiell auftretende und zu berücksichtigende Probleme bei der Emulation erläutert. Auch wird eine Abschätzung der zu erwartenden Ausführungsgeschwindigkeit vorgenommen. Anschließend werden Möglichkeiten zur Steigerung der Ausführungsgeschwindigkeit erläutert.

Die von einem Zielprogramm benötigte Funktionalität eines Prozessors muss in Form eines Emulators implementiert werden. Abgesehen vom Prozessor benötigen Programme in der Regel auch eine Betriebssystemumgebung, mit der sie interagieren können. Die hiervon benötigten Komponenten müssen ebenfalls emuliert werden. Da die Zielprogramme nicht im Kernel-Mode ausgeführt werden – somit nur beschränkte Zugriffsrechte haben –, ist die Implementierung eines ausreichenden Emulators deutlich einfacher als die Implementierung eines Emulators für Kernel-Mode-Code, wie beispielsweise Betriebssysteme [Magnusson, 1993, Seite 4]. Dadurch, dass ein Programm zu vielen Prozessortypen kompatibel sein sollte, durch die Emulation aber nur ein spezifischer Prozessortyp in dem vom Betriebssystem verwendeten Ausführungsmodus[9] nachgebildet zu werden braucht, wird der Implementierungsaufwand weiter reduziert.

Dadurch, dass das gesamte System durch Software emuliert wird, kann bei umfassender und ausreichend genauer Implementierung ein Punkt erreicht werden, an dem die emulierte Umgebung nicht mehr zuverlässig durch darin ablaufende Programme erkannt werden kann [Ferrie, 2006, Seite 9]. Eine Umgebung, die von Malware nicht von einer realen Umgebung unterschieden werden kann, ist notwendig, damit die Malware ihr Verhalten nicht ändern und so das Ergebnis verfälschen kann [Bayer u. a., 2006, Seite 3]. Der Punkt der Funktionssicherheit kann somit auch erfüllt werden. Zudem hat der Emulator zu jedem Zeitpunkt die volle Kontrolle über den Ablauf des Zielprogramms [Bayer u. a., 2006, Seite 4]. Dies ist eine wesentliche Voraussetzung für die im weiteren Verlauf dieser Arbeit beschriebene Überwachung des Entpackvorgangs.

Durch die vollständige Emulation wird die Ausführungsgeschwindigkeit des Zielprogramms stark reduziert [Robin u. Irvine, 2000, Seite 3], es gibt jedoch Möglichkeiten, diese

[9] Moderne Betriebssysteme versetzen den Prozessor in den „Protected Mode“. Die auch verfügbaren Modi „Real-Address Mode“ und „Virtual-8086 Mode“ werden nicht verwendet. Siehe hierzu auch [Intel, 2003].

Geschwindigkeitsreduktion weniger stark ausfallen zu lassen [Lawton, 1999]. Dennoch kann eine für einen spezifischen Laufzeitpacker erstellte statische Entpackroutine in jedem Fall schneller als der Emulator arbeiten. Selbst wenn durch eine dynamische Optimierung Code erhalten werden könnte, der genauso schnell wie der Code der statischen Entpackroutine ausgeführt werden kann, so ist der Emulator trotzdem noch langsamer, da eine dynamische Optimierung Zeit kostet. Eine Datei sollte laut [Natvig, 2002, Seite 14] wenn möglich mit einer statischen Entpackroutine anstelle des Emulators entpackt werden.

Der Ansatz der vollständigen Emulation kann also allen Anforderungen genügen. Daher wird dieser im Folgenden genauer betrachtet.

Die Idee, Malware in einer emulierten Umgebung auszuführen, ist nicht neu. So beschreibt bereits [Veldman, 1993] die Nutzung von Emulation für die Erkennung von Viren. [Kephart u. a., 1997] schlagen vor, das Verhalten von Viren in einem Emulator zu analysieren um ggf. automatisch eine Desinfektionsroutine erstellen zu können. Als Reaktion auf den Einsatz von Emulation durch Malware-Scanner werden Gegenmaßnahmen in Viren integriert [Szor, 2000]. Während Emulation meistens in Bezug auf Viren genannt wird, nennt [Natvig, 2002, Seite 13] diese in Bezug auf laufzeitgepackte Programme – allerdings um das Verhalten von gepackter Malware zu analysieren. Bevor das Verhalten der Malware analysiert werden kann, muss die Entpackroutine emuliert werden, dies geschieht beiläufig und wird nicht gesondert behandelt. [Graf, 2005] präsentiert eine Technik, die speziell darauf ausgelegt ist, laufzeitgepackte Programme durch Emulation zu entpacken.

4.2.3.1 Anweisungsset

Die auf einem Intel-Prozessor verfügbaren Anweisungen sind durch [Intel, 2003] dokumentiert. In der Regel werden nur auch auf sehr alten Prozessoren verfügbare Anweisungen von Entpackroutinen benötigt. Nach [Szor, 2000, Seite 55] verwenden einige Viren auch neuere Fließkomma- und MMX-Anweisungen – möglicherweise um einen Emulator aufzuhalten, der entsprechende Anweisungen nicht implementiert. Dies könnte auch in Entpackroutinen eingesetzt werden. Das Anweisungsset eines gebräuchlichen Prozessors muss also vollständig implementiert werden, um eine maximale Kompatibilität erreichen zu können.

Ein Problem hierbei ist die unterschiedliche Ausführung von Anweisungen durch verschiedene Prozessormodelle. So unterscheidet sich laut [Ferrie, 2006, Seite 3] das Ergebnis einer beispielhaft genannten Anweisung zwischen den verschiedenen Prozessortypen der Pentium-Serie. In [Intel, 2003] sind diverse Auswirkungen von Anweisungen gelegentlich mit „undefiniert“ beschrieben. Dies erlaubt also unterschiedliches Verhalten bei verschiedenen Prozessoren. Da ein spezifischer Prozessortyp konsistent emuliert werden soll, muss entsprechendes Verhalten ggf. anhand von Tests bestimmt werden. Eine konsistente Emulation ist wichtig, da Malware sonst die Emulation anhand der angetroffenen Abweichungen erkennen könnte [Szor, 2000, Seite 57].

Auch eine vollständige Implementierung anhand von [Intel, 2003] inklusive Berücksichtigung undefinierter Auswirkungen ist in einigen Fällen nicht ausreichend. Nach [Szor, 2000, Seite 57] verwendet ein Virus die undokumentierte „SALC“-Anweisung, um eine Emulation zu verhindern. Somit müssen auch ggf. vorhandene, aber undokumentierte Anweisungen korrekt implementiert werden.

4.2.3.2 Auftretende Fehler

Bei der Ausführung eines Programms können verschiedene Fehler[10], sogenannte „Exceptions“, auftreten. Tritt ein Fehler auf, wird das Betriebssystem vom Prozessor benachrichtigt und führt dann ggf. eine Fehlerbehandlungsroutine – einen „Exception-Handler“ – des Programms aus [Eilam, 2005, Seite 105ff].

Die Erkennung von Exceptions in einem emulierten Programm kostet viel Zeit [Hohensee u. a., 1996, Folie 27]. Bei jeder emulierten Anweisung müssten sämtliche Fehlerbedingungen geprüft werden. Dennoch müssen Exceptions unterstützt werden, da geplante Exceptions sowohl von Laufzeitpackern gegen Debugger eingesetzt als auch nach [Natvig, 2002, Seite 13] von Viren gegen Emulatoren verwendet werden.

Auftretende Exceptions können nur von einem Programm genutzt werden, wenn zuvor ein Exception-Handler beim Betriebssystem registriert wurde – falls keiner registriert ist, wird das Programm vom Betriebssystem im Fehlerfall beendet. Dies lässt sich möglicherweise zur Optimierung der Geschwindigkeit einsetzen. Erst wenn ein Exception-Handler registriert ist, erwartet das Programm auftretende Exceptions und kann darauf reagieren. Vorher auftretende Exceptions würden zu einer Programmbeendigung führen, sind also in der Regel nicht geplant. Daher reicht es möglicherweise aus, nur dann bei jeder Anweisung auf entsprechende Fehlerbedingungen zu prüfen, wenn auch ein Exception-Handler registriert ist.

4.2.3.3 Zu erwartende Geschwindigkeit

Emulatoren werden nicht nur im Malware-Bereich verwendet. Anhand von Testergebnissen anderer Emulatoren soll die zu erwartende Geschwindigkeit abgeschätzt werden.

Nach [Tijms, 2000] gibt es 5 verschiedene Genauigkeitsstufen für die Emulation eines Prozessors:

1. Datenpfadgenauigkeit
2. Taktgenauigkeit
3. Genauigkeit auf Anweisungsebene

[10] Beispielsweise eine Division durch Null, ein Zugriff auf eine nicht vorhandene Speicherseite oder die Ausführung einer zu langen Anweisung.

4. Genauigkeit auf Anweisungsblockebene
5. Semantische Genauigkeit

Je niedriger die Nummer einer Genauigkeitsstufe ist, desto genauer ist sie, aber desto langsamer ist auch die resultierende Ausführungsgeschwindigkeit.

Nach [Magnusson, 1993, Seite 3] ist eine Emulation mit Genauigkeit auf Anweisungsebene – hierbei hat jede Anweisung für das emulierte Programm die gleichen Auswirkungen wie auf einem realen Prozessor – gut geeignet, um Programmverhalten zu analysieren. Mit Genauigkeit auf Anweisungsebene arbeitende, auch „interpretierende Emulatoren“ genannte Programme bieten aber nach [Shaffer, 2004, Seite 10] nur eine sehr langsame Ausführungsgeschwindigkeit im Vergleich zu nativer Ausführung.

Daher wird die Größenordnung der relativen Ausführungsgeschwindigkeit interpretierender Emulatoren anhand vorhandener Arbeiten grob abgeschätzt. Da die Geschwindigkeit des Testsystems in einigen Fällen nicht angegeben wird, kann die relative Geschwindigkeit in diesen Fällen nur sehr ungenau ermittelt werden.

- Test u. a. eines für x86-32-Prozessoren geschriebenen Blowfish-Verschlüsselungs-algorithmus durch einen Emulator auf einer PowerPC-Plattform: Bei einer Datengröße von 1 MB wurde für die emulierte Ausführung ungefähr 95-mal mehr Zeit als für die native Ausführung benötigt [Shaffer, 2004, Appendix A].
- Emulation diverser Algorithmen – unter anderem QuickSort – durch SimX86 auf einem Pentium I - 133 MHz. Hier wurde nach [Shealy u. a., 1997, Seite 165] ein Geschwindigkeitsverhältnis von ungefähr 200 zu 1 erreicht.
- Emulation der Entpackroutine eines gepackten Programms: Es wurden ungefähr eine Million Anweisungen pro Sekunde auf einem Pentium III - 700 MHz emuliert [Natvig, 2002, Seite 14]. Dieser Prozessor führt ungefähr[11] 600 Millionen Ganzzahl-Operationen pro Sekunde aus. Daher wird die emulierte Ausführung in diesem Fall ungefähr 600-mal langsamer gegenüber der nativen Ausführung sein.
- Emulation kompletter Plattformen auf anderen Systemen durch [Magnusson u. a., 2002]: Hier wurden im x86-32-Bereich zwischen 2 und 6 – maximal 9 – Millionen Instruktionen pro Sekunde auf einen Pentium III - 933 MHz erreicht. Ausgehend davon, dass dieser ungefähr 900 Millionen Ganzzahl-Operationen pro Sekunde ausführt, wird hier höchstens ein Verhältnis von 100 zu 1 erreicht.

Dies entspricht der Angabe von [Stepan, 2005, Seite 41], wonach eine Emulation der Anweisungen mindestens 100-mal länger dauert als eine direkte Ausführung. Die erreichte Geschwindigkeit entspricht nicht der in den Anforderungen festgelegten Obergrenze von 100 zu 1. Daher muss diese erhöht werden.

[11] Siehe http://www.cpu-world.com/CPUs/Pentium-III/TYPE-Coppermine%20Socket%20370.html 16.07.2007.

4.2.3.4 Dynamische Kompilierung

Die Emulationsgeschwindigkeit kann durch Verwendung einer Zwischenrepräsentation oder einer anderen Genauigkeitsstufe gesteigert werden.

Nach [Shealy u. a., 1997, Seite 165] kann die Geschwindigkeit eines interpretierenden Emulators erhöht werden, in dem die x86-Anweisungen zunächst in eine Zwischenrepräsentation überführt werden, die mit weniger Zeitaufwand interpretiert werden kann. Hierbei wird die Emulation nicht ungenauer, aber die Geschwindigkeit wird auch nur leicht erhöht. Allerdings könnte der erzeugte Zwischencode möglicherweise optimiert werden [Stepan, 2005, Seite 43].

Eine höhere Geschwindigkeit kann nach [Tijms, 2000] auch durch die Verwendung von Genauigkeit auf Anweisungsblockebene erreicht werden. Als Anweisungsblock wird eine Sequenz von einer oder mehreren Anweisungen betrachtet, die mit einer Sprunganweisung endet [Probst, 2001, Seite 2] oder den Prozessorzustand auf eine nicht vorhersagbare Weise verändert [Bellard, 2005].

Die Einteilung von Anweisungen in Blöcke und die Übersetzung der Anweisungsblöcke muss zur Laufzeit erfolgen. Nur so kann der Ausführungspfad – und somit die zu übersetzenden Anweisungen – genau bestimmt werden. Dies wäre bei einer statischen Übersetzung schwer möglich [Kruegel u. a., 2004, Seite 2].

Eine Übersetzung der Anweisungsblöcke in nativen Maschinencode kann die erzielte Geschwindigkeit – auch aufgrund von hierbei durchgeführten Optimierungen – deutlich erhöhen. Siehe [Robin u. Irvine, 2000, Seite 11], [Stepan, 2005, Seite 40ff] und [Shaffer, 2004, Seite 11]. Da die für die Übersetzung einer Anweisung benötigte Zeit in etwa der für die Emulation der Anweisung benötigten Zeit entspricht [Stepan, 2005, Seite 41], kann hier insbesondere bei den in Entpackroutinen häufig vorkommenden Schleifen eine hohe Geschwindigkeitssteigerung erreicht werden.

Viele Anweisungen setzen sogenannte „Flags“; anhand dieser kann beispielsweise abgelesen werden, ob das Ergebnis einer Operation 0 oder größer als das Zielregister war oder sich das Vorzeichen-Bit verändert hat. Oft werden diese Informationen nicht von der Folgeinstruktion verwendet und hiervon überschrieben, ihre Berechnung kann also durch Verwendung optimierter Anweisungsblöcke eingespart werden [Probst, 2001, Seite 7]. Dies könnte auch für Registerwerte oder sogar komplette Anweisungen durchgeführt werden, siehe [Cifuentes, 1994] und [Christodorescu u. a., 2005b, Seite 8]. Ggf. kann es auch möglich sein, Flags erst dann zu setzen, wenn sie auch benötigt werden [Magnusson, 1993]. Hierdurch kann zusätzlich Rechenzeit eingespart werden.

Einzelne Anweisungsblöcke können verkettet und zu noch größeren Anweisungsblöcken zusammengesetzt werden. Hierdurch kann eine effizientere Optimierung, speziell im Hinblick auf die Entfernung überflüssiger Berechnungen, durchgeführt werden, siehe [Witchel u. Rosenblum, 1996, Seite 5] und [Probst, 2001, Seite 8].

Zu erwartende Geschwindigkeit: Der bereits zuvor erwähnte Blowfish-Algorithmus konnte im Test von [Shaffer, 2004, Appendix A] durch dynamische Kompilierung mit einer relativen Geschwindigkeit von 11 zu 1 anstelle von 95 zu 1 ausgeführt werden. Bei sehr häufig verwendeten Codeblöcken war sogar eine Geschwindigkeitssteigerung um den Faktor 100 gegenüber dem interpretierenden Emulator möglich [Shaffer, 2004, Seite 38].

Bei der Emulation verschlüsselter Viren konnte in Tests von [Stepan, 2005, Appendix B] eine Erhöhung der Geschwindigkeit um den Faktor 4 bis 8 gegenüber interpretierter Ausführung erreicht werden. Nach [Graf, 2005] sind auch teilweise Steigerungen um den Faktor 12 möglich. Die resultierende Geschwindkeit kann somit den Anforderungen entsprechen.

Probleme durch selbstmodifizierenden Code: Die Verwendung von dynamisch kompiliertem Code ist im Zusammenspiel mit selbstmodifizierendem Code laut [Ferrie, 2006, Seite 7ff] problematisch, da hierbei möglicherweise nicht mehr das exakte Prozessorverhalten nachgebildet wird.

In der von [Witchel u. Rosenblum, 1996] beschriebenen Implementierung werden alle übersetzten Anweisungsblöcke gelöscht, wenn ein Schreibvorgang auf eine bereits übersetzte Codeseite durchgeführt wird. Bei der weiteren Emulation müssten also alle Anweisungsblöcke erneut erstellt werden. Es wird auch darauf hingewiesen, dass dies verbessert werden sollte, wenn selbstmodifizierender Code häufiger vorkommt.

Da selbstmodifizierender Code bei Laufzeitpackern häufig anzutreffen ist, muss hier ein weniger zeitintensiver Weg gefunden werden. [Stepan, 2005, Seite 44] beschreibt hierzu Möglichkeiten, Blöcke nur dann neu zu übersetzen, wenn es notwendig wird. Hierbei wird auch eine Lösung für das Problem der selbstüberschreibenden Anweisungsblöcke sowie für die Verarbeitung von in Anweisungsblöcken aufgetretenen Exceptions erläutert.

Angriffe auf die dynamische Kompilierung: Mit Kenntnis der genauen Funktionsweise der Blockbildung und dynamischen Kompilierung können gezielt Angriffe auf diese durchgeführt werden, um die resultierende Geschwindigkeitssteigerung zu vermindern.

Wenn ein Anweisungsblock durch eine Sprunganweisung beendet wird, können hinter jeder Anweisung eingefügte Sprunganweisungen zu einer Erzeugung sehr kleiner Anweisungsblöcke führen. Hierdurch und durch die somit kaum mögliche Optimierung wird die Emulationsgeschwindigkeit verringert.

Eine Schleife könnte bei jedem Durchlauf ein – möglicherweise nicht relevantes – Byte innerhalb des Schleifencodes verändern. Somit wäre bei jedem Durchlauf eine neue Übersetzung notwendig, wobei dies möglicherweise zu einer Geschwindigkeit unterhalb der eines interpretierenden Emulators führen könnte.

Für diese und weitere Angriffe müssen ggf. Lösungen gefunden werden, um die aus dynamischer Kompilierung resultierende hohe Emulationsgeschwindigkeit beibehalten zu können.

4.3 Umgebungsparameter

Einige Parameter der emulierten Umgebung sind durch das emulierte Programm direkt abfragbar. Wie in Abschnitt 2.3.3 erklärt, sollte das zu emulierende Programm nicht umgebungsabhängig sein. Falls einige Merkmale der emulierten Umgebung jedoch typisch für diese sind und so bei realen Systemen nicht oder nicht häufig vorkommen, kann ein automatischer Entpackvorgang nach Erkennung dieser Merkmale durch das Programm verhindert werden. Auch kann das emulierte Programm – unabhängig von einer Erkennung für emulierte Umgebungen – nur mit speziellen Umgebungsparametern lauffähig sein.

Eine Verwendung statischer Umgebungsparameter für die emulierte Umgebung könnte also möglicherweise von einem Angreifer ausgenutzt werden, um diese zu erkennen und basierend auf einer Erkennung das Programmverhalten zu ändern. Ein sofortiger Programmabbruch würde in diesem Fall dazu führen, dass das Programm nicht entpackt werden könnte.

Um eine generische Erkennung anhand der Umgebungsparamter zu vermeiden, sollten diese – soweit möglich – für jede Instanz der Lösung auf einen zufälligen, aus einem vorgegebenen Bereich ausgewählten Wert gesetzt werden. Dies widerspricht nicht der in Kapitel 3 aufgestellten Anforderung der Funktionssicherheit, da die Ergebnisse unter Verwendung einer identischen Konfiguration reproduzierbar sind.

Der Prozessortyp kann von einem Programm über die „CPUID“-Anweisung[12] abgefragt werden. Wird hierbei bei allen Instanzen der Emulationsumgebung der gleiche Wert zurückgeliefert, so könnte sie hieran erkannt werden. Ist der eingestellte Prozessortyp ein weitverbreitetes Modell, so würde ein Ausschluss durch ein Programm dazu führen, dass es sowohl im Emulator als auch auf vielen mit dem Prozessor ausgerüsteten Systemen nicht mehr lauffähig wäre. Allerdings wäre es auf allen verbleibenden Systemen mit anderem Prozessortyp lauffähig und würde dort nicht in der eingesetzten Emulationsumgebung entpackt werden können. Daher sollte hier ein möglichst breites Spektrum an Prozessortypen zur Auswahl stehen.

Nach [Stepan, 2005, Seite 45] ist es schwer, die Umgebungsparameter so einzustellen, dass auch Programme mit speziellen Anforderungen lauffähig sind, zumal die Anforderungen der Programme unvereinbar sein können. Spezielle Anforderungen einer Entpackroutine würden allerdings dazu führen, dass sie nur auf wenigen Systemen lauffähig wäre.

[12] CPU IDentification.

Bei der Emulation eines Programms wird nur genau ein Ausführungspfad verfolgt. Daher kann auch ein unzureichender Umgebungsparameter zu einem frühzeitigen Abbruch des Programms führen. [Bayer u. a., 2006, Seite 11] schlagen daher vor, den aktuellen Emulatorzustand beim Antreffen eines bedingten Sprungs zu duplizieren und beide resultierende Ausführungspfade zu verfolgen.

Dies für jeden bedingten Sprung durchzuführen würde in einer zu hohen Anzahl von abzuarbeitenden Emulatorzuständen resultieren. Daher sollte dies nur – und auch nur in begrenztem Umfang – durchgeführt werden, wenn ein zuvor abgefragter Umgebungswert für den bedingten Sprung verwendet wird. Es kann allerdings schwer sein, dies genau zu bestimmen. Dieser Punkt könnte mit zunehmendem Einsatz von aktivem[13] Anti-Emulationscode stark an Relevanz gewinnen und sollte weiter erforscht werden. Dies ist jedoch nicht Zielsetzung dieser Arbeit.

4.3.1 Systemzeit

Ein Programm kann Datum und Uhrzeit des emulierten Systems abfragen. Hier kann aufgrund der Erkennungsmöglichkeit kein statischer Startwert verwendet werden. Auch ein Wert, der weit hinter oder vor dem aktuellen Datum liegt, könnte Probleme verursachen. Beispielsweise könnte die Entpackroutine so konstruiert sein, dass sie nicht zu einem – anhand der Systemzeit ermittelten – Zeitpunkt ausgeführt werden kann, der vor dem Zeitpunkt des Packvorgangs liegt. Auch wäre eine Beschränkung auf einen Zeitraum von beispielsweise 10 Tagen ab Packvorgang denkbar. Dies würde ausreichen, um die gepackt enthaltene Malware auszuführen und bei Bedarf Malware mit einem weiter verschobenen Zeitfenster nachzuladen.

Daher bleibt nur, die Systemzeit der emulierten Umgebung auf die aktuelle Zeit des realen Systems zu setzen. Dies gilt jedoch nur für den Startwert. Da bei einer emulierten Ausführung weniger Anweisungen pro Sekunde als bei einer realen Ausführung abgearbeitet werden, könnte die emulierte Umgebung ansonsten an der relativ gesehen zu schnell fortschreitenden Realzeit erkannt werden. Abgesehen davon würde dieses Verhalten nicht der in Kapitel 3 aufgestellten Anforderung der Funktionssicherheit genügen.

Daher müssen sich der Startwert der Systemzeit an der Zeit des realen Systems und das Fortschreiten der Systemzeit an den ausgeführten Anweisungen und durchgeführten Operationen orientieren.

Eine Abhängigkeit von der Systemzeit könnte, abgesehen von der feststellbaren Abfrage der Systemzeit, auch genauer durch das von [Crandall u. a., 2006] vorgestellte Verfahren erkannt und analysiert werden. Dies fällt aber – wie auch der Umgang mit Anti-Emulationscode – nicht unter die Zielsetzung dieser Arbeit.

[13] Aktiver Anti-Emulationscode versucht den Emulationsvorgang zu erkennen und darauf basierend das Verhalten des Programms zu ändern. Passiver Anti-Emulationscode soll zu einem Abbruch des Emulationsvorgangs führen oder die für den Emulationsvorgang benötigte Zeit stark erhöhen.

4.3.2 Prozessorzeit

Abgesehen von der Systemzeit gibt es auch noch die interne Prozessorzeit, welche mit der „RDTSC“-Anweisung[14] abgefragt werden kann. Die Prozessorzeit wird bei Systemstart auf 0 gesetzt und mit jedem Prozessortakt erhöht.

Anhand der Prozessorzeit kann ungefähr bestimmt werden, wie viel Zeit seit Systemstart vergangen ist. Auch bietet die Prozessorzeit eine präzise[15] Möglichkeit, die von einigen Anweisungssequenzen benötigte Zeit zu vergleichen. Durch einen Vergleich kann in einigen Fällen festgestellt werden, ob die Programmausführung durch einen Debugger überwacht wird.

Ausgehend davon, dass es nicht viel Sinn ergibt, die Ausführung einer Entpackroutine nur auf Systeme zu beschränken, die erst kurz zuvor angeschaltet oder vor längerer Zeit gestartet worden sind, kann der Wert nahezu beliebig gewählt werden. Ein genauer Wertebereich kann möglicherweise anhand einer Statistik über die durchschnittliche Einschaltzeit von Endnutzersystemen bestimmt werden.

Ein rein interpretierender Emulator wird Genauigkeit auf Anweisungsebene erreichen [siehe Tijms, 2000, Seite 2ff]. Jede Anweisung benötigt hierbei aus logischer Sicht die gleiche Zeit. Die Prozessorzeit kann also für jede ausgeführte Anweisung um 1 erhöht werden. Das emulierte Programm wird bei diesem Vorgehen korrekt ausgeführt, solange es keine speziellen Kenntnisse über den – durch die CPUID-Anweisung – vorgegebenen Prozessortyp besitzt.

Aktiver Anti-Emulationscode könnte davon ausgehen, dass eine komplexe Anweisung mehr Zeit für die Ausführung benötigt als ein „NOP“[16]. Wird dies unter Verwendung der Prozessorzeit getestet, könnte der interpretierende Emulator erkannt werden. Um dies zu verhindern, müsste der Emulator basierend auf Taktgenauigkeit [siehe Tijms, 2000, Seite 2ff] implementiert oder eine Möglichkeit gefunden werden, die Prozessorzeit für unterschiedliche Anweisungen um unterschiedliche Beträge zu erhöhen.

4.4 Betriebssystemkomponenten

Einige Komponenten des Betriebssystems sind von besonderer Relevanz für die Emulation der Entpackroutine. Diese müssen demnach vorhanden und ausreichend korrekt implementiert sein, damit reguläre Entpackroutinen emuliert werden können. Diese Komponenten sind in der Regel durch [Russinovich u. Solomon, 2004] oder Microsofts MSDN[17] dokumentiert.

[14] ReaD Time Stamp Counter.

[15] Es ist möglich, dass dem Programm durch das Betriebssystem kurzzeitig die Rechenzeit entzogen wird, damit ein anderes Programm ausgeführt werden kann. In diesem Fall wäre die Messung ungenau, da die Prozessorzeit leicht erhöht wird, ohne dass eine Anweisung des Programms ausgeführt wird.

[16] No OPeration – eine Anweisung, die den Zustand des Prozessors und des Arbeitsspeichers nicht verändert.

[17] Siehe http://msdn2.microsoft.com/en-us/library/aa286538.aspx 08.07.2007.

Einige Komponenten sind nicht offiziell dokumentiert. In vielen Fällen ist allerdings eine inoffizielle Dokumentation von NTinternals[18] verfügbar.

Nach [Surauer, 2002, Seite 4] muss hierbei auch spezifisches Verhalten, also Fehler und Abweichungen von der Spezifikation, übernommen werden, da sich ein Programm darauf verlassen könnte.

Im User-Mode ausgeführte Programme können nicht direkt auf die Hardware oder sich auf Kernel-Ebene befindende Komponenten des Betriebssystems zugreifen. Sie können nur auf die vom Betriebssystem im User-Mode angebotene Funktionalität zugreifen und hierüber mit der Umgebung interagieren [Bayer u. a., 2006, Seite 5]. Deswegen brauchen auch nur entsprechende User-Mode-Betriebssystemkomponenten implementiert zu werden. Komponenten auf Kernel-Ebene oder sogar im System vorhandene Hardware braucht, sofern überhaupt erforderlich, nur anhand der für ein User-Mode-Programm relevanten Außenwirkung implementiert zu werden. Hierdurch wird der Implementierungsaufwand erheblich verringert.

Abgesehen von dem Lademechanismus für PE-Dateien, welcher größtenteils aus der Dokumentation des PE-Formats [siehe Microsoft, 2006] und aus durchgeführten Tests rekonstruiert werden kann, werden die in den folgenden Punkten beschriebenen Komponenten benötigt:

API-Funktionen Wie in Abschnitt 2.2.1 beschrieben, können Programme über API-Funktionen mit dem Betriebssystem interagieren – beispielsweise Speicher anfordern oder Abhängigkeiten laden. Ist eine vom Programm benötigte API-Funktion nicht implementiert, so kann das Programm möglicherweise – sogar unabhängig von der benötigten Wirkung der API-Funktion – nicht weiter korrekt emuliert werden, wenn nicht bekannt ist, wie viele Parameter an die Funktion übergeben worden sind [Szor, 2000, Seite 52]. Eine Möglichkeit, diesen Effekt zu reduzieren, kann die Speicherung der – automatisch auslesbaren – Anzahl der von jeder Funktion benötigten Parameter sein. So kann das Programm, sofern es die Auswirkung des API-Aufrufs nicht zwingend benötigt, ggf. normal weiter emuliert werden.

Es sollten alle häufig von Entpackroutinen genutzten Funktionen implementiert werden. Da der Zeitaufwand, alle Funktionen der Windows-API zu implementieren, zu hoch[19] ist, könnte die Verwendung einer selten genutzten – und somit nicht implementierten – Funktion als Anti-Emulator-Code verwendet werden.

Systemstrukturen Das Betriebssystem verwaltet einige Strukturen, die im virtuellen Adressraum jedes Programms angelegt werden. Ein Programm hat somit direkten Zugriff auf diese Strukturen. Durch Prüfung dieser Strukturen kann in einigen Fällen festgestellt werden, ob das Programm unter einem Debugger ausgeführt wird. Auch kann so ohne

[18] http://undocumented.ntinternals.net/ 14.07.2007.

[19] Nach [Eilam, 2005, Seiten 89f] umfasst allein die Grundfunktionalität ungefähr 2000 API-Funktionen.

Verwendung von API-Funktionen beispielsweise festgestellt werden, welche Abhängigkeiten bereits geladen worden sind. Diese Strukturen werden oft von Packern mit Schutzfunktion gelesen oder auch modifiziert.

Speicherverwaltung Wie in vorherigen Abschnitten schon kurz angeführt, wird der Arbeitsspeicher in Form von 4096 Byte umfassenden Speicherseiten verwaltet. Diese Verwaltung sollte nachgebildet werden, da sich einige Packer auf Eigenheiten[20] dieser Speicherverwaltung verlassen können. Zugriffsrechte werden auch nur auf der Basis von Speicherseiten gesetzt. Der Packer „Petite“ verwendet dies als Anti-Debugger-Trick.

Prozesse und Threads In der Regel benötigt eine Entpackroutine keine zusätzlichen Prozesse oder Threads. Allerdings können diese als Anti-Debugg- oder Anti-Emulationsmaßnahme verwendet werden [Szor, 2000, Seite 57].

4.5 Überwachung des Entpackvorgangs

Um erkennen zu können, wann die Entpackroutine beendet und die erste Anweisung des entpackten Programms ausgeführt wird, muss der Entpackvorgang überwacht werden.

Da das entpackte Programm zunächst nicht vorhanden ist, muss es durch Schreibvorgänge generiert werden, bevor es ausgeführt werden kann. Werden also alle Schreibvorgänge sowie ausgeführte Anweisungen überwacht, kann der Moment, in dem neuer[21] Code ausgeführt wird, fehlerfrei bestimmt werden [Christodorescu u. a., 2005b, Seite 7].

Der neue ausgeführte Code muss allerdings nicht zwingend das gesuchte entpackte Programm sein. Insbesondere Packer mit Anti-Debug-Funktionalität entschlüsseln zur Laufzeit eigene Unterroutinen, bevor das eigentliche Programm entpackt wird. Es können also, bevor das Programm entpackt wird, beliebig viele neue Anweisungen ausgeführt werden.

Um diese neuen Codesequenzen nicht mit dem entpackten Programm zu verwechseln, wird hier eine auf den grundlegenden Eigenschaften von Laufzeitpackern beruhende und im folgenden Abschnitt näher erläuterte Heuristik verwendet.

4.5.1 Entpackfortschritt

Es wird versucht zu ermitteln, wie weit der Entpackvorgang fortgeschritten ist. Für diese Heuristik werden zwei Annahmen gemacht:

- Ein Laufzeitpacker fügt möglichst nur benötigte Codesequenzen und Daten ein, um die Dateigröße gering zu halten.

[20] Wenn beispielsweise 1000 Byte Speicher angefordert werden, stellt das Betriebssystem dennoch eine komplette Speicherseite – also 4096 Byte – zur Verfügung.

[21] Ohne weitere Prüfung kann es sich hierbei jedoch auch um kopierten Code handeln.

- Das Programm ist entpackt, wenn die vom Laufzeitpacker hinzugefügten Routinen komplett ausgeführt wurden.

Folglich wird davon ausgegangen, dass das Programm frühestens dann entpackt ist, wenn alle relevanten Routinen ausgeführt und alle relevanten Daten verwendet wurden. Da Code und Daten in der gleichen Repräsentation vorliegen und die Unterscheidung dazwischen äquivalent zum Halteproblem ist [siehe Horspool u. Marovac, 1980], wird hier nicht versucht, diese statisch zu unterscheiden. Das Problem wird also auf die Verwendung aller relevanten Elemente des Programms reduziert.

Welche Elemente zur Laufzeit relevant sein werden, kann statisch nicht präzise bestimmt werden. Durch die Kenntnis der Bestandteile einer gepackten Datei im PE-Format ist aber eine im nächsten Abschnitt beschriebene Abschätzung möglich.

Durch die Bestimmung aller relevanten Elemente und ihrer Überwachung zur Laufzeit kann nach dieser Heuristik der relative und absolute Entpackfortschritt errechnet werden. Diese Information wird später bei der Entscheidung über eine Unterbrechung des Entpackvorgangs verwendet.

Da diese Information auf einer Heuristik basiert, kann es hierbei zu Fehlern in Form von Über- und Unterschätzen kommen. Diese Fehler haben aber, wie sich in den durchgeführten Tests[22] zeigt, lediglich einen länger als notwendig dauernden Emulationsvorgang zur Folge. Genügend Zeit vorausgesetzt, wird das Ergebnis des Entpackvorgangs nicht negativ beeinflusst.

4.5.2 Abschätzung relevanter Elemente

Wie in Abschnitt 2.3.3 vorausgesetzt, muss eine gepackte Datei mindestens Code, also eine Entpackroutine, enthalten. Normalerweise sind auch Quelldaten – das gepackte Programm – enthalten. Dies wird hier beachtet, aber nicht vorausgesetzt. Weiterhin muss zur Laufzeit ein Speicherbereich existieren, in den die entpackten Daten geschrieben werden. Quell- und Zielbereich können, insbesondere wenn nur entschlüsselt wird, identisch sein.

Da weder bekannt ist, in welchen Speicherbereich das entpackte Programm geschrieben werden wird, noch, wie viel Platz es belegen wird, kann dies nicht in die Abschätzung mit einfließen.

Es verbleiben also noch der Code und die möglicherweise vorhandenen Quelldaten als relevante Elemente. Da hier nicht zwischen Code und Daten unterschieden wird, hat es also keine Auswirkung, dass die Quelldaten nur optional sind.

Da die gepackte Datei im PE-Format vorliegt, sind weitere Elemente dieses Formats enthalten. Hierbei handelt es sich hauptsächlich um den Header und den Importabschnitt. Diese werden beim Programmstart mit in den Arbeitsspeicher geladen. Es ist möglich, in

[22] Siehe Kapitel 6.

diesen Bereichen einige für den Entpackvorgang relevante Daten unterzubringen. Da die Daten dort aber nur eine normalerweise vernachlässigbare Größe haben können, werden diese Abschnitte nicht als relevant für den Entpackvorgang betrachtet. Es müssen also alle Bereiche im Speicherabbild der Datei als relevant betrachtet werden, die keinem[23] der Elemente des PE-Formats zuzuordnen sind.

Aufgrund der Unterbringung von Code und Daten in Sektionen können Lücken[24] entstehen. Diese werden normalerweise mit 0 oder einer INT-3-Anweisung[25] gefüllt. Die Lücken müssen identifiziert und als nicht relevant eingestuft werden.

Nicht alle Sektionen eines Programms müssen von einem Laufzeitpacker auch zwangsweise gepackt werden. In Tests wurde eine „.rdata“ genannte Sektion von verschiedenen Packern ignoriert. Die enthaltenen Daten wurden dementsprechend während des Entpackvorgangs nicht verwendet. Da größere Sektionen einen starken Einfluss auf die Abschätzung des Entpackfortschritts haben, sollten diese anhand statistischer Tests[26] ermittelt werden.

Werden zu viele Bereiche als relevant eingestuft, wird der Entpackfortschritt unterschätzt. Werden mehr Bereiche von der Entpackroutine verwendet als angenommen, wird der Entpackfortschritt überschätzt.

Der folgende Abschnitt behandelt die Überwachung der relevanten Bereiche sowie die Frage, was „Verwendung“ in diesem Zusammenhang bedeutet.

4.5.3 Überwachung der Verwendung

Im vorherigen Abschnitt wurden Speicherbereiche festgelegt, die vermutlich während des Entpackvorgangs als Code oder Quelldaten verwendet werden. Um abschätzen zu können, wie weit der Entpackvorgang fortgeschritten ist, muss ermittelt werden, wie viel bereits verwendet wurde.

Besser wäre es zu wissen, wie viel tatsächlich *verbraucht* wurde, also während der weiteren Ausführung keinen relevanten[27] Einfluss mehr auf den Entpackvorgang haben wird. Hierzu wird angenommen, dass ein Speicherbereich, der ausgeführt wurde, auch verbraucht ist. Dies trifft für Codesequenzen, die nur ein einziges Mal ausgeführt werden, auch zu. In Schleifen hingegen werden Anweisungen öfter verwendet.

---

[23] Abgesehen von den Sektionen selbst, da diese Programmdaten enthalten.

[24] Siehe Abschnitt 2.3.3.

[25] Die INT-3-Anweisung wird von Debuggern als Haltepunkt und bei Ausführung ohne Debugger als Fehler betrachtet.

[26] Siehe Abschnitt 5.1.1.

[27] Es wäre möglich, Daten in einem Alphabet zu enkodieren, welches CPU-Anweisungen ohne Auswirkung entspricht. Die Sequenz könnte also zunächst ausgeführt und anschließend von einer anderen Routine dekodiert werden. Dies würde aber den Platzbedarf erheblich steigern und wurde auch bei keinem Laufzeitpacker beobachtet.

Nun wird davon ausgegangen, dass eine Schleife, also beispielsweise eine Entpackroutine, allein mit mehrfachen Durchläufen keinen Beitrag zum Entpackfortschritt liefert. Sie benötigt Quelldaten, die entschlüsselt oder entpackt werden können. Dementsprechend können auch Schleifenanweisungen nach der ersten Verwendung als verbraucht gekennzeichnet werden. Für den Entpackfortschritt ist in dieser Situation die Datennutzung maßgeblich.

Es ist schwierig, den Verbrauch von Daten zu bestimmen, da nicht bekannt ist, ob diese während der Ausführung mehrmals verwendet werden. Es wird davon ausgegangen, dass Daten eines gepackten Programms nur einmal gelesen zu werden brauchen. Ein Schlüssel kann jedoch sehr oft angewendet werden. Da ein mehrfach verwendeter Schlüssel im Verhältnis zu den Daten eines gepackten Programms nur sehr wenig Platz belegt, kann davon ausgegangen werden, dass Daten nach einmaliger Nutzung verbraucht sind. Es hat also keine relevante Auswirkung, den Schlüssel bereits nach einmaliger Nutzung als verbraucht zu kennzeichnen, da für den weiteren Entpackvorgang noch die Programmdaten damit entschlüsselt werden müssen.

Daten gelten nach einmaliger Nutzung als verbraucht. Wenn eine Entpackroutine zunächst eine Checksumme der gepackten Daten errechnet, liest diese alle Daten einmal ein. Würde ein beliebiger Lesevorgang mit konkreter Nutzung gleichgesetzt werden, könnten alle Daten bereits als genutzt gelten, bevor auch nur ein einziges Byte des Programms entpackt wurde. Daher muss die Nutzung genauer spezifiziert werden.

Nutzung bedeutet im Idealfall „für den Entpackvorgang in relevanter Weise genutzt und damit verbraucht“. Um dies genauer festlegen zu können, werden die grundlegenden Eigenschaften der für die Wiederherstellung des gepackten Programms notwendigen Routinen verwendet. Eine Entschlüsselungsroutine wird normalerweise ein oder mehrere Quellbytes einlesen, diese mit einem mehrfach verwendeten Schlüssel entschlüsseln und die gleiche Anzahl Bytes in einen Zielbereich schreiben, der mit dem Quellbereich identisch sein kann. Eine Dekompressionsroutine wird ein oder mehrere Quellbytes einlesen, dekomprimieren und hierbei in einen Zielbereich schreiben, wobei in den Zielbereich durchschnittlich mehr geschrieben als aus dem Quellbereich eingelesen wird.

Die grundlegende Funktionsweise ist also, Daten einzulesen und meistens veränderte Daten wieder zu schreiben. Da der Versuch, die konkrete Nutzung der Quelldaten für die Erzeugung der Zieldaten bei der Ausführung zu verfolgen, zu viel Zeit in Anspruch nehmen kann, wird dies unter Inkaufnahme möglicher Fehler ungenauer, aber schneller geprüft.

Lesevorgänge auf als relevant markierte Daten werden gespeichert. Findet anschließend ein Schreibvorgang auf einen Bereich statt, der bisher noch nicht unter Verwendung von Quelldaten beschrieben wurde, so werden die zuvor gelesenen Daten als vebraucht gekennzeichnet.

Hierdurch haben Vorgänge wie eine vorherige Überprüfung der Quelldaten oder ein Überschreiben von Speicherbereichen mit fixen oder Zufallsdaten keine relevante Auswirkung auf die Abschätzung des Entpackfortschritts.

4.6 Unterbrechung der Emulation

Es gibt vier Gründe, aus denen der Emulationsvorgang zeitweise angehalten, oder auch komplett abgebrochen werden sollte:

- Erreichen einer Anweisung, an der das Programm möglicherweise entpackt sein könnte.

  Der Emulationsvorgang wird zeitweise angehalten, wenn die aktuelle Anweisung durch eine schnelle Überprüfung als mögliche erste Anweisung des entpackten Programms eingeordnet wurde. Anhand weiterer Tests wird dann genauer überprüft, ob das Programm an der aktuellen Anweisung wirklich entpackt sein könnte. Ist dies der Fall, wird die Emulation abgebrochen, da davon ausgegangen wird, dass das Programm vollständig entpackt wurde. Zeigt sich durch die Tests, dass das Programm an der aktuellen Anweisung wahrscheinlich noch nicht vollständig entpackt wurde, wird die Emulation fortgesetzt. Hierdurch soll eine andere Anweisung gefunden werden, an der das Programm zu einer größeren Wahrscheinlichkeit vollständig entpackt ist.

- Verwendung der Windows API durch das entpackte Programm.

  Es ist möglich, dass die erste Anweisung des entpackten Programms nicht erkannt wird. In diesem Fall wird auch das entpackte Programm emuliert. Dieses wird in der Regel Funktionen der Windows API[28] nutzen. Die genutzten Funktionen unterscheiden sich, besonders im Fall von mit Hochsprachen geschriebenen Programmen, von den typischerweise von Entpack- und Anti-Debug-Routinen verwendeten Funktionen. Einige Viren verwenden nach [Szor, 2001] bestimmte API-Aufrufe in der Entschlüsselungsroutine, um eine weitere Emulation zu verhindern. Auch in der Entpackroutine könnten nahezu beliebige API-Funktionen aufgerufen werden, um vorzutäuschen, dass das Programm bereits entpackt sei. Es kann daher schwer sein, anhand der aufgerufenen Funktion festzustellen, ob diese von der Entpackroutine oder vom entpackten Programm aus aufgerufen wurde.

- Überschreiten eines Limits.

  Der Emulationsvorgang darf nicht unendlich lange dauern können. Daher muss der Emulationsvorgang anhand einer Bedingung, beispielsweise des Ablaufs einer bestimmten Zeitspanne oder der Emulation einer festgelegten Gesamtzahl an Anweisungen, abgebrochen werden.

- Erreichen eines Endzustands.

  Ein Endzustand ist erreicht, wenn die Emulation des Programms nicht normal fortgesetzt werden kann. Im Normalfall würde auch die native Ausführung des Programms in so einem Zustand durch das Betriebssystem abgebrochen werden.

[28] Siehe Abschnitt 2.2.

Typische Endzustände werden durch die Ausführung einer das Programm regulär beendenden Anweisung, sowie das Erzeugen eines nicht abgefangenen Fehlers erreicht.

Die Verwendung von API-Aufrufen zur Erkennung eines erfolgreichen Entpackvorgangs wird hier nicht näher erörtert. Dieses Kriterium würde bei Verwendung nur in einigen Fällen Zeit sparen können, dafür aber das Risiko eines zu frühen Abbruchs des Entpackvorgangs steigern. Auf die anderen drei Punkte wird in den folgenden Unterkapiteln näher eingegangen.

4.6.1 Möglicherweise erste Anweisung des entpackten Programms

Wie schon in Kapitel 4.5 erläutert, ist die Ausführung neuer Daten eine notwendige, aber keine hinreichende Bedingung dafür, dass das Programm vollständig entpackt ist. Daher müssen zunächst weitere Tests durchgeführt werden. Ein entsprechendes Flussdiagramm ist Abbildung 4.1 dargestellt. Die hier aufgeführten Aktionen und Tests werden in den folgenden Abschnitten erläutert.

Der von [Christodorescu u. a., 2005b, Seite 7] angegebene Algorithmus generiert, sobald eine zuvor geschriebene Anweisung ausgeführt wird, aus allen Schreibvorgängen innerhalb der Sektionen der Datei im Speicher ein neues „entpacktes“ Programm. Da aber bei vielen Packern die erste ausgeführte neue Anweisung nicht der ersten Anweisung des entpackten Programms entspricht, wird zumindest der OEP falsch bestimmt. Einige Entpackroutinen schreiben zudem, wie in Abschnitt 2.3.3 beschrieben, relevante Daten außerhalb der Sektionsgrenzen. Diese werden im angegebenen Algorithmus verworfen. Folglich können solche gepackten Dateien nicht mit dem angegebenen Algorithmus entpackt werden.

In den von [Christodorescu u. a., 2005b, Seite 10] durchgeführten Tests wurden ausschließlich[29] Packer verwendet, bei denen das Programm bereits innerhalb der Sektionsgrenzen entpackt ist, wenn die erste zuvor geschriebene Anweisung ausgeführt wird. Daher wurden alle Programme vollständig entpackt. Der gefundene OEP war bei einigen Tests allerdings nicht korrekt. Das ist darauf zurückzuführen, dass die Entpackroutine eine eigene generierte Anweisung ausführt, bevor zum bereits entpackten Programm gesprungen wird.

Der von [Christodorescu u. a., 2005b, Seite 7] angegebene Algorithmus kann unter den hier geltenden Bedingungen nicht verwendet werden. Es existieren viele Packer, die neue Anweisungen vor Beginn des Entpackvorgangs ausführen. Hiermit gepackte Dateien würden also nicht entpackt werden können. Die Ausführung neuer Daten besteht aber weiterhin als notwendige Bedingung.

[29] Die Versionsnummer der verwendeten Packer wurde nicht angegeben. Zu jedem genannten Packer wurde aber nach durchgeführten Analysen mindestens eine Version gefunden, die das im Weiteren genannte Verhalten zeigt.

4.6.1.1 Tests nach Eintreten der notwendigen Bedingung

Wenn die zuvor genannte notwendige Bedingung zutrifft, muss anhand weiterer Tests abgeschätzt werden, ob das Programm bereits entpackt sein kann und die ausgeführte Anweisung die erste Anweisung des neuen Programms ist.

Bei diesen Tests muss die Markierung der zuvor geschriebenen, aber noch nicht ausgeführten Speicherbereiche beibehalten werden. Ansonsten könnte die erste Anweisung des entpackten Programms verpasst werden, wenn sie nicht der ersten neuen Anweisung entspricht. Dies könnte passieren, wenn beispielsweise die Routine zur Behandlung der Importtabelle sowie das eigentliche Programm von der Entpackroutine generiert werden. Danach wird die Routine zur Behandlung der Importtabelle ausgeführt. Da sie neu ist, wird ein Zwischentest durchgeführt. Werden hierbei die Markierungen gelöscht, gilt die erste Anweisung des entpackten Programms bei der späteren Ausführung nicht mehr als neu.

Die Entpackroutine darf, wie in Abschnitt 2.3.3 beschrieben, nicht direkt in das entpackte Programm übergehen. Es wird daher ein – ggf. indirekter – Sprung von dem Code des Laufzeitpackers zum OEP des entpackten Programms vorausgesetzt. Es wird angenommen, dass die letzte Anweisung der Entpackroutine und die erste Anweisung des entpackten Programms nicht in unmittelbarer Nähe zueinander liegen.

Aufgrund vorheriger Annahme werden nur Sprünge weiter betrachtet, bei denen Start und Ziel mindestens 127 Byte auseinanderliegen und sich nicht innerhalb derselben Speicherseite (4096 Byte) befinden (siehe die in Abbildung 4.2 dargestellten Speicherseiten). Hierdurch wird die Wahrscheinlichkeit reduziert, dass zuvor entschlüsselte Programmteile der Entpackroutine als mögliche OEPs überprüft werden. Dies vereinfacht und beschleunigt die Suche nach dem OEP.

[Royal u. a., 2006, Seite 2] schlagen vor, den neuen Code mit Codeabschnitten zu vergleichen, die bei einer im Vorfeld durchgeführten Analyse der Datei gefundenen wurden. Hierdurch wird sichergestellt, dass der Code wirklich neu ist, und es sich nicht um bereits vorhandenen Code handelt der nur in einen anderen Speicherbereich kopiert wurde.

Sprünge zu neuen Anweisungen, die diesen Kriterien genügen, werden als mögliche Sprünge zum OEP betrachtet. Daher wird der aktuelle Zustand des Emulators im diesem Fall zur späteren Verwendung gespeichert.

4.6.1.2 Tests für einen möglichen OEP

Nun werden weitere Tests durchgeführt, um festzustellen, ob die mögliche OEP-Adresse auch wahrscheinlich die korrekte OEP-Adresse ist. Zunächst wird der in Abschnitt 4.5 beschriebene Entpackfortschritt überprüft. Er wird erst an dieser Stelle verwendet, damit sich ein mögliches Unterschätzen des Entpackfortschritts nicht negativ auf die OEP Ermittlung auswirken kann. Der Entpackvorgang gilt als weit genug fortgeschritten, wenn mindestens

95% der relevanten Daten verwendet wurden oder weniger als 100 Byte ungenutzte Daten verbleiben. Diese festen Werte könnten ggf. nach weiteren Tests und Anwendung von Heuristiken auch dynamisch zur Laufzeit gesetzt werden.

Das entpackte Programm besteht aus mindestens einer Sektion, welche mindestens eine Speicherseite umfasst. Entsprechend Abschnitt 2.3.3 kann sich Code der Entpackroutine nur außerhalb der zusammenhängenden Sektionen des entpackten Programms oder in möglichen Sektionslücken befinden (siehe Abbildung 4.3. Anhand dieser Voraussetzung und der während des Entpackvorgangs erfolgten Überwachung kann überprüft werden, ob die aktuelle Anweisung nicht der OEP sein kann.

Entsprechend den Voraussetzungen müssen alle Daten vom Beginn der Speicherseite bis zur aktuellen Anweisung zum entpackten Programm gehören (siehe Abbildung 4.4. Alle folgenden Daten auf der Speicherseite könnten auch zur Entpackroutine gehören. Dies kann verifiziert werden, indem geprüft wird, ob die Daten vom Beginn der Speicherseite bis zur aktuellen Anweisung *neu* sind. Daten sind neu, wenn sie zuvor geschrieben und seit dem Schreibvorgang nicht ausgeführt wurden. Eine mögliche Ausführung vor dem letzten Schreibvorgang ist somit nicht von Bedeutung.

Wären die Daten zuvor nicht geschrieben oder zwischenzeitlich teilweise ausgeführt worden, wäre die aktuelle Anweisung der Entpackroutine zuzuordnen – sie könnte also nicht der OEP sein.

An dieser Stelle besteht noch die Möglichkeit, dass es sich bei der aktuellen Anweisung um eine zur Laufzeit generierte Unterroutine des Entpackcodes handelt, die sich auf einer neuen Speicherseite befindet und mit einem längeren Sprung ausgeführt wird. Um die Wahrscheinlichkeit hierfür zu reduzieren, wird auch die Speicherseite vor der aktuellen Speicherseite überprüft.

Falls die aktuelle Anweisung tatsächlich die erste Anweisung des entpackten Programms ist, wird die davorliegende Speicherseite auch zum entpackten Programm gehören. Falls die erste Anweisung in der ersten Speicherseite der ersten Sektion liegt, könnte die vorherige Sektion auch zum Header gehören, sofern dieser vorhanden ist.

Damit die Daten der vorherigen Seite zum entpackten Programm oder dem Header gehören können, dürfen sie nicht zum Entpackfortschritt beigetragen haben oder müssen seitdem überschrieben worden sein (siehe Abbildung 4.5. Hierdurch werden allerdings zwei Möglichkeiten gegeben, bei denen die aktuelle Anweisung zunächst fälschlicherweise nicht als erste Anweisung des entpackten Programms erkannt werden wird.

Falls sich der OEP direkt in der ersten Speicherseite der ersten Sektion befindet und der Packer Code im Header ausgeführt[30] hat, der nicht überschrieben wurde, kann die aktuelle Anweisung zunächst nicht als OEP erkannt werden (siehe zuvor genannte Abbildung 4.5).

[30] Daten des Headers werden wie in Abschnitt 4.5.2 erläutert nicht als relevant eingestuft.

Dies könnte auch der Fall sein, wenn sich der OEP ungewöhnlicherweise nicht innerhalb der ersten Sektion befindet und eine Lücke in der vorherigen Sektion von der Entpackroutine genutzt worden wäre.

Sind alle Bedingungen erfüllt, gilt die aktuelle Anweisung als wahrscheinlicher OEP.

4.6.1.3 Falsifizierung des OEP

Es könnte zunächst versucht werden, den wahrscheinlichen OEP zu verifizieren. Eine Überprüfung ist wichtig, da der Emulationsvorgang ansonsten zu früh abgebrochen werden könnte. Die Frage, ob die gefundene Anweisung die erste Anweisung des entpackten Programms ist und das Programm somit normal ausgeführt werden wird, ist vermutlich äquivalent zum Halteproblem und somit nicht generell entscheidbar. Daher wird der möglicherweise einfachere Weg der Falsifizierung gewählt. Das Problem ließe sich vereinfachen, indem davon ausgegangen wird, dass es sich bei dem entpackten Programm um ein in einer Hochsprache geschriebenes Programm handele. Solche Programme können meistens, insbesondere am Einstiegspunkt, an compilergenerierten Strukturen erkannt werden. [Graf, 2005] schlägt daher vor, die Emulation an compilererzeugten Einstiegspunkten abzubrechen. Bei polymorphen Viren kann die Emulation nach [Szor, 2005, Kapitel 11.4] abgebrochen werden, wenn die folgenden Anweisungen einen bekannten Hashwert ergeben. Entsprechend könnte dies auch hier geprüft werden.

Durch diese Vereinfachung wäre diese Prüfung aber nicht mehr generisch. Bei einigen compilergenerierten Programmen können die Anweisungen am Einstiegspunkt entfernt oder verändert werden, so dass es nicht mehr als solches erkannt wird. In dem Fall würde die Verifikation fehlschlagen. Deswegen wird eine Falsifizierung versucht.

Der Einstiegspunkt eines normalen[31] Programms muss spezielle Eigenschaften besitzen. Denn an dieser Stelle wird das Programm gestartet – es muss also unabhängig von den Werten in den regulären CPU-Registern oder einer speziellen Stackbelegung funktionieren. Durch Ausnutzung dieser Bedingung kann möglicherweise erkannt werden, ob die aktuelle Anweisung nicht der OEP des entpackten Programms sein kann.

Wenn der Einstiegspunkt eines Programms modifiziert wird, so dass es an einer beliebigen anderen Anweisung gestartet wird, stürzt es mit hoher Wahrscheinlichkeit beim Start ab, da ein zuvor gesetzter Kontext in den CPU-Registern oder einigen Speicherbereichen erwartet wird. Dieser Umstand wird für diesen Test genutzt.

Alle CPU-Register sowie der Stack werden auf die Belegung bei Programmstart zurückgesetzt. Jeglicher Speicher, der nicht mit dem Speicherblock an der aktuellen Anweisung verbunden ist, wird freigegeben. Es wird also ein Programmstart an der aktuellen Anweisung

[31] „Normal“ bedeutet hier, dass das Programm keine spezielle Laufzeitumgebung voraussetzen darf und dementsprechend auch nicht auf einen Packer vorbereitet sein kann. Es wird also beispielsweise unter Windows 2000 und Windows XP unabhängig vom installierten Servicepack funktionieren.

simuliert. Durch Emulation einer festgelegten Anzahl von Anweisungen wird geprüft, ob ein Endzustand durch einen aufgetretenen Fehler erreicht wird. Tritt dieser Fall ein, gilt die zuvor gefundene Anweisung nicht mehr als wahrscheinlicher OEP.

Wenn der wahrscheinliche OEP durch den vorherigen Test nicht falsifiziert werden konnte, kann ein weiterer Test zum Nachweis einer Kontextabhängigkeit durchgeführt werden. Für diesen wird eine Aufzeichnung der bei vorherigen Test emulierten Anweisungen sowie der vor dem Test gesicherte Emulatorzustand benötigt. Das Programm wird aus dem zuvor gesicherten Zustand heraus normal weiter emuliert. Hierbei werden die ausgeführten Anweisungen mit der vorherigen Aufzeichnung verglichen. Ergeben sich hier keine Unterschiede, konnte die Annahme, bei der entsprechenden Anweisung handele es sich um den OEP, nicht falsifiziert werden. Sie gilt somit weiterhin als OEP.

Treten bei dem Vergleich der Anweisungen Unterschiede auf, kann dies zwei Gründe haben. Im einfacheren Fall befindet sich der OEP nicht an der angenommenen Stelle. Dementsprechend verhält sich der Code mit dem nun vorhandenen Kontext anders. Der komplexere Fall tritt ein, wenn es sich zwar um den OEP handelt, im folgenden Programm aber Junk Code[32] ausgeführt wird.

Ein Beispiel hierfür ist der in Abbildung 4.6 dargestellte Junk Code. Hier wird abhängig von C unterschiedlicher Code ausgeführt. Da die Funktion des Codes jedoch identisch ist, wird diese unabhängig von C durchgeführt. Der Codefluss würde sich also ändern, obwohl das Ergebnis der ausgeführten Anweisungen identisch bleibt. Daher sollte, um diese Überprüfung verlässlicher zu machen, versucht werden, Junk Code zu entfernen. Die Entfernung von Junk Code wird in der Arbeit von [Christodorescu u. a., 2005b] näher erläutert.

Mit den genannten Methoden kann die Annahme einer OEP-Position falsifiziert werden. Allerdings kann ein falscher OEP, insbesondere aufgrund des Anweisungslimits, nicht in jedem Fall damit erkannt werden.

4.6.2 Nicht rechtzeitig unterbrochen

Wenn der Entpackfortschritt unterschätzt wird oder der OEP unter ungünstigen Bedingungen nicht durch die in Abschnitt 4.6.1 genannten Heuristiken erkannt wird, kann der Emulationsvorgang über den OEP hinauslaufen. Er wird dann beendet, wenn das Zeitlimit überschritten oder ein Endzustand erreicht wurde. Anhand des errechneten Entpackfortschritts kann anschließend grob abgeschätzt werden, ob das Programm bereits entpackt worden ist.

Falls entsprechend Abschnitt 4.6.1.1 während der Emulation mögliche OEPs aufgezeichnet wurden, kann der OEP möglicherweise noch nachträglich gefunden und das Programm somit korrekt entpackt werden.

[32] Code ohne eigentliche Funktion bzw. bedingt ausgeführter Code mit identischer Funktion, der dazu gedacht ist, eine Analyse zu erschweren.

Hierzu wird der Zeitpunkt des letzten relevanten Schreibvorgangs (entsprechend Abschnitt 4.5.3) mit dem Zeitpunkt der OEP-Aufzeichnungen verglichen. Der erste mögliche OEP, der nach dem letzten relevanten Schreibvorgang gefunden wurde, gilt als wahrscheinlicher OEP.

Dies basiert auf der Annahme, dass entpackte Programme keine relevanten Schreibvorgänge durchführen können. Die Programme wurden erst beim Entpackvorgang generiert, wodurch ihre ebenfalls generierten Datenbereiche nicht als relevant gelten. In dem Fall, dass ein Teil des Programms nicht gepackt wurde, wäre es allerdings möglich, dass es relevante Schreibvorgänge durchführt. Deswegen sollte ein Falsifizierungsversuch für den wahrscheinlichen OEP durchgeführt werden.

Falls das gesamte Programm gepackt war, in einen nicht als relevant eingestuften oder bereits verbrauchten Bereich geschrieben wurde und die Anweisung am OEP vor Abbruch des Emulationsvorgangs aufgezeichnet wurde, kann das entpackte Programm durch eine Wiederherstellung des Emulatorzustands so gespeichert werden, als wäre die Emulation direkt am OEP unterbrochen worden.

4.6.3 Endzustand erreicht

Es gibt verschiedene Arten von Endzuständen und verschiedene Wege, diese zu erreichen.

- Programmbeendigung, bevor das Programm entpackt wurde.

  Da die Entpackroutine zur Aufgabe hat, das Programm vollständig zu entpacken, kann in diesem Fall von einem Fehler ausgegangen werden. Typischerweise passiert dies durch eine unzureichende Emulation oder ungeeignete Umgebungsparameter.

  - Unzureichende Emulation: Durch einen Fehler im Emulator können Berechnungen oder API-Aufrufe ein falsches Ergebnis liefern. Als Folge könnte das Programm, wenn die Ergebnisse nicht zuvor geprüft werden, durch einen Zugriff auf ungültige Speicherbereiche oder auch durch Ausführung ungültiger Anweisungen beendet werden. Falls eine Routine zur Fehlerbehandlung eingerichtet wurde, kann sich das Programm nach so einem Fehler auch regulär beenden.
  - Ungeeignete Umgebungsparameter: Wenn das Betriebssystem ein Programm lädt und benötigte Abhängigkeiten nicht verfügbar sind, wird der Start mit einer Fehlermeldung abgebrochen. Wie in Abschnitt 2.3.1 bereits dargestellt, übernimmt die Entpackroutine meistens das Laden der vom entpackten Programm benötigten externen Funktionen. Ist eine benötigte Funktion nicht verfügbar, kann dies als Fehler erkannt und das Programm durch die Entpackroutine regulär beendet werden.

- Programmbeendigung, nachdem das Programm entpackt worden ist. Die Ausführung kann auch vom Programm selbst aus verschiedenen Gründen beendet werden.
  - Reguläre Beendigung: Der ablaufende Programmpfad ist möglicherweise unter den gegebenen Umgebungsbedingungen sehr kurz. Beispielsweise könnte sich das Programm, sofern es sich nicht bereits in einem Systemverzeichnis befindet, dorthin kopieren, einen Autostart-Eintrag anlegen und sich beenden. Es wird in diesem Fall erst beim nächsten Systemstart wieder aktiv werden. Natürlich kann der ablaufende Programmpfad auch unabhängig von den Umgebungsbedingungen nur sehr kurz sein.
  - Unzureichende Emulation: Wie in der Entpackroutine könnte auch hier ein aufgrund unzureichender Emulation auftretender Fehler zur Beendigung des Programms führen. Dies geschieht meistens durch unzureichende Implementierung der API. Programme nutzen in der Regel sehr viel mehr Funktionen als eine Entpackroutine. Wenn diese Funktionen nicht vollständig im Emulator implementiert sind, kann es zu einem Fehler und der damit verbundenen Programmbeendigung kommen.

Möglicherweise wird von der Entpackroutine oder dem Programm auch gezielt nach einer Emulationsumgebung gesucht. Bei einer Erkennung kann so die weitere Ausführung in einer emulierten Umgebung verhindert werden.

Generell ist es schwer zu entscheiden, ob die Beendigung regulär oder aufgrund eines Fehlers erfolgte. Auch ob das Programm während der Ausführung der Entpackroutine oder des bereits entpackten Programms beendet wurde, ist nicht einfach zu bestimmen. Daher wird nach Programmbeendigung anhand der bei der Überwachung des Entpackvorgangs aufgezeichneten Daten nach einem Zeitpunkt gesucht, an dem das Programm möglicherweise bereits entpackt war. Das Verfahren wurde bereits in Abschnitt 4.6.2 beschrieben.

4.6.4 Zeitlimit überschritten

Der Entpackvorgang darf, wie in den Anforderungen (siehe Kapitel 3) festgelegt, nicht sehr lange und keinesfalls unendlich lange dauern. Allerdings ist vorher nicht bekannt, wie lange der Entpackvorgang für eine konkrete Datei dauern wird.

Beim Emulationsvorgang werden die Anweisungen des Programms interpretiert und ihre Wirkung simuliert. Falls das Programm eine Endlosschleife ausführt, würde diese auch unendlich lange emuliert werden. Wenn das Programm sich beendet oder das entpackte Programm erkannt wird, endet auch der Emulationsvorgang.

Dies ist nach [Cifuentes, 1994, Seite 2] ein unlösbares, aber partiell berechenbares („semientscheidbares“) Problem, da ein Verfahren – der Emulationsvorgang – existiert, welches beendet wird, wenn das Programm beendet wird, und ansonsten nicht terminiert.

Das Problem ist also äquivalent zum Halteproblem. Es existiert kein Verfahren, mit dem im Vorwege allgemein feststellbar ist, ob ein Endzustand erreicht werden wird oder nicht.

Dementsprechend kann auch nicht generell zwischen einer langen Berechnung und einer möglicherweise durch einen Fehler verursachten oder auch bewusst konstruierten Endlosschleife unterschieden werden. Unabhängig davon, was das Programm gerade macht, muss es nach einer festgelegten Zeit beendet werden. Anschließend sollte entsprechend Abschnitt 4.6.2 nachträglich der OEP gesucht werden, da dieser möglicherweise schon überschritten wurde.

Durch die Verwendung eines absoluten Zeitlimits, beispielsweise 10 Sekunden, würde der Entpackvorgang auf jedem Computer höchstens 10 Sekunden dauern. Dieses Vorgehen entspricht aber nicht der Anforderung der Funktionssicherheit. Denn der Entpackerfolg könnte so direkt von der Prozessorgeschwindigkeit oder der aktuellen Prozessorlast abhängen.

Besser wäre es, das Limit durch die Anzahl emulierter Anweisungen festzulegen. Dadurch kann das Ergebnis des Entpackvorgangs, unabhängig vom verwendeten Prozessor und der aktuellen Prozessorlast, identisch sein.

Das festgelegte Anweisungslimit kann aber von einem Angreifer durch Tests ermittelt werden. Der Angreifer könnte dann eine Entpackroutine so modifizieren, dass das Anweisungslimit erreicht wird, bevor das Programm entpackt ist [Stepan, 2005, Seite 41].

Das Anweisungslimit dynamisch, also basierend auf dem Programm und seinem Verhalten zur Laufzeit, festzulegen würde das Austesten erschweren. Weiterhin könnten dadurch auch Programme mit langsamer Entpackroutine, die möglicherweise aufgrund eines festen Limits abgebrochen würde, erfolgreich entpackt werden.

Die Überwachung „aktiver Anweisungen“ ist eine für polymorph verschlüsselte Viren gebräuchliche Methode zur Erhöhung des Anweisungslimits [Szor, 2005, Kapitel 11.4]. Eine Anweisung gilt als aktiv, wenn sie den Speicher an zwei aufeinanderfolgenden Adressen verändert. Dies ist typisch für einen Entschlüsselungsvorgang. Das Vorhandensein aktiver Anweisungen kann auch beim Entpackvorgang dazu genutzt werden, das Anweisungslimit dynamisch zu erhöhen.

Es können nicht nur Anweisungen, sondern auch die Speichernutzung überwacht werden. Die in Abschnitt 4.5.1 definierte Abschätzung des Entpackfortschritts kann auch verwendet werden, um das Anweisungslimit bei kontinuierlichem Fortschritt zu erhöhen. Dadurch könnte der vorzeitige Abbruch eines nahezu abgeschlossenen langsamen Entpackvorgangs vermieden werden.

Abgesehen von der Erhöhung des Anweisungslimits durch das Programmverhalten, sollte auch vor Emulationsstart ein Grundlimit gesetzt werden. Dieses kann auf der Dateigröße basieren, da bei größeren Dateien normalerweise mehr Anweisungen für den Entpackvorgang benötigt werden als bei kleinen Dateien. Unabhängig von der Dateigröße muss das

Grundlimit dennoch ausreichend hoch gesetzt werden, so dass auch kleine Dateien, die eine langsame Entpackroutine besitzen, erfolgreich entpackt werden können.

4.7 Rekonstruktion der entpackten Datei

Nachdem der Entpackvorgang erfolgreich abgeschlossen ist, befindet sich das entpackte Programm im (emulierten) Arbeitsspeicher. Durch den korrekt erkannten OEP ist auch die Position einer Speicherseite der Codesektion des entpackten Programms bekannt.

Laut [Szor, 2005, Kapitel 7.6.3] kann auch ein kleiner Abschnitt des entpackten Programms ausreichen, um es als Malware zu identifizieren. Je nach Malware und vorhandenen Signaturen kann es aber sein, dass die Speicherseite des OEPs nicht für eine Erkennung ausreicht. Daher sollte das gesamte entpackte Programm für eine Überprüfung verfügbar sein.

Alle Sektionen eines Programms folgen direkt aufeinander. Zwischen diesen Sektionen kann es also keine Adressbereiche geben, denen vom Betriebssystem kein Speicher zugewiesen[33] wurde. Dementsprechend enthält der den OEP umgebende Speicherbereich mindestens das komplette entpackte Programm. Wird vom OEP aus in beide Richtungen nach nicht zugewiesenen Speicherseiten gesucht, können die Grenzen bestimmt werden, zwischen denen sich das entpackte Programm befinden muss.

Das entpackte Programm muss nun als Datei im PE-Format gespeichert werden. Hierbei ergeben sich drei Fragestellungen:

- Wie kann ein gültiger PE-Header garantiert werden?
- Welche Bereiche innerhalb der gefundenen Obergrenzen enthalten ausschließlich das entpackte Programm?
- Wie lässt sich die Importtabelle wiederherstellen?

Auf diese Fragen wird in den folgenden Abschnitten eingegangen.

4.7.1 Gültiger PE-Header

Damit eine Datei im PE-Format korrekt les- und ausführbar ist, muss diese einen der PE-Spezifikation entsprechenden PE-Header enthalten. Die sich im PE-Header befindenden Strukturinformationen müssen eine fehlerfreie Ausführung des entpackten Programms zulassen. Daher sollten sie möglichst genau den ursprünglichen Strukturinformationen des ungepackten Programms entsprechen. Hierzu werden drei verschiedene Ansätze vorgestellt, von denen nur der dritte ein verlässliches Ergebnis zur Folge hat.

[33] Jedes Programm kann zwar über einen großen virtuellen Adressraum verfügen, jedoch ist nicht jede Adresse sofort benutzbar. Zunächst muss dort Speicher vom Betriebssystem angefordert werden.

In jedem Fall sollte der Einstiegspunkt im PE-Header korrekt eingetragen werden. In den Tests von [Christodorescu u. a., 2005b, Seite 10] wurde festgestellt, dass einige rekonstruierte Programme erst dann durch einen Malware-Scanner erkannt wurden, wenn der EP auf den richtigen Wert gesetzt wurde. Dies kann passieren, wenn EP-basierende Signaturen verwendet werden. Mit diesen werden Daten in relativer Entfernung zum EP geprüft. Ist der EP also nicht richtig eingetragen, können diese Signaturen im Normalfall zu keiner Erkennung führen.

4.7.1.1 Übernahme des PE-Headers aus der Datei

Die gepackte Datei enthält einen gültigen PE-Header. Dieser kann unter bestimmten Voraussetzungen direkt übernommen werden. Lediglich der angegebene Einstiegspunkt muss auf den gefundenen OEP gesetzt werden.

Falls die Struktur des ungepackten Programms beibehalten und die Sektionen lediglich verschlüsselt wurden, wird der Header auch noch für die entschlüsselten Sektionen gültig sein. Es hat sich in diesem Fall nur der für den PE-Header nicht weiter relevante Inhalt der Sektionen verändert.

Es ist aber schwer festzustellen, ob das Programm wirklich nur aus den zuvor entschlüsselten Daten besteht. Da die Sektionen in der Datei an 512 Byte, die Sektionen im Speicher aber an 4096 Byte ausgerichtet sind, können die bereits in Abschnitt 2.3.3 erläuterten Lücken auftreten. Werden während des Entpackvorgangs Daten in so eine Lücke geschrieben, können diese keinen Daten der ursprünglichen Datei zugeordnet werden, da sie sich außerhalb der für die Datei spezifizierten Sektionsgrenzen befinden. Da Entpackroutinen diese Lücken auch für eigene Daten nutzen können, kann nicht mit absoluter Sicherheit ausgeschlossen werden, dass die Daten zum entpackten Programm gehören. In diesem Fall müsste der Header angepasst werden.

Der PE-Header enthält Zugriffsrechte für die einzelnen Sektionen. Hier wird festgelegt, ob die Daten einer Sektion gelesen, geschrieben oder ausgeführt werden dürfen. Diese Berechtigungen können dynamisch zur Laufzeit verändert werden, ohne dass die Werte im PE-Header hierzu verändert werden müssen. Es wäre also möglich, dass der PE-Header unpassende Zugriffsrechte für die einzelnen Sektionen des entschlüsselten Programms enthält.

Die vom PE-Header spezifizierten Import-Einträge können zur Entschlüsselungsroutine gehören, aber auch zum entschlüsselten Programm. Ggf. ist nicht eindeutig bestimmbar, ob die Import-Einträge auf die verwiesen wird, auch die richtigen Einträge für das entschlüsselte Programm sind. Für weitere Informationen zur Rekonstruktion der Importtabelle wird auf Abschnitt 4.7.3 verwiesen.

Der vorhandene PE-Header kann also unter sehr speziellen Bedingungen aus der Datei übernommen werden. Allerdings ist das Zutreffen dieser Bedingungen möglicherweise nicht

genau bestimmbar. Falls das Programm entpackt wurde – möglicherweise sogar in einen Speicherbereich ausserhalb der Sektionsgrenzen – ist es nicht sinnvoll, den Header zu übernehmen. Daher wird diese Methode hier nicht verwendet.

4.7.1.2 Übernahme des PE-Headers aus dem Speicherabbild

Der PE-Header wird genau wie die Sektionen bei Programmstart in den Speicher geladen. Wenn die erste Anweisung des entpackten Programms ausgeführt wird, kann sich der PE-Header in vier möglichen Zuständen im Speicher befinden.

- PE-Header unverändert gegenüber der Datei.

  Wenn das Programm, wie in Abbildung 2.4 dargestellt, in vorhandene Sektionen entpackt wird, bleibt der PE-Header im Speicher oft unverändert. Dies kann durch einen Vergleich mit dem in der Datei enthaltenen PE-Header exakt festgestellt werden. Bei der Übernahme würden also die bereits im vorherigen Abschnitt 4.7.1.1 beschriebenen Bedingungen gelten.

- PE-Header entspricht dem des ungepackten Programms.

  Insbesondere wenn das entpackte Programm in einen neu angeforderten Speicherbereich entpackt wird, kann der geschriebene Header exakt dem des ungepackten Programms entsprechen. Einige Laufzeitpacker bieten auch die Option an, den ursprünglichen PE-Header aus Kompatibilitätsgründen in jedem Fall wiederherzustellen. Dies ist insbesondere für mehrfach gepackte Dateien relevant, da im PE-Header benötigte Daten enthalten sein können.

  Es ist unmöglich zu verifizieren, ob der gefundene PE-Header der des ungepackten Programms ist, da dieser eben nicht bekannt ist. Daher kann nur überprüft werden, ob dieser Header auf die gefundenen Daten möglicherweise zutreffen kann. Dies wird im folgenden Unterpunkt beschrieben.

- PE-Header wurde modifziert.

  Eine Modifikation des bestehenden PE-Headers im Speicher kann durch einen Vergleich mit der Datei exakt festgestellt werden. Die Modifikationen können verschiedene Gründe haben. Hier ist die Frage wichtig, ob der PE-Header durch die nachträgliche Modifikation unbrauchbar gemacht wurde oder ob er dadurch eher dem PE-Header des ungepackten Programms entspricht.

  Als Test kann der gefundene OEP mit dem im Header spezifizierten OEP verglichen werden. Auch können die angegebenen Strukturinformationen mit der Struktur der tatsächlichen Daten verglichen werden. Werden hierbei keine Unterschiede festgestellt, kann der gefundene PE-Header nicht als ungültig klassifiziert werden. Dennoch gilt er hierdurch nicht automatisch als gültig.

Der PE-Header könnte zur Erschwerung der Rekonstruktion des Programms auf eine Importtabelle verweisen, die nicht der vom Programm benötigten Importtabelle entspricht. Dies ist möglich, da diese Information nur beim Ladevorgang durch das Betriebssystem relevant ist. Es ist nicht generell feststellbar, ob die Importsektion auf die verwiesen wird, der vom Programm benötigten Importsektion entspricht. Diese braucht entsprechend Abschnitt 2.3.3 auch nicht vorhanden zu sein.

- PE-Header ist nicht vorhanden.

  Das in den Speicher geschriebene Programm beginnt möglicherweise direkt mit der Codesektion. Für davorliegende Adressen wurde kein Speicher vom Betriebssystem angefordert. In diesem Fall ist kein PE-Header vorhanden, was einfach anhand der PE-Header-Signatur überprüft werden kann. Programme können, müssen aber nicht nach Programmstart auch ohne den PE-Header lauffähig sein.

Sofern ein dem entpackten Programm zuzuordnender PE-Header im Speicher vorhanden ist, kann nicht mit Sicherheit bestimmt werden, ob das entpackte Programm unter Verwendung des im Speicher gefundenen PE-Headers korrekt ausgeführt werden wird. Daher kann dieser auch nicht generell verwendet werden.

4.7.1.3 Erstellung eines neuen PE-Headers

Um die zuvor aufgeführten Unsicherheiten bei einer Übernahme eines bestehenden PE-Headers zu vermeiden, wird ein komplett neuer PE-Header generiert. Hierdurch wird sichergestellt, dass das rekonstruierte Programm korrekt geladen und im Normalfall auch korrekt ausgeführt werden kann. Falls allerdings, wie schon in Abschnitt 2.3.3 beschrieben, Nutzdaten im PE-Header untergebracht sind, wird sich das rekonstruierte Programm wahrscheinlich nicht wie das ungepackte Originalprogramm verhalten.

Die für den PE-Header benötigten Angaben können direkt aus dem Emulatorzustand übernommen oder auf allgemein verwendbare Standardwerte gesetzt werden.

Durch eine Analyse der mit 0 gefüllten Bereiche am Ende von Speicherseiten, also durch Auffinden von Sektionslücken, kann ggf. die Sektionstruktur des entpackten Programms rekonstruiert werden. Falls keine Sektionen bestimmt werden können, reicht es normalerweise auch aus, alle Daten in eine einzige Sektion zu schreiben.

4.7.2 Exakte Bestimmung der Speicherbereiche des entpackten Programms

Grenzen, zwischen denen sich das entpackte Programm befinden muss, lassen sich wie schon im Einleitungstext (Abschnitt 4.7) beschrieben finden. In diesem Bereich können aber auch noch Code und Daten der Entpackroutine enthalten sein. Wenn diese identifiziert werden, können sie bei der Rekonstruktion ausgeschlossen werden, wodurch sich die Größe der rekonstruierten Datei der des ungepackten Programms annähert.

Hierbei dürfen aber keinesfalls Bereiche des entpackten Programms ausgeschlossen werden. Während zusätzlich in die Datei geschriebene Daten die Erkennung normalerweise[34] nicht negativ beeinflussen, können herausgelöschte Bereiche eine Erkennung verhindern und der Funktionalität des Programms schaden. Dies spricht dafür, keine Bereiche von der Rekonstruktion auszuschließen.

Allerdings ist es im Fall einer mehrfach gepackten Datei für die weitere Abschätzung relevanter Speicherbereiche (siehe 4.5.2) besser, möglichst alle nicht vom entpackten Programm belegten, also bei einem weiteren Entpackdurchgang auch nicht mehr verwendeten Bereiche auszuschließen.

Anhand der zur Laufzeit aufgezeichneten Informationen (siehe 4.5.3) kann der Speicherbereich des entpackten Programms möglicherweise eingegrenzt werden.

Speicherbereiche, die entsprechend Abschnitt 4.5.3 als verbraucht gelten und seit der Verwendung nicht überschrieben wurden, werden der Entpackroutine zugeordnet. Da die Heuristik zur Bestimmung des Verbrauchs gelesener Daten kleinere Ungenauigkeiten aufweisen kann, dürfen hier nur ausreichend große verbrauchte Blöcke betrachtet werden.

Entsprechend Abschnitt 2.3.3 können sich die Entpackroutine und ihre Daten, wenn der OEP erreicht ist, nur vor oder hinter dem entpackten Programm sowie in dessen Sektionslücken befinden. Falls mindestens der Anfangsbereich einer Speicherseite der Entpackroutine zugeordnet wurde, so wird die komplette Speicherseite der Entpackroutine zugeordnet. Es kann sich hier also nicht um Teile der Entpackroutine in einer Sektionslücke handeln, da diese Sektionslücken am Ende einer Speicherseite zu finden sind.

Allerdings ist noch nicht bekannt, ob sich diese Speicherseite vor oder hinter dem entpackten Programm befindet. Für die Bestimmung wird der gefundene OEP verwendet. Liegt der OEP hinter der Speicherseite, können alle vor der Speicherseite liegenden Bereiche der Entpackroutine zugeordnet werden, da das entpackte Programm keine Lücken enthält, die größer als eine Sektion sind. Liegt der OEP vor der Speicherseite, können entsprechend alle folgenden Speicherseiten der Entpackroutine zugeordnet werden.

Falls kein Anfangsbereich einer Speicherseite der Entpackroutine zugeordnet werden kann, ist es schwer, hier eine weitere Aussage über die Speicherbelegung zu machen. Die Entpackroutine könnte sich, unabhängig davon, ob der Anfang der Speicherseite zuvor beschrieben wurde oder nicht, in einer Sektionslücke befinden.

Zuvor von der Entpackroutine ausgeführter Code kann anschließend mit dem entpackten Programm überschrieben werden. Daher können verbrauchte, aber nachträglich beschriebene Bereiche nicht eindeutig der Entpackroutine zugeordnet werden. Angenommen, die Entpackroutine (siehe Abbildung 2.4) würde sich vor dem Sprung zum OEP selbst mit 0

[34] Falls die Erkennung auf einem einfachen Hashwert basiert, wird die rekonstruierte Datei nicht mehr erkannt werden.

überschreiben. Durch den Schreibvorgang allein kann nicht entschieden werden, ob ein weiterer Teil des entpackten Programms geschrieben wurde oder ob sich die Entpackroutine selbst überschrieben hat.

Hier wären ggf. genauere Analysen notwendig. Es könnte aufgezeichnet werden, ob Quelldaten für den Schreibvorgang verbraucht wurden. Wurden keine oder nur wenige Quelldaten verbraucht, hat sich die Entpackroutine wahrscheinlich selbst überschrieben. Es könnte aber auch sein, dass es sich um einen Datenbereich des entpackten Programms handelt, der stark komprimiert werden konnte.

Bei näherer Betrachtung der geschriebenen Werte könnte vermutet werden, dass sich über mehr als eine Speicherseite erstreckende Nullen keinen Beitrag zum entpackten Programm darstellen. Schwieriger wäre es, wenn sich die Routine mit einem komplexen Algorithmus selbst verschlüsseln würde. Hier wäre es schwierig festzustellen, ob die geschriebenen Daten eine Relevanz für das entpackte Programm besitzen.

Es kann also möglich sein, den durch das entpackte Programm belegten Speicherbereich exakter oder sogar ganz exakt zu bestimmen. Dies muss aber, wie zuvor beschrieben, nicht in jedem Fall möglich sein.

4.7.3 Passende Importtabelle

Den Tests von [Christodorescu u. a., 2005b, Seite 11] zufolge ist eine intakte Importtabelle für die Erkennung durch Malware-Scanner in der Regel nicht erforderlich. Es ist nicht bekannt, ob die Importtabelle möglicherweise, falls keine passende Signatur vorhanden ist, für eine Erkennung durch Heuristiken benötigt wird.

Für eine bestmögliche Rekonstruktion müsste die Importtabelle des entpackten Programms lokalisiert und anschließend aufbereitet werden. Allerdings braucht die Importtabelle nach Abschnitt 2.3.3 nicht zwingend vorhanden oder intakt zu sein. Trifft dies zu, wird die Aufbereitung dadurch erschwert.

Das Problem wird hier durch einen anderen Ansatz umgangen. Wie in Abschnitt 5.1.5 beschrieben wird, werden für alle geladenen Abhängigkeiten die exakten Adressen einer spezifischen Betriebssystemversion verwendet. Dadurch sind, wenn der OEP korrekt ermittelt wurde, alle Adressen der verwendeten Funktionen bereits korrekt im entpackten Programm eingetragen.

Dadurch braucht dem Programm nur eine einfache Importsektion hinzugefügt zu werden, die die während des Emulationsvorgangs geladenen Abhängigkeiten erneut lädt, aber keinerlei Eintragungen im Programm vornimmt. Das rekonstruierte Programm wird somit sowohl bei einem weiteren Emulatordurchlauf als auch bei nativer Ausführung unter der zuvor spezifizierten Betriebssystemversion regulär ausgeführt werden können. Hierdurch wird eine weitere sowohl manuelle als auch automatische Analyse des entpackten Programms ermöglicht.

4.7.4 Mögliche Probleme bei mehrfach gepackten Dateien

Nachdem der Entpackvorgang abgeschlossen worden ist, könnte sich herausstellen, dass das rekonstruierte Programm auch gepackt ist. Ein bereits gepacktes Programm könnte beispielsweise nachträglich noch verschlüsselt worden sein. Für einen nächsten Entpackschritt sollte die rekonstruierte Datei nicht verwendet werden. Wie in Abschnitt 4.7.1.3 beschrieben, wird bei der Rekonstruktion ein neuer PE-Header generiert. Entpackroutinen, die den Originalheader benötigen, würden dann nicht mehr korrekt funktionieren. Daher muss für einen weiteren Entpackschritt der zuvor gesicherte Emulatorzustand verwendet werden.

Bei einem weiteren Entpackschritt könnte die Abschätzung des Entpackfortschritts ungenauer sein. Nach jedem Entpackschritt können, wie in Abschnitt 4.7.2 erklärt, Teile der vorherigen Entpackroutine nicht vom entpackten Programm unterschieden werden. In dem Fall würden zu viele Speicherbereiche als relevant eingestuft werden. Der Entpackfortschritt würde unterschätzt werden, und der OEP könnte verpasst werden. Hierdurch können weitere Entpackschritte mehr Zeit beanspruchen, als diese bei einer vergleichbaren nur einmal gepackten Datei beanspruchen würden.

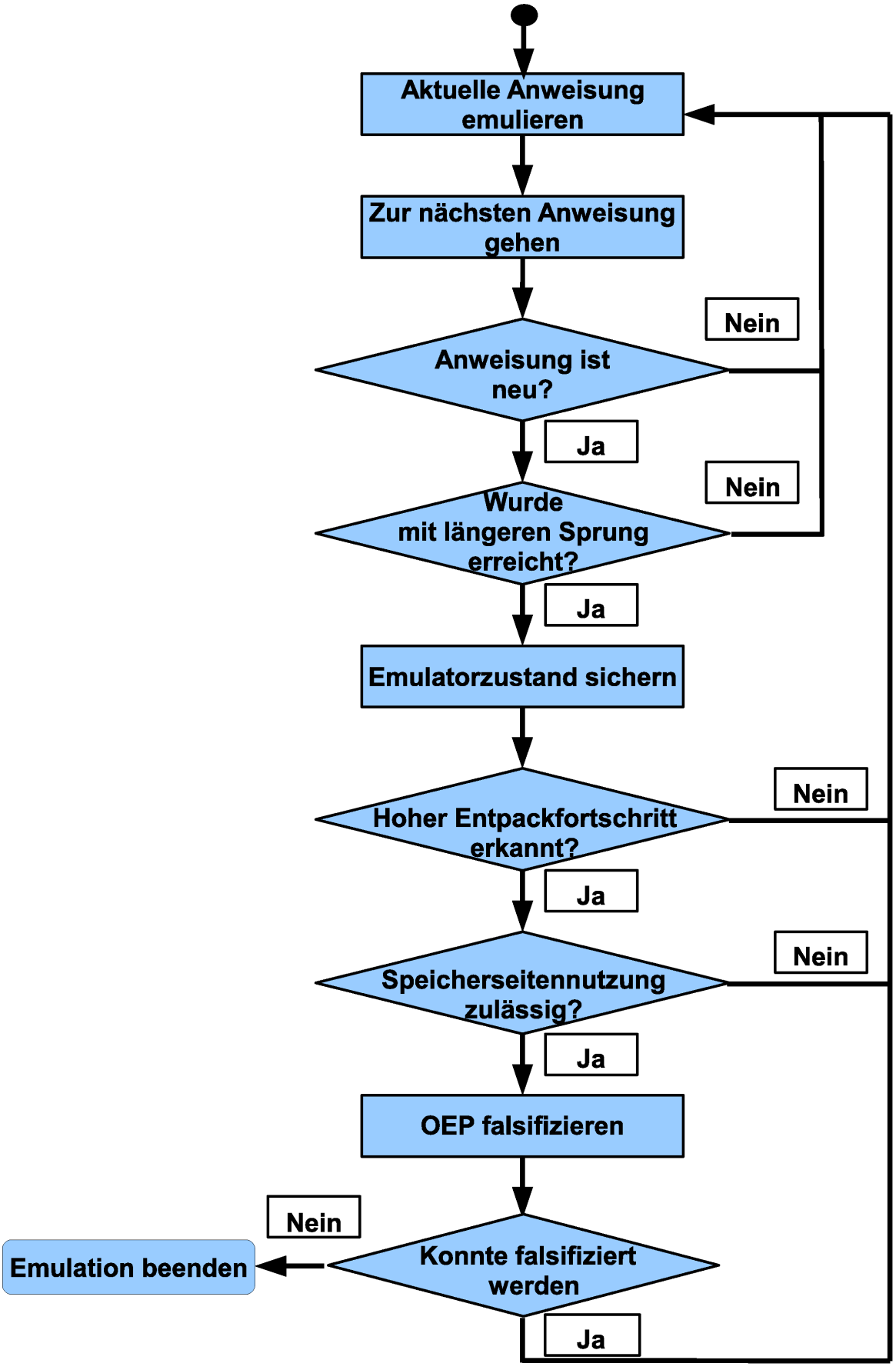

Abbildung 4.1: Flussdiagramm zur OEP-Findung ohne sonstige Abbruchbedingungen

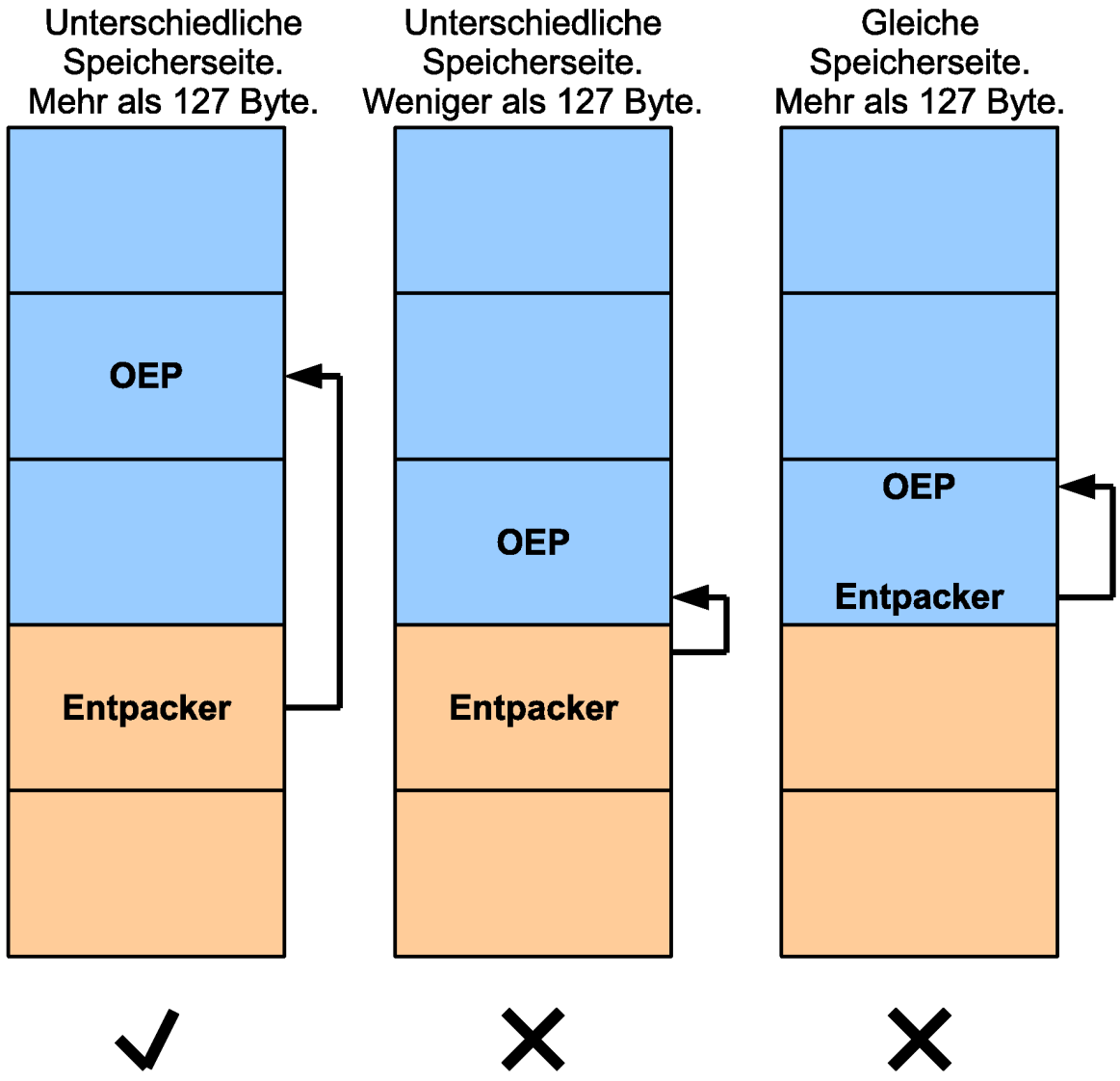

Abbildung 4.2: Entfernungsvorgabe für den Sprung von der Entpackroutine zum OEP

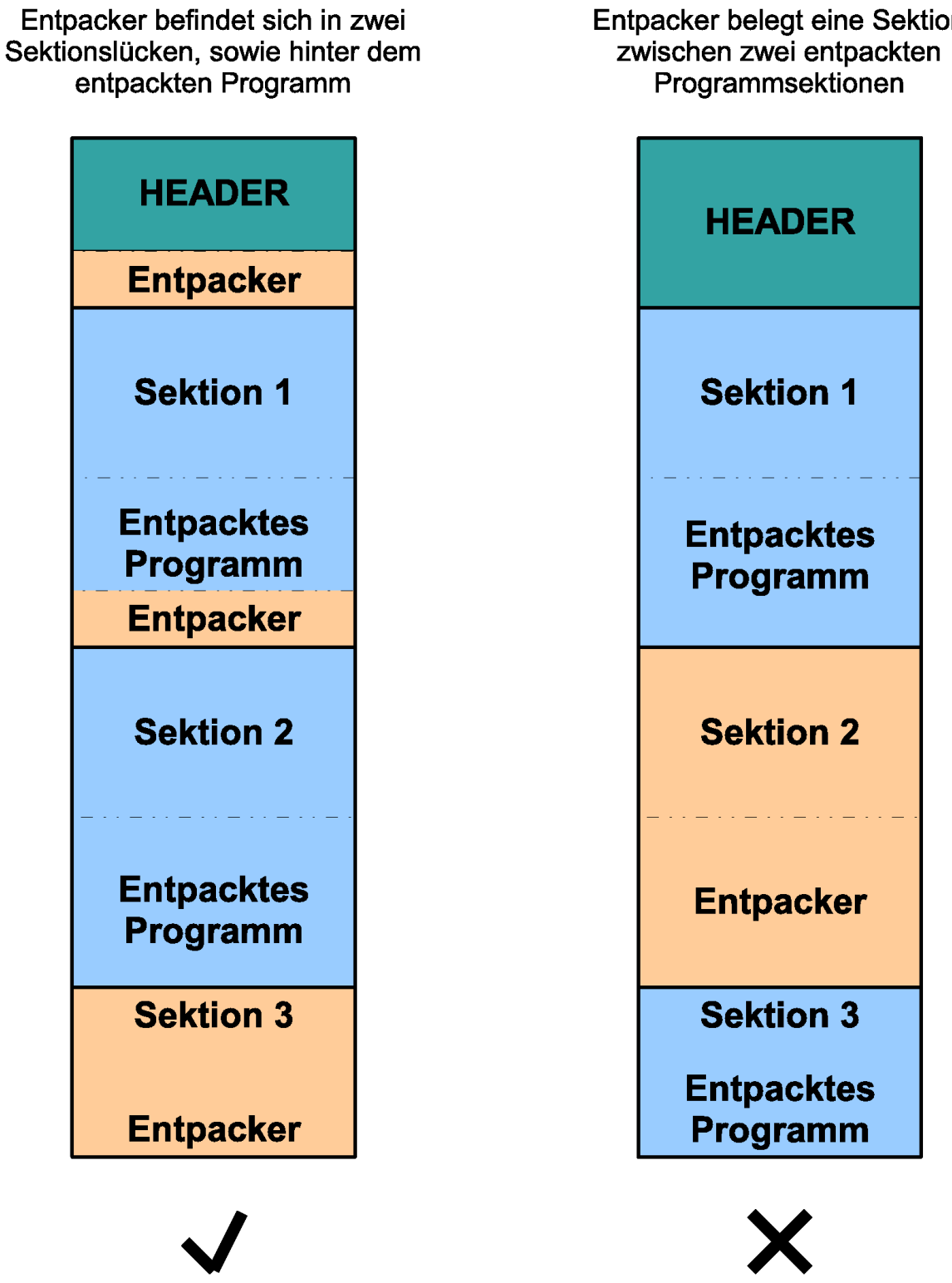

Abbildung 4.3: Mögliche Positionen für den Entpacker

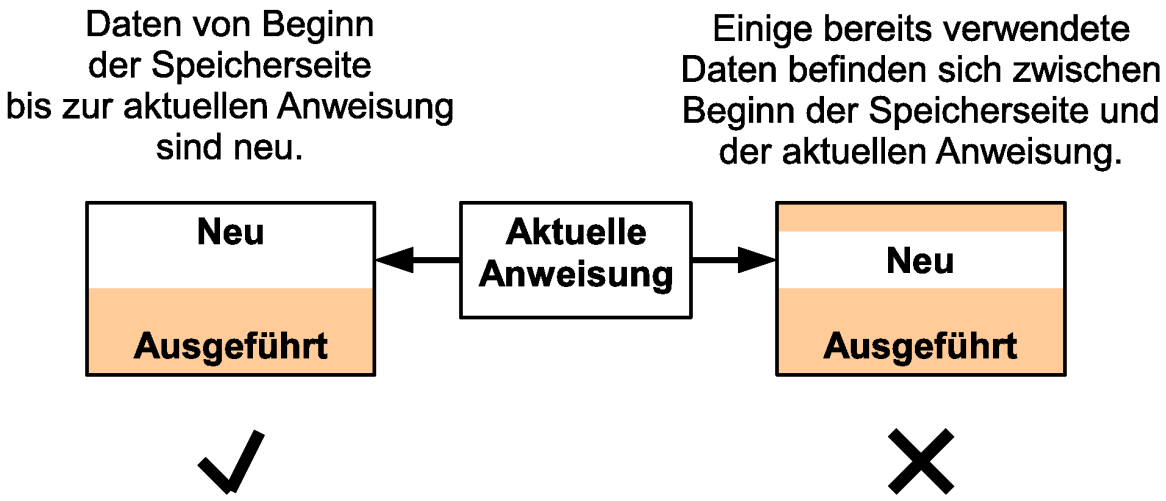

Abbildung 4.4: Neue und ausgeführte Daten in einer Speicherseite

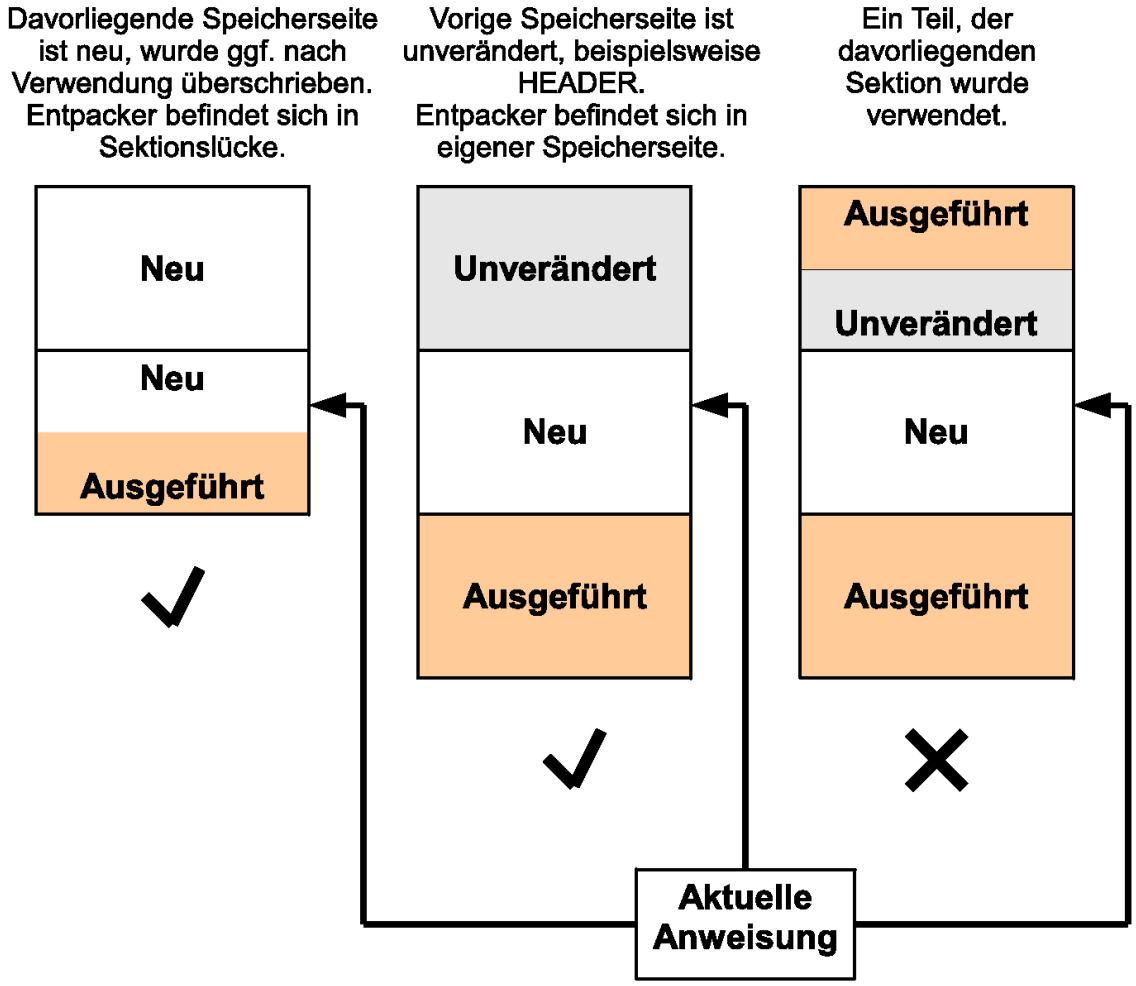

Abbildung 4.5: Zustand der davorliegenden Speicherseite

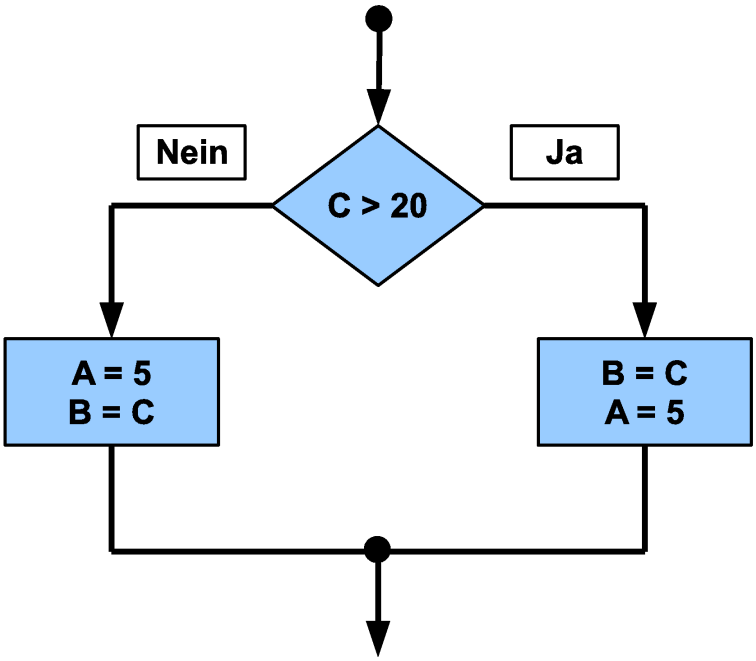

Abbildung 4.6: Junk-Code Beispiel

5 Prototypische Implementierung

Einige bestehende CPU-Emulatoren sind als Quelltext verfügar, beispielsweise QEMU[1] und Bochs[2]. Durch eine Modifikation des Quelltextes und die Einbindung der für das beschriebene Verfahren notwendigen Bestandteile, hätte eine – aufgrund der dynamischen Kompilierung – relativ schnell ablaufende Anwendung konstruiert werden können. Allerdings wurde der Aufwand einer Einarbeitung in den Quelltext, der notwendigen Anpassungen sowie der in C(++) möglicherweise aufwendigen Fehlersuche als zu hoch eingeschätzt.

Aus diesen Gründen und, da eine schnelle Ausführungsgeschwindigkeit für die Funktionalität des Prototyps nicht relevant ist, wurde der Prototyp auf Basis eines neuen Emulators implementiert. Zudem besteht so die Möglichkeit, mehr Detailwissen über die internen Abläufe eines Emulators zu erlangen.

Als Sprache wurde die interpretierte und dynamisch typisierte Sprache Lua[3] gewählt. Diese eignet sich zur schnellen[4] Entwicklung von Software – und demnach auch für Prototypen. Wichtig war hier eine gute Testbarkeit sowie eine einfache Fehlersuche und Implementierung.

Die Sprache kennt keine nativen bitweisen Operationen, welche aber für eine CPU-Emulation erforderlich sind. Entsprechende Operationen können langsam emuliert werden. Allerdings wurde hier das Modul BitLib[5] verwendet, um die Ausführungsgeschwindigkeit nicht zusätzlich zu verlangsamen.

Die Ausführungsgeschwindigkeit des in Lua erstellten Prototyps wurde durch die Verwendung des Just-In-Time-Compilers LuaJIT[6] stark erhöht. Hierzu war keine Anpassung des Quelltextes notwendig.

Aufgrund des großen Umfangs der Konzeption wurden einige Aspekte nicht oder nur teilweise implementiert. Insbesondere die dynamische Kompilierung wurde nicht implementiert, da sie nur zu einer Geschwindigkeitssteigerung geführt aber ansonsten nicht weiter zur Funktionalität beigetragen hätte. Der generelle Aufbau und die ggf. nicht vollständig implementierte Funktionalität wird in den folgenden Abschnitten beschrieben.

---

[1] Siehe http://fabrice.bellard.free.fr/qemu/ 16.07.2007.
[2] Siehe http://bochs.sourceforge.net/ 16.07.2007.
[3] Siehe http://www.lua.org 16.07.2007.
[4] [Siehe ggf. Hirschi, 2007].
[5] Siehe http://luaforge.net/projects/bitlib 16.07.2007.
[6] Siehe http://luajit.org/ 16.07.2007.

5.1 Genereller Aufbau

Es wurden Klassen für verschiedene Funktionalität der Hardware und des Betriebssystems implementiert. Da die exakt benötigte Funktionalität bei Implementierungsbeginn nicht feststand, wurde der Prototyp evolutionär entwickelt.

5.1.1 Controller

Der Controller ist keine Klasse, sondern eine lange Anweisungssequenz, welche alle Schritte vom Laden des zu emulierenden Programms bis zur Rekonstruktion des entpackten Programms, unter Verwendung von Instanzen der vorhandenen Klassen, durchführt. Abgesehen davon, ist hier die Funktionalität zur Abschätzung relevanter Speicherbereiche (siehe Abschnitt 4.5.2), sowie zur nachräglichen OEP-Findung (siehe Abschnitt 4.6.2) enthalten.

Beim Start wird die zu emulierende Datei zunächst in eine Stream-Klasse gekapselt, welche ein Auslesen von 8-, 16- und 32-Bit Werten erlaubt. Diese wird an eine Instanz der PE-Header-Klasse übergeben, welche die benötigten Strukturinformationen ausliest. Mit der Stream- und PE-Klasse wird anschließend der Arbeitsspeicher initialisiert. Die initialisierte Arbeitsspeicherklasse wird an eine neue CPU-Instanz übergeben. Das Programm ist nun geladen und startbereit.

Bevor der Emulationsvorgang gestartet werden kann, müssen zunächst noch die für die Überwachung des Entpackvorgangs (siehe Abschnitt 4.5) wichtigen relevanten Speicherbereiche abgeschätzt werden. Hierzu werden unter Nutzung der Informationen der PE-Header Klasse zunächst alle dem PE-Format zuzuordnenden Speicherbereiche ausgeschlossen. Sektionslücken werden anhand einer Suche fortlaufender Nullen an Sektionsenden erkannt und ebenfalls ausgeschlossen. Durch eine Suche fortlaufender gleicher Bytes sowie möglichen Wörtern, werden weitere nicht relevante Bereiche und bei einem ausreichend hohen Anteil an der Sektion sogar ganze Sektionen ausgeschlossen.

Nachdem die Emulation gestartet und aus einem der angegebenen Gründe (siehe Abschnitt 4.6) wieder beendet worden ist, wird ggf versucht den OEP nachträglich zu bestimmen. Wurde ein wahrscheinlicher OEP gefunden, wird der zugehörige Emulatorzustand wiederhergestellt und das Programm mittels der Arbeitsspeicherklasse rekonstruiert.

5.1.2 PE-Header

Die PE-Header-Klasse liest alle Strukturinformationen des PE-Headers ein. Dazu gehören auch die Informationen über Größe und Position aller Sektionen. Es wird zudem eine für andere Klassen wichtige Methode zur Verfügung gestellt, mit der virtuelle Adressen auf Positionen innerhalb der Datei abgebildet werden können.

Beim Einlesen werden einige Eigenheiten von Windows, wie beispielsweise der Umgang mit nicht an der Größe eines Festplattensektors ausgerichteten Sektionen, berücksichtigt und entsprechend emuliert. Solch spezifisches aber dennoch für den Ladevorgang wichtiges Verhalten, konnte nur anhand von durchgeführten Tests korrekt nachgebildet werden. Insbesondere mit FSG 1.33 oder UPack 3.99 gepackte Dateien besitzen Struktureigenschaften, die schwer mit der PE-Spezifikation [siehe Microsoft, 2006] vereinbar sind.

5.1.3 Arbeitsspeicher

Die Arbeitsspeicher-Klasse erfüllt in dieser Implementierung deutlich mehr Aufgaben als die Abarbeitung von Lese und Schreibzugriffen – nämlich nahezu alle Aufgaben die den Speicher direkt betreffen. Die enthaltene Funktionalität könnte bei einem Refactoring ggf. in verschiedene Klassen unterteilt werden.

Lua verwendet ausschließlich dynamische Arrays. Für diese muss keine feste Größe vorgegeben werden und es kann ohne einen Mehrverbrauch an Speicher an eine beliebige Position geschrieben werden; Dies vereinfacht die Speicherverwaltung. Speicherseiten wurden nicht implementiert, allerdings war eine Emulation der Auswirkung von 4 KB großen Speicherseiten an einigen Stellen notwendig, beispielsweise für Sektionslücken.

Für die Initialisierung erhält die Arbeitsspeicherklasse die Strukturinformationen des PE-Headers sowie die Stream-Klasse des Programms und lädt dessen relevante Bestandteile in die entsprechenden Speicherbereiche. Hierbei werden auch die in der Importabelle spezifizierten Abhängigkeiten geladen, und einige benötigte Systemstrukturen (siehe Abschnitt 4.4) angelegt.

Programme können zur Laufzeit Arbeitsspeicher vom Betriebssystem anfordern. Dies wird auch von der Arbeitsspeicherklasse bearbeitet. Hierbei wird die Arbeitsspeicherverwaltung von Windows möglichst genau simuliert, so dass die Adressen der angelegten Speicherbereiche möglichst genau denen entsprechen, die auch bei der regulären Ausführung zugewiesen worden wären. Hierdurch wird eine spätere Fehlersuche deutlich erleichtert.

Die Klasse enthält neben einer Methode, die ein reguläres Speicherabbild liefert auch eine Methode, die eine entpackte Datei ensprechend Abschnitt 4.7 rekonstruieren kann.

5.1.4 CPU

Die CPU-Klasse enthält alle relevanten Register eines regulären x86-32 Prozessors. Diese werden bei der Initialisierung auf typische Werte gesetzt, die dort beim Start eines Programms vom Betriebssystem gesetzt werden. Die sehr selten von Laufzeitpacker vollständig genutzten Segmentregister wurden nur minimal implementiert, wodurch viel Zeit eingespart und sogar die Ausführungsgeschwindigkeit erhöht werden konnte.

Die Hauptschleife arbeitet wie ein regulärer, interpretierender Emulator durch Einlesen und Dekodieren der aktuellen Anweisung. Es ist zu Testzwecken die Möglichkeit vorgesehen, die aktuell ausgeführte Anweisung sowie alle dadurch in Registern veränderten Werte auszugeben. Hier ist auch ein Abschnitt für die Behandlung von Exceptions und den Aufruf eines verfügbaren Exception Handlers integriert – dies ist eigentlich Betriebssystemfunktionalität.

Als Methoden wurde die Hauptfunktionalität eines Prozessors implementiert – welche im Wesentlichen Rechenoperationen wie Addition oder Division umfasst. Die mit verschiedenen Parametern aufrufbaren Methoden emulieren eine entsprechende Operation durch einen realen Prozessor, setzen die benötigten Flags und geben das Ergebnis zurück. Diese Methoden werden von den konkret implementierten Anweisungen verwendet.

Viele Prozessoranweisungen unterscheiden sich nur minimal, siehe beispielsweise [Intel, 2003, Seite 60]. Um den Implementierungsaufwand zu verringern und die Wartbarkeit zu erhöhen wurde ein Präprozessor erstellt, welcher vor Programmstart zunächst anhand von Vorlagen die konkreten Anweisungen generiert. Durch die Verwendung des Präprozessors werden aus den 67 KB belegenden 2000 Zeilen für die Anweisungen ungefähr 5000 Zeilen generiert, die 142 KB belegen.

Das Anweisungsset enthält nahezu alle im User-Mode zur Verfügung stehenden Anweisungen. Allerdings wurde ein Großteil der Fließkomma-, MMX- und SSE-Anweisungen [siehe Intel, 2003] nicht implementiert, da sie von vielen Laufzeitpackern nicht benötigt werden. Dadurch wurde sehr viel Zeit eingespart. Um weitere Zeit einzusparen wurde keine Multithread-Unterstützung implementiert. Diese wäre aber von der gewählten Architektur her problemlos möglich.

5.1.5 API

Während der Emulation eines Programms können diverse API-Funktionen von diesem benötigt werden. Ein Teil dieser Funktionen ist in der API-Klasse implementiert worden. Es wurden insgesamt 40 Funktionen aus dem Bereich der Grundfunktionalität implementiert. In der Norman SandBox [siehe Norman, 2003] wurde in diesem Bereich die zehnfache Anzahl eingebaut[7]. Allerdings sollen hier auch nicht komplette Programme, sondern nur Entpackroutinen emuliert werden.

Das API wurde anhand der Spezifikationen[8] implementiert. Allerdings sollte auch abweichendes spezifisches Verhalten implementiert werden, da sich Programme darauf verlassen könnten [Surauer, 2002, Seite 4]. Die Programme könnten als Folge möglicherweise nicht korrekt funktionieren, oder den Emulator erkennen können.

Die API-Klasse ist weiterhin für das Laden von Abhängigkeiten zuständig. Diese müssen an die gleichen Speicheradressen wie in einem realen System geladen werden, damit das

[7] Siehe http://vx.7a69ezine.org/re/nsandbox.zip 19.07.2007.
[8] Siehe http://msdn2.microsoft.com/en-us/library/aa286538.aspx 08.07.2007.

rekonstruierte Programm lauffähig sein kann (siehe Abschnitt 4.7.3). Um die entsprechenden Adressen verwenden zu können, wurden zuvor entsprechende Adressdaten aus den vorhandenen Systembibliotheken ausgelesen und in Form einer von der API-Klasse nutzbaren Datenbank gespeichert. Hierdurch kann die API-Klasse auch entscheiden, ob eine zu ladende Abhängigkeit überhaupt vorhanden ist.

So kann auch eine vollständige Exporttabelle generiert werden, so dass es nicht möglich ist, die emulierte Umgebung wie die Norman SandBox anhand einer geringen Anzahl vorhandener Funktionen zu erkennen. Dennoch gibt es ausreichend viele Möglichkeiten diese prototypische Implementierung zu erkennen.

Ein Ladeversuch für nicht existierende Abhängigkeiten könnte als Anti-Emulationsroutine eingesetzt werden, dies wurde jedoch in Tests nicht beobachtet.

Für jede geladene API-Funktion wird eine kleine Anweisungssequenz im Speicher generiert, die unter Verwendung der „SYSENTER“ Anweisung zu der entsprechenden Implementierung der API-Funktion im Emulator springen kann. Hierdurch wird es den Entpackroutinen möglich, den Code der Funktionen – welcher durch eine kleine Anweisungssequenz simuliert wird – beispielsweise auf durch Debugger gesetzte Haltepunkte zu überprüfen. Wird eine nicht implementierte API-Funktion aufgerufen, so wird der Aufruf ignoriert. Auf diese Art kann die Ausführung ggf. normal fortgesetzt werden. Für den Einsatz in der Praxis sollte hier aber eine Lösung entsprechend Abschnitt 4.3.2 implementiert werden.

5.1.6 Exceptions

Es wurde eine Unterstützung für die bei Tests am häufigsten aufgetretenen Exceptions implementiert. Diese durch spezielle Rückgabewerte von Funktionen im Fehlerfall zu realisieren, würde den Emulationsprozess stark verlangsamen. Da Lua über keine Sprunganweisungen verfügt wurden Exceptions mittels Lua-Fehlern emuliert.

Lua Code kann eine Fehlerbedingung melden und hierbei eine beliebige Nachricht übergeben. Gemeldete Fehler werden in der CPU-Klasse abgefangen und ausgewertet. Entspricht die Fehlermeldung einer für Exceptions reservierten Meldung, so wird der Exception Handler der CPU-Klasse ausgeführt, welcher ggf. einen vom emulierten Programm registrierten Exception-Handler anspringt. Dieser kann dann ggf. einen Fortsetzung des Programmablaufs ermöglichen.

5.2 Integration der Heuristiken

Ein Teil der in Abschnitt 4.5 und 4.6 beschriebenen Verfahren wurde wie bereits zuvor beschrieben im Controller untergebracht. Die verbleibende Funktionalität wurden durch Callback-Funktionen in der CPU- und Arbeitsspeicher-Klasse realisiert. Diese werden für

jeden Lese-, Schreib- und Ausführungszugriff aufgerufen. Hierbei wird die betreffende Adresse sowie die Anzahl der bisher ausgeführten Anweisungen übergeben.

Durch diese Aufrufe kann der Entpackfortschritt entsprechend Abschnitt 4.5 bestimmt werden. Wird eine zuvor geschriebene Anweisung ausgeführt, werden die in Abschnitt 4.6 beschriebenen Tests durchgeführt. Durch die Verwendung der Anzahl bisher ausgeführter Anweisungen können hier gesicherte Emulatorzustände später zugeordnet werden. Die in Abschnitt 4.6.1.3 beschriebene Falsifizierung eines OEP wurde nicht implementiert, sondern bei Bedarf manuell durchgeführt und dann an dieser Stelle simuliert.

Das Verfahren zur OEP-Falsifikation wurde nicht implementiert, da es nur sehr selten eingesetzt werden musste um einen korrekten OEP zu erhalten (siehe auch 6.1.5.1). Bei Auftreten entsprechender Fälle wurde das Verfahren mit einem Debugger sowie der rekonstruierten Datei simuliert.

5.3 Umgebungsabhängige Programme

In einigen Fällen fügen Testversionen von Laufzeitpackern einer gepackten Datei Anweisungen für die Anzeige eines Nachrichtenfensters („MessageBox“) hinzu. Diese wird ggf. vor Beginn des Entpackvorgangs angezeigt. Bei einigen Packern muss der Start des entpackten Programms mit „Ja“ bestätigt werden; bei anderen musste „Nein“ gewählt werden, um die Ausführung nicht abzubrechen. Die Rückgabe eines Wertes, welcher weder „Ja“ noch „Nein“ entspricht, erwies sich hier als praktikable Lösung, um eine Fortsetzung des Entpackvorgangs zu veranlassen. Dies sollte allerdings ggf. wie in 4.3 beschrieben gelöst werden.

Einige Entpackroutinen fragen die Prozessorzeit (siehe Abschnitt 4.3.2) ab. Hier wurde testweise nur ein konstanter Wert zurückgegeben. Dies führte erstaunlicherweise nur bei „yoda’s protector“ zu Problemen – nämlich zu einer Endlosschleife.

5.4 Testbarkeit

Ein komplexes System wie ein Emulator muss einfach zu testen sein. Das Vorhandensein von relevanten Fehlern lässt sich in der Regel einfach dadurch feststellen, dass eine Entpackroutine nicht korrekt emuliert wird, das gepackte Programm also nie zurückerhalten wird. Wenn das Vorhandensein eines Fehlers festgestellt wird muss dieser auch einfach zu lokalisieren sein.

Hierzu wird der Debugger OllyDbg[9] verwendet. Mit diesem Debugger werden gepackte Programme im Einzelschrittmodus ausgeführt. Hierbei wird jede ausgeführte Anweisung sowie deren Auswirkung auf den CPU-Zustand protokolliert. Es kann also automatisch eine

[9] Siehe http://www.ollydbg.de/ 16.07.2007.

Auflistung aller ausgeführten Anweisungen bis zum Erreichen des OEP eines gepackten Programms erstellt werden.

Wie in Abschnitt 5.1.4 beschrieben kann auch die CPU-Klasse entsprechende Informationen für das emulierte Programm ausgeben. Da alle Werte so gewählt sind, dass sie sich zwischen realer und emulierter Ausführung fast nie[10] unterscheiden, sind die Ausgaben der CPU-Klasse zu der von OllyDbg meistens identisch.

Anhand von Abweichungen kann dann einfach festgestellt werden an welcher Anweisung die Abweichung auftrat. Hierdurch kann die fehlerhaft implementierte Anweisung gefunden und entsprechend korrigiert werden.

[10] Es war u.a. nicht möglich die Speicherverwaltung von Windows perfekt nachzubilden.

6 Tests

Die Funktionalität des Prototypen wurde an zwei Testsets erprobt. Das erste Testset besteht aus harmlosen, nachträglich mit den in Abschnitt 2.3.2 aufgezählten Laufzeitpackern gepackten Dateien. Das zweite Testset enthält die bereits in Abschnitt 1.3 angeführte automatisch gesammelte Malware.

Anhand der Testsets wurden primär zwei Versuche durchgeführt: zuerst die Rekonstruktion der entpackten Datei durch den Prototypen, danach die Erkennung der rekonstruierten Datei durch den Malware-Scanner.

Als Malware-Scanner wurde ClamAV[1] devel-20070312 verwendet. Die Verwendung eines Open-Source-Malware-Scanners erleichtert die im Rahmen der Tests durchgeführten Modifikationen[2]. Weiterhin können die erhaltenen Ergebnisse aufgrund der vorhandenen Transparenz besser nachvollzogen werden.

Der kommandozeilenbasierte Scanner „clamscan“ wurde in zwei Versionen erstellt: einmal unverändert aus dem ClamAV-Quellcode und einmal mit deaktivierten statischen Entpackroutinen. Durch die Deaktivierung soll die Verwendung nicht unterstützter oder modifizierter Laufzeitpacker simuliert werden. Dies war einfacher, als die entsprechenden gepackten Dateien so zu modifizieren, dass die statischen Entpackroutinen unwirksam werden. Auch konnte so festgestellt werden, welche gepackten Dateien erst nach einem internen Entpackvorgang und welche bereits im Vorwege anhand von Signaturen erkannt wurden.

6.1 Testdateien

Mit den Testdateien sollen Unterschiede bei der Behandlung von Dateien verschiedener Größe und Struktur – sowohl durch die Laufzeitpacker als auch durch den Prototypen – gezeigt werden. Auch war anhand der bekannten Originaldateien eine bessere Verifikation der Ergebnisse möglich.

---

[1] http://www.clamav.net/ 08.07.2007.

[2] Deaktivierung der internen Entpackroutinen und Hinzufügen eigener Signaturen.

6.1.1 Auswahl und Packvorgang

Als Packziel wurden fünf harmlose Programmdateien gewählt, von denen mindestens eine von jedem der verwendeten Laufzeitpacker gepackt werden konnte. Die Dateien bestehen aus einem für diesen Test geschriebenen „Hello World“-Programm, dessen Varianten durch eine unterschiedlich gewählte Struktur sowie Fülldaten 1024 Byte, 10240 Byte und 10752 Byte belegen. Um den mittleren Dateigrößenbereich abzudecken, wurde der Lua[3]-Interpreter mit 192512 Byte sowie base64tool[4] mit 162304 Byte verwendet.

Es wurde sowohl die kleinste der Zieldateien als auch die größte mit den angegebenen Laufzeitpackern gepackt. War dies nicht möglich[5], wurden die von der Dateigröße her ähnlichsten Alternativen aus zuvor angegebenen Zieldateien verwendet.

Wenn ein Laufzeitpacker Einstellungsmöglichkeiten anbot, durch die der Inhalt der gepackten Datei beeinflusst werden konnte, so wurde pro Einstellung[6] oder Kombination von Einstellungsmöglichkeiten eine weitere gepackte Datei erstellt. Zwei Kopien jeder ausgewählten Datei wurden mit den gleichen Einstellungen gepackt. Unterschieden sich die gepackten Dateien voneinander, beispielsweise weil eine polymorphe Entschlüsselungsroutine eingesetzt wurde, so wurden beide Dateien zu dem Testset hinzugefügt.

Das so erstellte Testset besteht aus 591 gepackten Dateien, die keine identischen Daten enthalten. Mit jeder Laufzeitpackerversion wurden 1 bis 48 gepackte Dateien erstellt. Wie aus Abbildung 6.1 hervor geht, wurden 60% der gepackten Dateien durch nur 20% der Laufzeitpacker erzeugt. Die Anzahl der mit jedem Laufzeitpacker erzeugten Dateien unterscheidet sich also stark.

6.1.2 Erstellung und Test der Signaturen

Da die harmlosen Testdateien auch in ungepackter Form nicht von ClamAV erkannt wurden, mussten entsprechende Signaturen für die fünf Testdateien erstellt werden.

ClamAV bietet nach [Cla, 2007] die Möglichkeit, MD5-Hashwerte der Datei oder einer Dateisektion sowie selbst spezifizierte Hexadezimalausdrücke als Signatur zu verwenden. Wie in Abschnitt 4.7 erklärt, ist es sehr unwahrscheinlich, dass die rekonstruierte Datei mit der ungepackten Datei Byte-identisch sein wird. Daher sind MD5-Hashwerte hier nicht als Signaturen anwendbar.

---

[3] http://www.lua.org/ 08.07.2007.

[4] http://www.bunkus.org/videotools/mkvtoolnix/ 08.07.2007.

[5] Der Laufzeitpacker konnte die Datei nicht verarbeiten, wurde durch einen Programmfehler beendet, oder die gepackte Datei war nicht korrekt ausführbar.

[6] Bei von ACProtect 2.0 mit der „OEP obfuscation“-Option erzeugten Dateien werden die Daten im Bereich des EPs der entpackten Datei nicht korrekt wiederhergestellt. Dies entspricht nicht den in Abschnitt 2.3.3 angegebenen Bedingungen. Die Option wurde daher nicht verwendet.

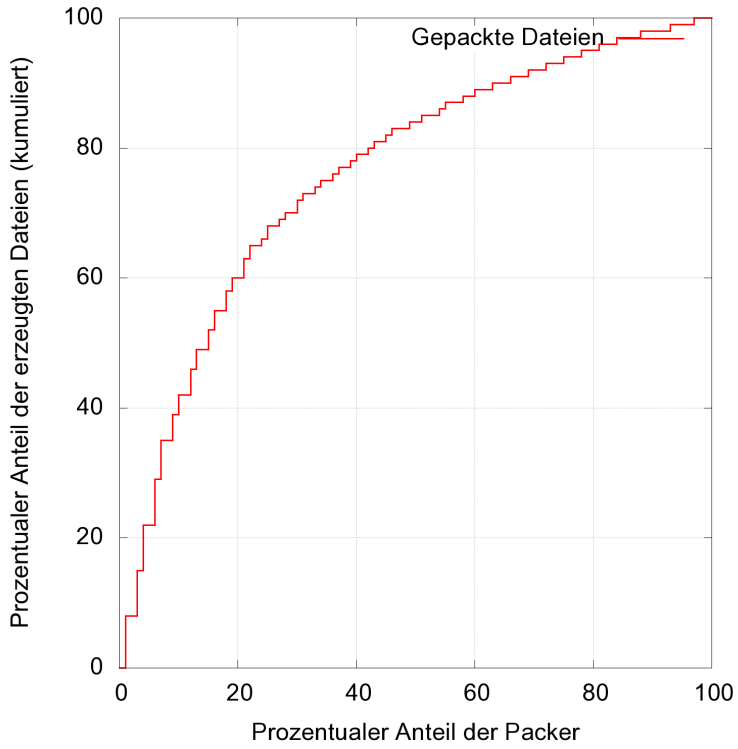

Abbildung 6.1: Anteil der Laufzeitpacker gegenüber dem Anteil der erstellten Dateien

Die Verwendung speziell auf die Testdateien abgestimmter Signaturen im Hexadezimalformat, die leichte Ähnlichkeiten zu regulären Ausdrücken besitzen, ermöglicht eine zuverlässige Erkennung der korrekt entpackten Datei. In den Tests von [Christodorescu u. a., 2005b, Seite 7] wurden einige entpackte Programme nicht erkannt, bis der Einstiegspunkt korrekt eingetragen wurde. Demnach wurden in einigen Fällen EP-basierte Signaturen verwendet. Diese Signaturen erfordern neben ausreichend entpackten Daten auch einen korrekt bestimmten OEP. Es wird also eine Signatur in relativer Entfernung zum EP gesucht.

Pro Testdatei wurden zwei nahezu identische Signaturen erstellt. Der Unterschied besteht nur darin, dass die erste Signatur EP-basiert ist und die zweite nicht. Hiermit soll die korrekte OEP-Erkennung verifiziert werden. Für den Fall, dass der OEP nicht korrekt erkannt wurde, wird noch die nicht-EP-basierte Signatur erkannt werden. Wurde ein Programm nicht hinreichend korrekt entpackt, wird keine Signatur gefunden werden. Die Signaturen wurden umfassender[7] erstellt, als dies für reguläre Malware gemacht wird. Hierdurch soll gesichert werden, dass das rekonstruierte Programm auch das vollständig entpackte Originalprogramm enthält.

Diese Signatur wurde für die 10240-Byte-Version des „Hello World“-Programms erstellt: `??00100000CC{4096}586A0068{100-}6F6F6F6F{4092}556E7061636B6564`.

[7] Vergleiche mit Signaturdatenbank von ClamAV.

Wird diese EP-basiert verwendet, bedeutet dies, dass das Byte am Einstiegspunkt beliebig[8] sein kann. Danach müssen die durch einen zweistelligen Hexadezimalwert repräsentierten Bytes `00 10 00 00 CC` folgen. Nach exakt 4096 weiteren beliebigen Bytes muss die Bytefolge `58 6A 00 68` zu finden sein. Mit mindestens 100 Byte Abstand folgt der nächste Signaturteil. Alle erstellten Signaturen sind im Anhang in Tabelle A.2 aufgelistet.

Die korrekte Erkennung der ungepackten Testdateien wurde mit clamscan unter Verwendung der erstellten Signaturen verifiziert. Die jeweilige EP-basierte Signatur wurde in allen fünf Dateien erkannt.

6.1.3 Überprüfung der gepackten Dateien

Um die initiale Erkennung zu testen, wurden die 591 gepackten Dateien entsprechend überprüft. Hierbei wurden die in Tabelle 6.1 genannten Dateien anhand der zuvor erstellten Signaturen erkannt. Die EP-Spalte ist markiert, wenn die Datei anhand der EP-basierten Signatur erkannt werden konnte. Der Zusatz „Nur“ wird verwendet, wenn noch andere mit dem Packer gepackte Dateien im Testset enthalten sind, aber nicht erkannt wurden.

Packer	Datei	EP
ACProtect 2.0	Nur Hello World 1024 Byte	
ASPack 2.12	Nur Hello World 1024 Byte	
ASProtect 1.23 rc4	Nur Hello World 1024 Byte	
FSG 1.33 und 2.0 (nicht aber 1.2 und 1.31)	Alle Dateien	X
MEW 11	Alle Dateien	X
Upack 3.99 (nicht aber 0.25b)	Alle Dateien	
UPX 1.20, 1.91 beta, 2.02 (nicht aber 2.92b)	Nur Lua	X
UPX-Scrambler	Nur Lua	X
WWPack 1.20	Lua	X

Tabelle 6.1: Mit clamscan erkannte gepackte Testdateien

Es wurden von 12 der 67 Laufzeitpacker gepackte Testdateien mindestens teilweise erkannt. Nach Wiederholung des Tests mit deaktivierten internen Entpackroutinen konnte nur noch die mit ACProtect 2.0, ASProtect 1.23 rc4 und ASPack 2.12 gepackten Hello World-Dateien erkannt.

Dies bedeutet, dass 9 Laufzeitpacker von den internen Entpackroutinen zumindest teilweise unterstützt werden und die Signatur des Hello World-Programms in den zuvor genannten gepackten Dateien direkt enthalten ist.

[8] Wurde das Byte auf den korrekten Wert gesetzt, wurde das geprüfte Programm nicht erkannt. Hierbei handelt es sich anscheinend um einen Fehler.

6.1.3.1 Analyse der unerwarteten Ergebnisse

Die Frage, warum die Signatur des Hello World-Programms in den von drei Laufzeitpackern erstellten Dateien gefunden wurde, muss beantwortet werden, bevor weitere Tests durchgeführt werden können.

Bei einer Analyse des mit ACProtect 2.0 gepackten Hello World-Programms wurde erkannt, dass dieses im Klartext an der richtigen Stelle in der ersten Sektion enthalten ist. Zur Laufzeit wird es nicht generiert, sondern nachträglich von dem hinzugefügten Code angesprungen. Da das Programm nicht zur Laufzeit generiert wird, werden die entsprechenden Dateien aus dem Testset ausgeschlossen. Dies trifft nicht auf die anderen mit ACProtect 2.0 gepackten Dateien zu. Es verbleiben noch 587 Dateien im Testset.

Bei den mit ASPack 2.12 und ASProtect 1.23 rc4 gepackten Dateien findet sich die Signatur im Header, nicht aber in einer der Sektionen. Dies liegt daran, dass der Header der ungepackten Datei, unabhängig von seiner tatsächlichen Größe, mit mindestens 1024 Byte in die gepackte Datei übernommen wird. Der Header wird allerdings nachträglich an die neue Struktur der Datei angepasst. Da die kleinste Testdatei exakt 1024 Byte belegt, wird sie also komplett in die gepackte Datei übernommen. Dennoch wird das Programm beim Entpackvorgang an der richtigen Stelle generiert und ausgeführt. Bei der Rekonstruktion von Dateien wird der Header wie in Abschnitt 4.7.1.3 angegeben ersetzt. Daher kann die im ursprünglichen Header vorhandene Signatur nicht zu einer positiven Erkennung einer nicht vollständig entpackten Datei führen. Die entsprechenden Dateien können also im Testset verbleiben.

Da das Hello World-Programm zwar in den gepackten Dateien im Klartext enthalten ist, nicht aber direkt bei Programmstart ausgeführt wurde, konnten die entsprechenden Dateien nur anhand der nicht-EP-basierten Signatur erkannt werden.

6.1.4 Anwendung des Prototypen und Prüfung der Ergebnisse

Der Prototyp wurde auf das 587 Dateien umfassende Testset angewendet. Wurde ein wahrscheinlicher OEP gefunden, wurde das rekonstruierte Programm als neue Datei gespeichert. Dies war bei 242, also bei 41%, der gepackten Dateien der Fall. Die Anzahl der entpackten Dateien ist entsprechend der in Abbildung 6.1 dargestellten ungleichmäßigen Verteilung nicht repräsentativ für die Anzahl der unterstützten Laufzeitpacker. Daher wird das Ergebnis auf der Basis der verwendeten Packer und nicht auf der Basis der Dateianzahl ausgewertet. Dies wird in Abbildung 6.2 dargestellt und im folgenden Absatz erläutert.

Alle durch 47 der Laufzeitpacker erzeugten Dateien wurden rekonstruiert. Bei 3 Laufzeitpackern konnte ein Teil der Dateien, abhängig von den verwendeten Einstellungen beim Packvorgang, rekonstruiert werden. Für die verbleibenden 17 Laufzeitpacker konnte kein Ergebnis[9] erzielt werden.

[9] Dies basiert auf fehlender oder fehlerhafter Emulatorfunktionalität. Konnte der vorab bekannte OEP des entpackten Programms nicht durch den reinen Emulationsvorgang erreicht werden, so zählt dies als „kein Ergebnis“. Allerdings kann das Programm bereits, wie in Tests von [Christodorescu u. a., 2005b, Seite 11] bestätigt, entpackt sein, bevor der OEP erreicht wird. Dies war in diesem Test auch bei mindestens zwei Packern der Fall. Es wurde hier aber nicht berücksichtigt, da der OEP nicht erreicht werden konnte.

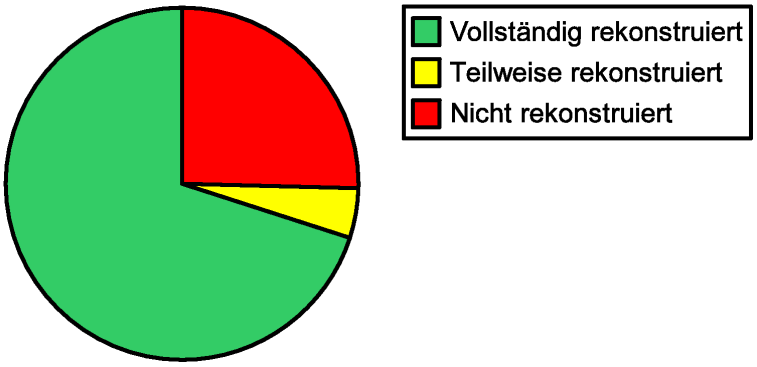

Abbildung 6.2: Resultat des Entpackvorgangs der Testdateien

Die Erstellung einer rekonstruierten Datei, ausgelöst durch einen wahrscheinlich gefundenen OEP, ist aber noch keine Garantie für ein Entpackergebnis, welches eine Erkennung durch einen Malware-Scanner ermöglicht.

Daher wurden die rekonstruierten Dateien mit clamscan überprüft. Hierbei blieb die interne Entpackfunktionalität deaktiviert. So wurde sichergestellt, dass eine Erkennung nur durch den vorherigen Entpackvorgang und nicht durch interne Entpackroutinen ermöglicht wurde.

Bei der Überprüfung durch clamscan wurden alle 242 rekonstruierten Dateien anhand der zuvor für die fünf Testdateien erstellten Signaturen korrekt erkannt. In keinem Fall wurde die nicht-EP-basierte Signatur erkannt. Dies bedeutet, dass der OEP in allen Fällen korrekt gefunden und die Dateien wahrscheinlich korrekt rekonstruiert wurden. Das Ergebnis entspricht also der in Abbildung 6.2 dargestellten Verteilung.

Einige der rekonstruierten Dateien wurden als Stichprobe regulär gestartet. Sie wurden in allen Fällen ohne ein Auftreten von Fehlern ausgeführt. Dies bestätigt die Vermutung, dass die Programme korrekt rekonstruiert wurden. Ein selektiver Binärvergleich der einzelnen Sektionen der rekonstruierten Programme mit den ungepackten Originalprogrammen wurde nicht durchgeführt. Es ist zu unwahrscheinlich, dass alle von den Signaturen abgedeckten Programmteile korrekt entpackt wurden und dennoch Fehler bei anderen Programmteilen aufgetreten wären. Auch ein fehlerfreier Programmablauf wäre unwahrscheinlich, wenn beliebige Stellen des Programms verändert wären.

6.1.5 Detailanalyse der Ergebnisse

Alle rekonstruierten Dateien sind ausreichend genaue Rekonstruktionen der ungepackten Testdateien. Für einige der Laufzeitpacker konnte keine Datei rekonstruiert werden. Rekonstruierte Dateien, die keine ausreichend genaue Rekonstruktion enthielten, sind nicht aufgetreten. Das Zustandekommen dieses Ergebnisses wird hier näher untersucht. Eine tabellarische Detailauflistung der Ergebnisse, auf deren Basis die Auswertung erfolgte, ist im Anhang in Abschnitt A.1.1 aufgeführt.

Daraus, dass keine fehlerhaft rekonstruierten Dateien erzeugt worden sind, ist zu schließen, dass die benötigte Emulatorfunktionalität ausreichend genau implementiert wurde. Allerdings ist eine korrekt implementierte Emulatorfunktionalität keine hinreichende Bedingung für die Erzeugung einer korrekt

rekonstruierten Datei. Hierfür muss zusätzlich der OEP mit ausreichender Genauigkeit bestimmt werden.

6.1.5.1 Rekonstruierte Dateien

Der für die Rekonstruktion der Datei wichtige OEP kann direkt (siehe Abschnitt 4.6.1), nachträglich (siehe Abschnitt 4.6.2) oder auch erst durch Falsifizierung vorheriger OEPs (siehe Abschnitt 4.6.1.3) gefunden werden. Ein inkorrekt bestimmter OEP trat nicht auf. Die Art der OEP-Findung wird für die rekonstruierten, zuvor von 50 verschiedenen Laufzeitpackern gepackten Dateien aufgeschlüsselt. Hierbei wird zwischen kleinen und großen Originaldateien unterschieden.

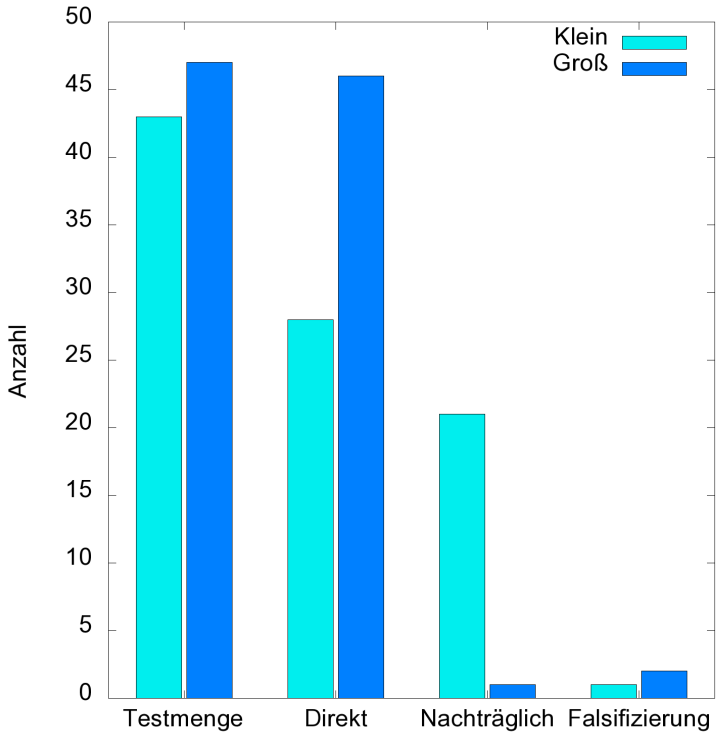

Abbildung 6.3: Art der OEP-Findung

In Abbildung 6.3 ist die Verteilung der OEP-Findung nach Dateigröße gruppiert dargestellt. Die drei Hello World-Programme gelten als „Klein“, die verbleibenden zwei Programme als „Groß“. Die Gesamtzahl der großen und kleinen Dateien pro Packer unterscheidet sich. Dies liegt daran, dass einige Packer keine der kleinen oder keine der großen Dateien packen konnten. Die Gesamtmenge wurde nicht normiert, da der Unterschied für die dargestellte Verteilung nicht relevant ist

Da in der Regel mehrere Dateien pro Packer erstellt wurden, ist es aufgrund der Verwendung unterschiedlicher Optionen hierbei möglich, dass alle drei Methoden der OEP-Findung bei einem Packer genutzt wurden. Bei den kleinen Dateien wurde der OEP bei 6 Packern sowohl direkt als auch nachträglich gefunden. Bei den großen Dateien gab es keine solche Überlappung. Wenn der korrekte OEP erst durch Falsifizierung gefunden wurde, muss dieser dennoch entweder direkt oder

nachträglich gefunden worden sein. Dies ist aus der zuvor genannten Abbildung 6.3 nicht direkt zu entnehmen. Bei den großen Dateien wurde der OEP nach vorheriger Falsifizierung in jedem Fall direkt gefunden. Im Fall der kleinen Dateien geschah dies erst nachträglich.

Auffallend an dem in der Abbildung dargestellten Verhältnis ist, dass der OEP bei den großen Dateien fast immer direkt und bei den kleinen Dateien sehr oft erst nachträglich gefunden wurde. Daraus ist abzuleiten, dass die angewendeten Heuristiken bei kleinen Dateien weniger genau sind. Hierfür gibt es zwei mögliche Erklärungen. Einerseits kann der OEP zunächst verpasst werden, wenn eine Speicherseite der Entpackroutine direkt vor der Speicherseite des OEP liegt (siehe Abschnitt 4.6.1.2). Dies kann bei den kleinen Dateien häufiger zutreffen, da die gepackte Version meistens nur wenige Speicherseiten belegt und der EP des ungepackten Programms immer in der ersten Speicherseite der ersten Sektion liegt. Andererseits können schon kleine ungenutzte Bereiche der Entpackroutine zu einem relevanten Unterschätzen führen.

Einige Entpackroutinen enthalten spezielle Abschnitte, die nur unter den neueren, NT-basierten Windows-Versionen ausgeführt werden. In der ein Windows-NT-basiertes Betriebssystem nachbildenden emulierten Umgebung werden diese Abschnitte dementsprechend nicht ausgeführt werden. Sie wurden als relevant eingestuft, können aber nicht zum Entpackfortschritt beitragen. Hierdurch wird der Entpackfortschritt unterschätzt. Da das gepackte Programm nur sehr wenig Platz belegt, kann die Nichtausführung einiger Codeabschnitte zu einem relevanten Unterschätzen des Entpackfortschritts führen. Auch möglicherweise vorhandene ungenutzt bleibende Datenbereiche können hier einen relevanten Einfluss haben. Wird der Entpackfortschritt stark unterschätzt, wird nicht direkt am OEP angehalten. Er kann also erst nachträglich gefunden werden.

6.1.5.2 Nicht rekonstruierte Dateien

Bei 17 Packern konnte keine einzige der gepackten Dateien rekonstruiert werden. Dies geschah unabhängig von den verwendeten Heuristiken, da der OEP bei keinem Emulationsvorgang erreicht werden konnte. Daraus kann gefolgert werden, dass die implementierte Emulationsfunktionalität für diese Packer nicht ausreichte oder nicht ausreichend korrekt umgesetzt wurde.

Anhand einer schnellen Analyse wurde festgestellt, wo Fehler auftraten oder welche Funktionalität fehlte. Bei der folgenden Auflistung wurde nur das erste angetroffene Problem berücksichtigt. Die Ergebnisse sind nach Anzahl des Auftretens bei den verschiedenen Packern sortiert.

1. Nicht implementierte Funktionen der Windows-API.
2. Unzureichend emulierte Struktur der Windows-DLL-Dateien.
3. Keine Unterstützung für mehrere Prozesse oder Threads.
4. Keine Unterstützung für die Debugg-Register des Prozessors.
5. Keine Unterstützung für Speicherseitenrechte.
6. Unnötig hohe Speichernutzung bei der Sicherung des Emulatorzustands.
7. Nicht implementierte CPU-Anweisungen.

Die hier verwendete Nummerierung entspricht der Nummerierung der in Tabelle A.1 eingetragenen Gründe.

6.1.6 Zusammenfassung

Alle gepackten Dateien, bei denen die vorhandene Emulatorfunktionalität für die Entpackroutine ausreichend war, konnten auch ausreichend genau rekonstruiert werden. Dies beinhaltet die korrekte Bestimmung des OEPs sowie die dadurch bedingte Lauffähigkeit. Für die von 70% der Laufzeitpacker generierten Dateien war die Emulatorfunktionalität ausreichend.

6.2 Gesammelte Malware

Anhand regulärer – teilweise gepackter – Malware soll die Auswirkung des Prototyps auf die Erkennungsrate in der Praxis getestet werden.

Die hier verwendete Malware wurde nicht durch eine automatische Suche im Internet, wie sie vom „AGN-Malware Crawler“[10] durchgeführt wird, gefunden. Stattdessen wurden Chaträume in einem der größten IRC-Netzwerke[11] überwacht. Hierbei wurden, analog zum Verhalten von „IRC Virus Patrol“[12], alle bei der Überwachung der Chaträume registrierten Web-Adressen überprüft.

Wenn beim Aufruf der Web-Adresse – möglicherweise erst nach einigen Weiterleitungen – eine Dateiübertragung gestartet wurde, so ist die Datei heruntergeladen und automatisch überprüft worden. Jede hierbei als Malware identifizierte Datei wurde gespeichert. Über den angegebenen Funktionsumfang von „IRC Virus Patrol“ hinausgehend, wurde auch Malware, die durch Ausnutzung diverser Sicherheitslücken übertragen werden sollte, gespeichert. Falls die Web-Adresse also zu einer präparierten Web-Seite führte, die unter Ausnutzung von Sicherheitslücken Malware übertragen und ausführen sollte, so wurde die zu übertragende Datei unabhängig von einer Identifikation als Malware gespeichert.

6.2.1 Vorbereitung und Überprüfung der Malware-Sammlung

Zunächst wurde jegliche – vermutlich unabsichtlich – mit einem Dateivirus infizierte Malware entfernt, da der Umgang mit entsprechenden Dateien nicht der Zielsetzung dieser Arbeit entspricht. Zudem wurden alle Duplikate entfernt, so dass keine identischen Dateien im Testset vorhanden sind. Die verbleibenden Dateien wurden mit PEiD[13] klassifiziert. Hierbei wurden 27 Laufzeitpacker in insgesamt 53 Versionen erkannt. Identifizierte selbstentpackende Dateiarchive (RAR SFX, ZIP SFX etc.) wurden

[10] [Siehe Freitag, 2000].

[11] Internet-Relay-Chat-Netzwerke. Ein Verbund aus Servern, über die Nutzer Nachrichten in Chaträumen und privaten Konversationen austauschen können. Das IRC-Protokoll ist in RFC 1459 spezifiziert, siehe http://www.irchelp.org/irchelp/text/rfc1459.txt 08.07.2007.

[12] Siehe [Gryaznov, 2006] 08.07.2007.

[13] Programm zur Erkennung von laufzeitgepackten und compilergenerierten Dateien anhand statischer Signaturen, siehe http://peid.has.it/ 08.07.2007.

entfernt, da sie weder den Kriterien eines Laufzeitpackers genügen noch direkt als Malware erkannt werden.

Die verbleibenden 1878 Dateien wurden in 5 Kategorien aufgeteilt:

1. Mit einem Laufzeitpacker gepackt, der nicht in der Liste aus Abschnitt 2.3.2 enthalten ist – und somit auch nicht im vorherigen Testset verwendet wurde.
2. Mit einem bereits im Testset angetroffenen Laufzeitpacker gepackt, von dem mindestens eine Datei entpackt werden konnte. Hierbei wird nicht zwischen den unterschiedlichen Versionen differenziert.
3. Mit einem bereits im Testset angetroffenen Laufzeitpacker gepackt, vom dem keine Datei entpackt werden konnte.
4. Mit einer Hochsprache (Delphi oder C++) erstelltes Programm.
5. Unbekannt. In diesen Programmen wurde weder die Signatur eines Laufzeitpackers noch die eines Compilers gefunden. Auch Dateien, die nicht als gültige[14] PE-Dateien erkannt wurden, sind hier enthalten, da es möglich ist, scheinbar ungültige PE-Dateien zu konstruieren, die dennoch von Windows ausgeführt werden können.

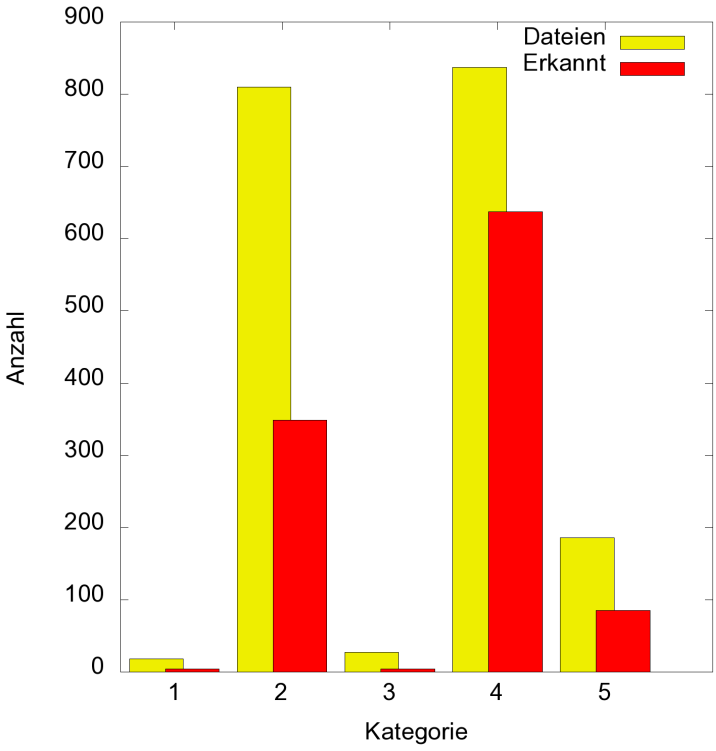

Abbildung 6.4: Erkennung kategorisierter Malware durch ClamAV

[14] In einigen Fällen wurden Dateien anscheinend per FTP im Textmodus übertragen, wodurch diese verändert worden sind.

Wie in Abbildung 6.4 entsprechend den zuvor aufgelisteten Kategorien zu erkennen ist, ist der Anteil der mit einem nicht unterstützten oder ungetesteten Laufzeitpacker erstellten Dateien minimal. Die Anzahl der in einer Hochsprache geschriebenen – und somit vermutlich ungepackten – sowie der mit einem mindestens teilweise unterstützten Packer erstellten Dateien ist nahezu identisch. Nur 10% Dateien konnten von PEiD nicht kategorisiert werden oder sind als ungültige PE-Datei eingestuft worden.

Auffallend ist, dass nur 76% der durch eine Hochsprache erzeugten Dateien von ClamAV als Malware erkannt wurden. In einem deswegen mit einem kommerziellen Malware-Scanner durchgeführten Vergleichstest wurde in dieser Gruppe eine Erkennungsrate von 91% erreicht. Es ist also davon auszugehen, dass ClamAV entsprechende Signaturen fehlen. Dementsprechend wird wahrscheinlich auch ein Teil der korrekt entpackten Malware mangels passender Signaturen nicht erkannt werden können.

Es wurden 44% der gepackten Dateien erkannt. Allerdings sind bei einem Vergleichstest mit deaktivierter Entpackfunktionalität noch 31% der gepackten Dateien erkannt worden. Somit basiert die überwiegende Anzahl der Erkennungen auf Signaturen, die nachträglich für die gepackte Malware erstellt worden sind, oder auf möglicherweise vorhandenen ungepackten Dateiabschnitten. Daher wird ein erfolgreicher Entpackvorgang die Erkennungsrate nur bedingt erhöhen können, da die ungepackte Datei in einigen Fällen bereits erkannt wird.

Durch eine testweise Entfernung der MD5-basierten Signaturen aus der ClamAV-Signaturliste wurde die Erkennungsrate nur um 1% verändert. Demnach müssen fast alle gepackten Dateien durch Signaturen im Hexadezimalformat erkannt worden sein. Da der Aufwand, alle entsprechenden Signaturen, die ausschließlich für gepackte Daten einer Datei erstellt worden sind, zu entfernen, zu hoch ist, wurde die Signaturliste nicht weiter verändert.

6.2.2 Anwendung des Prototyps und Prüfung der Ergebnisse

Durch die Anwendung des Prototyps wurden zwei Ergebnisse erzeugt: einerseits die rekonstruierten Dateien und andererseits die Anzahl von Dateien, bei deren Emulation zuvor geschriebene Anweisungen ausgeführt worden sind. Im Vergleich mit der Gesamtzahl der als gepackt erkannten Dateien lässt sich mit der Anzahl der Dateien, bei denen Codeerzeugung festgestellt wurde, abschätzen, wie ausreichend die Emulatorfunktionalität war.

Der Prototyp wurde auf alle 1878 Dateien angewendet. Hierbei wurde in 896 Fällen Codeerzeugung festgestellt und in 839 Fällen eine rekonstruierte Datei erzeugt. Im zweiten Durchlauf wurde bei 20 der rekonstruierten Dateien erneut Codeerzeugung festgestellt. Für alle 20 Dateien wurde eine neue rekonstruierte Datei erzeugt und die vorherige Datei dadurch ersetzt. Es wurde sogar eine einzelne dreifach gepackte Datei gefunden. Diese konnte aber aufgrund von Einschränkungen des Emulators nicht entpackt werden.

Nur 54% der rekonstruierten Dateien wurden von ClamAV als Malware erkannt. Da die Erkennungsrate für ungepackte Malware vergleichsweise niedrig war, wurde auch hier wieder eine Kontrolle mit einem kommerziellen Malware-Scanner vorgenommen. Dabei wurden immerhin 84% der rekonstruierten Dateien als Malware erkannt. Dies bestätigt die im vorherigen Abschnitt aufgestellte Vermutung, dass die Erkennungsrate trotz entpackter Dateien nicht stark ansteigen wird, da ClamAV-Signaturen für die ungepackten Dateien fehlen.

In Abbildung 6.5 sind die Ergebnisse gruppiert nach den in Abschnitt 6.2.1 aufgezählten Kategorien dargestellt. Zusätzlich zu der bereits in Abbildung 6.4 dargestellten Anzahl der Dateien sowie der direkten Erkennung durch ClamAV wurden noch die Werte für folgende vier Punkte eingetragen:

Code_erzeugt Während des Emulationsvorgangs wurde mindestens eine ausgeführte Anweisung zuvor geschrieben. Dies ist ein typisches Anzeichen für eine gepackte Datei. Falls bei einer gepackten Datei keine Codeerzeugung erkannt wurde, so war die Emulatorfunktionalität für den verwendeten Laufzeitpacker nicht ausreichend[15]. Insgesamt wurde bei 95% der durch PEiD als gepackt eingestuften Dateien Codeerzeugung festgestellt.

Rekonstruiert Es wurde eine rekonstruierte Datei erzeugt. Die erzeugte Datei muss aber nicht zwingend das korrekt entpackte Programm enthalten. In einigen Fällen wurde zwar Codeerzeugung festgestellt, aber keine Datei rekonstruiert. Dies geschieht u. a., wenn aufgrund fehlender Emulatorfunktionalität ein Endzustand erreicht worden ist, bevor der OEP erreicht werden konnte.

Rek_erkannt Die rekonstruierten Dateien wurden mit ClamAV überprüft. Hierbei wurden insgesamt 54% der Dateien als Malware erkannt.

Ges_erkannt Die Auswirkung des Entpackvorgangs auf das Gesamtergebnis wurde überprüft. In einigen Fällen wurden Dateien sowohl in gepackter als auch entpackter Form erkannt. Daher kann die Anzahl erkannter entpackter Dateien nicht direkt zum bisherigen Ergebnis addiert werden. In 45 Fällen wurden Dateien nur in gepackter Form erkannt. Ob die rekonstruierten Dateien aufgrund einer fehlerhaften Rekonstruktion oder aufgrund fehlender Signaturen nicht erkannt wurden, ist unbekannt.

Wie in Abbildung 6.5 zu erkennen ist, konnte für fast alle der mit einem bekannten und möglicherweise unterstützten Laufzeitpacker gepackten Dateien eine rekonstruierte Datei erzeugt werden. Abgesehen von den vermutlich ungepackten Dateien, wurde auch bei einem hohen Anteil der Dateien aller anderen Kategorien Codeerzeugung festgestellt. Diese konnten jedoch oft aufgrund fehlender Emulatorfunktionalität nicht entpackt werden.

Bei 40% der Dateien, die von PEiD keinem Laufzeitpacker zugeordnet werden konnten, wurde Codeerzeugung entdeckt. Auch wurde bei zwei der als nicht gültige PE-Dateien eingestuften Dateien Codeerzeugung entdeckt. Hier hat also entweder der Emulator Dateien verarbeitet, die vom Betriebssystem nicht geladen werden würden, oder PEiDs Erkennung ungültiger PE-Dateien arbeitete fehlerhaft.

Erstaunlicherweise konnte die Erkennungsrate unter Verwendung der entpackten Dateien insgesamt nur leicht verbessert werden, obwohl mehr als die Häflte der rekonstruierten Dateien als Malware erkannt worden sind. Hier gibt es also eine starke Überlappung bei der Erkennung.

Möglicherweise sind gepackte Dateien anhand von Signaturen für die gepackten Daten und entpackte Dateien anhand der Signaturen für ungepackte Malware erkannt worden. Die Erkennungsrate wäre unter diesen Bedingungen trotz erfolgreichen Entpackvorgangs nicht erhöht worden, da die gepackte Form der Malware bereits zuvor bekannt war. Dies wurde anhand der 855 von PEiD als gepackt erkannten Dateien näher untersucht. Von den 375 hiervon direkt erkannten Dateien wurden in 80% der Fälle auch die zugehörigen rekonstruierten Dateien erkannt. Bei der Hälfte dieser gemeinsamen

[15] Mindestens eine gepackte Datei war nur unter Windows 9x korrekt ausführbar, nicht aber unter dem hier emulierten Windows 2000.

Erkennungen wurde jedoch eine unterschiedliche Signatur erkannt, was die Annahme für einen Teil der Dateien bestätigt.

In der rekonstruierten Datei sind implementierungsbedingt sehr oft Teile der ursprünglichen Datei enthalten. Eine genaue Unterteilung, ob die verbleibenden entpackten Dateien anhand eines ungepackt gebliebenen Teils der Originaldatei oder anhand eines übernommenen gepackten Fragments erkannt worden sind, wurde aufgrund des damit verbundenen Aufwands nicht vorgenommen.

Betrachtet man die ungepackten durch eine Hochsprache erzeugten Dateien – auf deren Erkennung das in dieser Arbeit vorgestellte Verfahren keine Auswirkung hat – nicht, so konnte die Erkennungsrate durch den Einsatz des Prototyps lediglich von 42% auf 56% gesteigert werden.

Durch einen Mehrfachvergleich wurde festgestellt, dass 22% der rekonstruierten Dateien Duplikate bereits vorhandener rekonstruierter Dateien sind. Das Testset selbst enthielt keine identischen Dateien. In dem von [Royal u. a., 2006, Seite 8] durchgeführten Test wurden 28% Duplikate generiert. Hierbei ist zu berücksichtigen, dass die rekonstruierten Dateien auch Daten der Originaldatei enthalten können. Eine identische Originaldatei würde also aufgrund der in der Rekonstruktion zusätzlich vorhandenen unterschiedlich verschlüsselten Daten nicht als identisch erkannt werden. Bei einer Sortierung der 839 rekonstruierten Dateien nach Größe wurde festgestellt, dass es 96 verschiedene Dateigrößen gibt. Hierbei enthalten allerdings die 4 größten Gruppen bereits 35% aller Dateien. Bei stichprobenartigen Dateivergleichen innerhalb der Gruppen wurde festgestellt, dass sich einige Dateien oft nur wenig unterscheiden. Typische Unterschiede waren die Adresse, von der aus das betroffene System kontrolliert werden kann, oder die Adresse, an die ausgelesene Passwörter gesendet werden.

6.2.3 Zusammenfassung

Was die Erkennung laufzeitentpacken Verhaltens betrifft, konnten 95% der von PEiD statisch als gepackt erkannten Dateien ebenfalls anhand der Codeerzeugung als wahrscheinlich gepackt erkannt werden. Wird die Erkennung der von PEiD nicht als gepackt erkannten Dateien hinzugezählt, ist das Ergebnis von der Anzahl her nahezu identisch – auch wenn unterschiedliche Dateien als gepackt erkannt worden sind.

Auf die von PEiD als gepackt erkannten Dateien bezogen, konnte das Erkennungsergebnis um 16 Prozentpunkte gesteigert werden – durch Einsatz des Prototyps konnten nun mehr als die Hälfte der Dateien erkannt werden. Eine stärkere Steigerung der Erkennungsrate wurde durch das Vorhandensein von Signaturen für die gepackten Dateien verhindert. Dies wäre bei neu erzeugten gepackten Dateien meistens nicht der Fall. Auch fehlten ClamAV, wie durch einen Vergleichstest mit einem anderen Malware-Scanner verifiziert wurde, entsprechende Signaturen, um die Erkennungsrate durch die entpackten Dateien stärker steigern zu können.

Durch den Test konnte auch gezeigt werden, dass identische Schadprogramme durch den Packvorgang eine nicht mehr identische Form erhalten. Laufzeitpacker werden also verwendet, um eine Erkennung anhand bestehender Signaturen zu verhindern.

6.3 Ergebnis

Die Erzeugung von neuem Code ist, sofern die Emulatorfunktionalität ausreichend ist, innerhalb eines gegebenen Zeitlimits einfach festzustellen. Dies ist eine notwendige Voraussetzung für die Rekonstruktion der ursprünglichen Datei. Durch die Erkennung von Codeerzeugung könnten auch Dateien, bei denen keine Rekonstruktion erfolgen konnte, als potentielles Risiko eingestuft werden. Da dies unabhängig vom gepackten Inhalt geschieht, wären hier Fehl-Erkennungen möglich. Für einen großen Anteil der beiden verwendeten Testsets konnten Dateien rekonstruiert werden. Aufgrund der beim ersten Testset bekannten Originaldateien konnte die Korrektheit aller rekonstruierten Dateien festgestellt werden. Im zweiten Testset war dies aufgrund unbekannter Originaldateien, für die ClamAV teilweise keine passenden Signaturen enthielt, nicht möglich. Die Erkennungsrate durch ClamAV unterschied sich deutlich von der im Test von [Christodorescu u. a., 2005b, Seite 10] angegebenen Rate. Dort wurde aber auch zuvor sichergestellt, dass die ungepackte Originaldatei korrekt erkannt wurde. Nur durch den mit einem kommerziellen Malware-Scanner durchgeführten Test konnte hier eine vergleichbare Erkennungsrate erreicht werden.

Damit das in dieser Arbeit vorgestellte Verfahren zu einer Erkennung zuvor nicht erkannter Dateien führen kann – was anhand des ersten Testsets gezeigt wurde –, müssen einem Malware-Scanner Signaturen für die ungepackten Dateien zur Verfügung stehen. Diese Signaturen müssen so beschaffen sein, dass auch durch den Entpackvorgang möglicherweise leicht veränderte Dateiformen der Malware erkannt werden können.

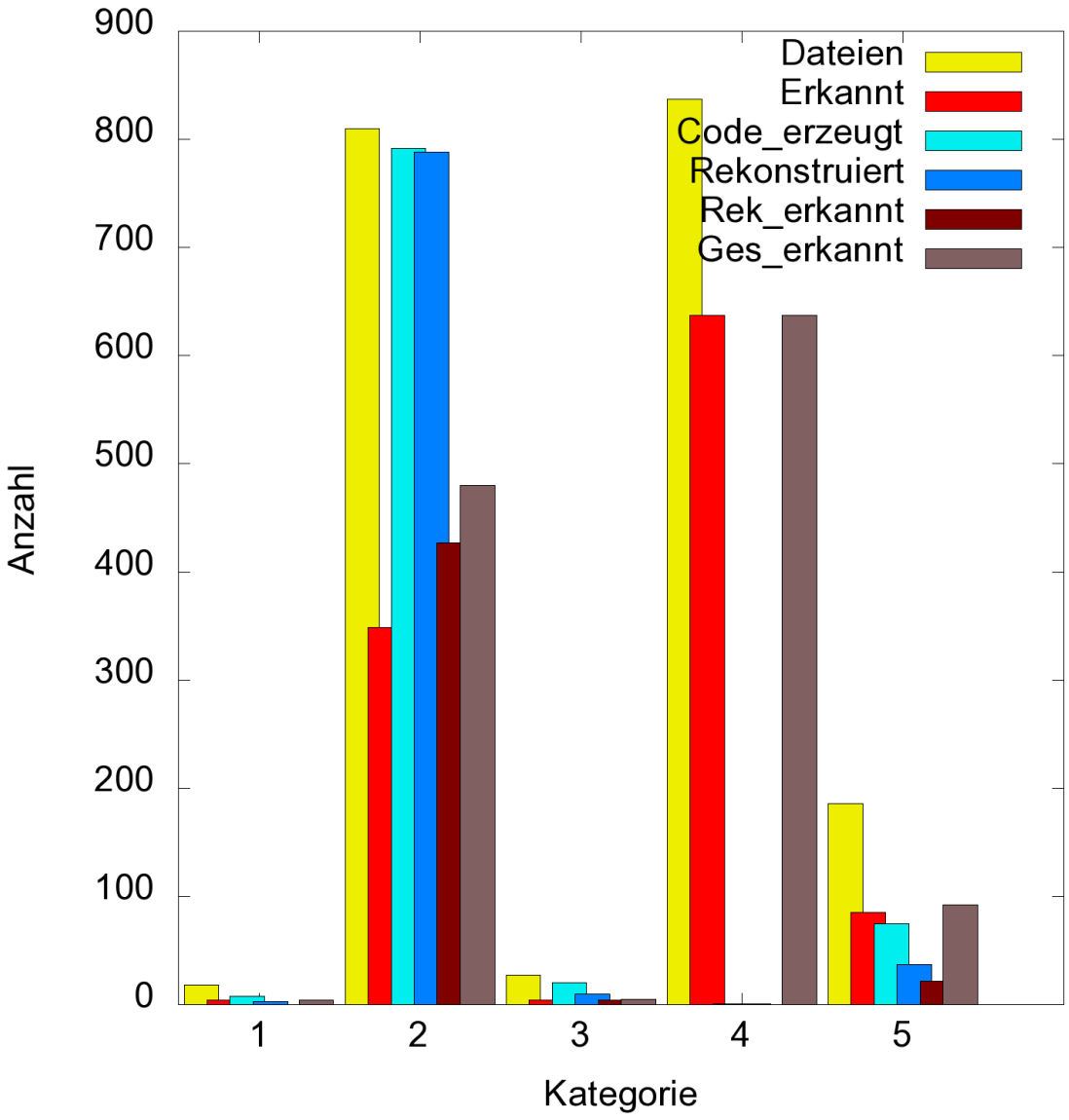

Abbildung 6.5: Ergebnis des Entpackvorgangs und Erkennung durch ClamAV

7 Fazit

In diesem Kapitel werden zunächst die Ergebnisse der Diplomarbeit zusammengefasst. Anschließend werden mögliche weitere Verbesserungen für den Prototyp und das zugrundeliegende Verfahren vorgestellt. Außerdem wird auch auf eventuell auftretende Probleme eingegangen, und es werden ggf. Lösungsansätze hierzu erläutert. Abschließend erfolgt ein Ausblick auf die weitere Verwendung der Lösung außerhalb des gesetzten Ziels sowie eine Abschätzung der möglichen zukünftigen Relevanz generischer Lösungen bezüglich der Erkennung von Malware in der Praxis.

7.1 Zusammenfassung der Arbeit

Gegenstand der Arbeit ist die Ermöglichung der Wiedererkennung bereits bekannter, aber nachträglich vor Erkennung geschützter Malware. Anhand der mit dem implementierten Prototyp durchgeführten Tests konnte gezeigt werden, dass eine Wiedererkennung in sehr vielen Fällen möglich ist. Dennoch kann eine Erkennung aufgrund des Halteproblems sowie dem Malware-Scanner möglicherweise fehlender Signaturen nicht in jedem Fall erreicht werden.

Damit die Erkennungsrate durch Einsatz des hier entwickelten Verfahrens verbessert werden kann, muss der Malware-Scanner in der Lage sein, das ungepackte Programm anhand von Signaturen oder Heuristiken zu erkennen.

Eine nach dem vorgestellten Verfahren implementierte Lösung kann einfach in Malware-Scanner integriert werden. Sogar der Einsatz als eigenständige Anwendung, mit der Dateien direkt vor der Überprüfung durch einen beliebigen Malware-Scanner entpackt werden, ist möglich.

7.2 Mögliche zukünftige Probleme, Lösungsansätze und weitere Verbesserungen

Im Verlauf der Arbeit haben sich einige für die weitere Verwendung und Verbesserung des vorgestellten Verfahrens relevante Punkte ergeben, auf die in den folgenden Abschnitten eingegangen wird.

Außerdem wurden die vorgestellten Heuristiken inzwischen weiter optimiert. So sind sie nun robuster und verkürzen unter bestimmten Bedingungen die benötigte Zeit zur OEP-Findung. Diese Dateilverbesserungen, die sich während der Testphase ergeben haben, konnten nicht mehr in den vorigen Kapiteln erläutert werden. Die vorgestellten Testergebnisse beruhen dennoch ausschließlich auf der Anwendung der in der Arbeit beschriebenen Verfahren.

7.2.1 Nicht komplett wiederhergestellte Programme

Wie in den durchgeführten Tests[1] festgestellt wurde, kann ein Packer, der sich nicht wie ein regulärer[2] Laufzeitpacker verhält, einige Anweisungen am OEP des entpackten Programms entfernen und die Ausführung entsprechend bei der ersten verfügbaren Anweisung beginnen. Das Programm bleibt lauffähig, da die entfernten Anweisungen schon vor dem Sprung zu dem entpackten Programm ausgeführt oder ihre Auswirkungen durch den Einsatz funktional äquivalenter Anweisungen beibehalten worden sind.

Falls die ersten Anweisungen des entpackten Programms den regulären von einem Compiler erzeugten Startanweisungen entsprechen und nicht alle vom Compiler erzeugten Startanweisungen entfernt wurden, könnte dies vom Prototyp erkannt und die entfernten Anweisungen ggf. automatisch wiederhergestellt werden. Auf diese Weise könnten einige von bekannten Compilern erzeugte Programme komplett wiederhergestellt werden, obwohl Anweisungen am OEP zuvor entfernt worden sind.

Es ist in einigen Fällen möglich, nahezu beliebige Teile des Programms zu entfernen und ihre Auswirkungen bei Bedarf zu generieren. Diese und ähnliche Techniken werden noch im Ausblick (Abschnitt 7.3) angesprochen. Um auch lückenhaft rekonstruierte Programme noch erkennen zu können, sollte ein Malware-Scanner mehrere redundante Signaturen für ein Programm enthalten, die nicht EP-basiert sind.

7.2.2 Interpretierte Programme

In Bytecode vorliegende, beispielsweise mit Visual Basic oder .NET erstellte Laufzeitpacker und Entpackroutinen stellen – sofern es hiermit gepackte native Programme gibt – ein Problem dar. Einerseits müsste der Emulator Routinen für den Umgang mit dem entsprechenden Bytecode besitzen, andererseits ist unbekannt, inwieweit die vorgestellten Heuristiken hier anwendbar wären.

Ein Sonderfall ist eine in interpretiertem Code erstellte Entpackroutine, deren Interpreter direkt in die ausführbare Datei mit eingebettet wurde. Diese könnte, sofern der Interpreter keine besonderen CPU-Anweisungen oder API-Funktion benötigt, mit dem vorhandenen Prototyp emuliert werden. Da keine entsprechenden Programme gefunden wurden, konnte die Anwendbarkeit der vorgestellten Heuristiken hierauf nicht getestet werden.

7.2.3 Schutz der emulierten Umgebung

Es würde sehr lange dauern, eine emulierte Umgebung zu erstellen, die von einem darin ausgeführten Programm nicht von einer realen Laufzeitumgebung unterschieden werden kann. Die emulierte Umgebung besitzt auch in der Regel Eigenschaften, die in einem realen System so nicht vorkommen. An diesen könnte die emulierte Umgebung einfach von einem Programm erkannt werden. Damit so eine Erkennung in ein Programm eingebaut werden kann, müssen aber zunächst entsprechende

---

[1] Siehe Abschnitt 6.1.1.

[2] Reguläre Laufzeitpacker erfüllen die in Abschnitt 2.3.3 genannten Bedingungen.

erkennbare Eigenschaften aufgedeckt werden. Hierzu bieten sich zwei Methoden an: einerseits reguläres Reverse Engineering des Emulators, und andererseits möglicherweise einfacheres Auslesen der emulierten Umgebung mit einem darin ablaufenden Programm. Um die Umgebung auslesen zu können, muss das Programm die gelesenen Daten – aus der emulierten Umgebung heraus – an den Nutzer weitergeben können.

Wenn die in dieser Arbeit beschriebene emulierte Umgebung als Vorstufe für einen Malware-Scanner genutzt wird, wäre keine direkte Rückmeldung aus der emulierten Umgebung zu erhalten. Nur das Ergebnis der Prüfung – ob die geprüfte Datei als Malware eingestuft wurde oder nicht – kann durch das zu prüfende Programm beeinflusst werden. Wird das zu prüfende Programm so konstruiert, dass es wahlweise Malware generieren oder sich regulär beenden kann, so können hiermit einzelne Bits aus der emulierten Umgebung heraus übermittelt werden.

Werden die zu prüfenden Programme automatisch generiert, durch den Malware-Scanner überprüft und die Ergebnisse zusammengesetzt, dann lassen sich so beliebige Daten mit entsprechendem Zeitaufwand auslesen. Die Belegung der CPU-Register bei Programmstart, Prozessorzeit, Prozessortyp und Implementation der API-Funktionen könnten beispielsweise eine Erkennung ermöglichen, sofern diese Merkmale nicht oder nur selten in der Praxis vorkommen.

Ein korrektes Auslesen entsprechender Daten durch einen Angreifer kann verhindert werden, indem für jede Datei immer zwei Emulatordurchläufe durchgeführt werden. Bei beiden Durchläufen sollten sich relevante Werte und Speicherbereiche möglichst stark unterscheiden. Alle Bits, die sich bei diesen beiden Durchläufen unterscheiden, werden somit inkorrekt ausgelesen.

Als Gegenmaßnahme kann jedes Bit sowohl auf den Zustand „gesetzt“ als auch „nicht gesetzt“ überprüft werden – es wären also doppelt so viele Testdateien zum Auslesen der Daten notwendig. Hierdurch kann erkannt werden, welche Bereiche sich unterscheiden. Das Ergebnis des Auslesevorgangs wären dann alle in beiden Durchläufen gleichen Bits sowie die Kenntnis aller variablen Bits. Inwieweit sich die so erhaltenen Daten für eine Erkennung der emulierten Umgebung eignen, hängt stark von den Unterschieden zwischen den beiden Emulatordurchläufen ab.

7.2.4 Schutzmechanismen der emulierten Programme

In der Regel brauchen Laufzeitpacker mit Schutzfunktion nur vor den von Angreifern oft genutzten Debuggern oder virtuellen Maschinen zu schützen. Diese Schutzmechanismen sind kein Hindernis für einen umfangreichen Emulator. Es besteht auch wenig Bedarf, Schutzmechanismen gegen Emulation einzubauen, da die meisten Autoren entsprechender Schutzprogramme keinen Grund haben, eine Überprüfung durch einen Malware-Scanner zu verhindern[3].

Nach [Szor, 2005, Kapitel 15.4.4.9] wird allerdings zunehmend emulatorbasiertes Debugging verwendet. Hierbei wird zwar weiterhin mit einem debuggerähnlichen Programm gearbeitet, das Zielprogramm wird jedoch nicht direkt ausgeführt, sondern nur emuliert. Hierdurch werden entsprechende Schutzmechanismen, die den Angreifer am Einsatz eines regulären Debuggers hindern, nahezu wirkungslos.

[3] Eine bekannte Ausnahme: „Morphine“ enthält ab Version 2.0 eine Routine, die im Quelltext als „fake loop against Norton AntiVirus“ bezeichnet wird.

Wenn Emulatoren zukünftig zu einem Problem für Laufzeitpacker mit Schutzfunktion werden sollten, dann ist mit dem Einsatz von Anti-Emulations-Techniken in aktuelleren Versionen der entsprechenden Programme zu rechnen. Dies könnte auch die in Malware-Scannern eingesetzten Emulatoren am korrekten Entpacken eines Zielprogramms hindern. Auch könnte von möglicherweise erkanntem Anti-Emulationscode nicht mehr auf ein wahrscheinlich gepackt enthaltenes Schadprogramm geschlossen werden.

7.2.5 Beschleunigung des Entpackvorgangs

Der implementierte Prototyp selbst ist ein interpretativer Emulator, welcher in einer interpretierten Sprache implementiert wurde. Auf eine dynamische Kompilierung wurde, wie in Abschnitt 5 angegeben verzichtet. Der Prototyp arbeitet also für den praktischen Einsatz in einem Malware-Scanner zu langsam. Ggf. könnte aber die dem im gleichen Abschnitt genannten JIT-Compiler zugrundeliegende Bibliothek für Codeerzeugung verwendet werden, um dynamische Kompilierung durchzuführen. Hierdurch wäre der in einer plattformunabhängigen Sprache implementierte Prototyp zwar immer noch langsamer als ein optimiertes, in einer maschinennahen Sprache geschriebenes Programm, aber möglicherweise schnell genug für den Einsatz in der Praxis.

Viele Packer verwenden bekannte – teilweise leicht modifizierte – Dekompressionsroutinen [Vnuk u. Návrat, 2006, Seite 171]. Beim Vergleich einiger bei der Überwachung des Entpackvorgangs[4] aufgezeichneter Daten wurden anhand der grafischen Darstellung starke Ähnlichkeiten festgestellt. Falls es möglich wäre, anhand der ausgeführten Anweisungen, des Verhaltens oder der bisher angefallenen Quell- und Zieldaten die genaue Routine zu ermitteln, könnte ihre emulierte Ausführung durch die direkte Ausführung der intern vorhandenen Entpackroutine ersetzt werden. Insbesondere große gepackte Programme könnten dadurch deutlich schneller entpackt werden, als es mit dynamischer Kompilierung möglich wäre.

Nach [Tijms, 2000, Seite 2ff] gibt es 5 verschiedene Genauigkeitsstufen für die Emulation. Je niedriger die Genauigkeit ist, desto höher ist die Stufe und die erreichte Ausführungsgeschwindigkeit. Ein rein interpretierender Emulator wird Genauigkeit auf Anweisungsebene (Stufe 3) erreichen. Mit dynamischer Codeerzeugung wird Anweisungsblockgenauigkeit (Stufe 4) erreicht. Die fünfte Stufe ist die Genauigkeit auf semantischer Ebene. Hier kann also eine komplett unterschiedliche Routine verwendet werden, solange sie die gleiche Funktionalität wie die in sich abgeschlossene Originalroutine besitzt. Laut [Tijms, 2000, Seite 3] ist dies aber nur selten automatisiert durchführbar.

7.3 Ausblick

Das im Rahmen dieser Arbeit vorgestellte Verfahren – bzw. der implementierte Prototyp – könnte sich auch noch in anderen verwandten Bereichen als nur dem Entpacken gepackter Malware als nützlich erweisen. Zudem könnte sich als Reaktion auf einen generellen Einsatz der beschriebenen Technik in Malware-Scannern auch das eingesetzte Verfahren, mit dem eine Erkennung bekannter Malware verhindert wird, ändern.

---

[4] Siehe Abschnitt 4.5.

7.3.1 Weitere Verwendung und Auswirkungen

Wie schon in Abschnitt 4.2.3 erläutert, wäre eine statische Entpackroutine für einen Laufzeitpacker deutlich schneller als die Emulation der vorhandenen Entpackroutine. Zur Erhöhung der Geschwindigkeit ist es also sinnvoll, eine entsprechende Routine für häufig anzutreffende Laufzeitpacker zu erstellen. Anstelle einer pro Laufzeitpackerversion neu erstellten Entpackroutine schlagen [Vnuk u. Návrat, 2006] vor, ein in ihrer Arbeit beschriebenes Entpack-Framework zu verwenden. Hiermit kann eine statische Entpackroutine mit deutlich weniger Arbeitsaufwand erstellt werden. Dies basiert auf der Beobachtung, dass in vielen Laufzeitpackern ähnliche Algorithmen verwendet werden.

Bei der Überwachung des Entpackvorgangs[5] fallen viele Daten an, die sich für einen besseren Überblick auch visualisieren lassen. Hieraus könnte möglicherweise auch eine einfache textuelle Beschreibung der Vorgänge in der Entpackroutine generiert werden. Diese würde für die Erstellung einer entsprechenden Entpackroutine nützlich sein. Falls aus dem aufgezeichneten Verhalten der Entpackroutine – wie schon im vorangegangenen Abschnitt beschrieben – auf eine konkret verwendete Dekompressionsroutine geschlossen werden kann, wird die Erstellung einer entsprechenden Entpackroutine mit dem von [Vnuk u. Návrat, 2006] vorgestellten Framework noch weiter vereinfacht.

Das in dieser Arbeit vorgestellte Verfahren kann zwar für viele, aber wahrscheinlich nicht für alle laufzeitgepackten Dateien erfolgreich angewendet werden. Insbesondere nicht, wenn diese nicht den Kriterien[6] eines regulären Laufzeitpackers entsprechen. Um die Erkennung bereits bekannter Malware zu verhindern, wird also ggf. auf Laufzeitpacker ausgewichen, auf die das vorgestellte Verfahren nicht erfolgreich angewendet werden kann. Hierdurch werden die nun unterstützten Laufzeitpacker wahrscheinlich nicht mehr mit Malware verwendet werden. Dementsprechend brauchen hiermit gepackte harmlose Programme – insbesondere da der Laufzeitpacker unterstützt wird, die eigentliche Datei also rekonstruiert werden kann – nicht mehr aufgrund des verwendeten Laufzeitpackers als potentielles Risiko eingestuft zu werden.

7.3.2 Alternativen zu Laufzeitpackern

Laufzeitpacker sind aber nur ein mögliches Mittel, um eine erneute Erkennung bereits bekannter Malware zu verhindern. Primär soll eine Erkennung anhand von Signaturen und Heuristiken verhindert werden. Dies wird durch die von Laufzeitpackern eingesetzte Kompression oder Verschlüsselung des ursprünglichen Programms erreicht. Die Erkennung kann auch auf andere Arten[7] verhindert werden.

Eine Technik, mit der die Wiedererkennung bereits bekannter Malware verhindert werden kann, ist Code Obfuscation. Hierbei wird zwar die Erscheinungsform des Programmcodes, nicht aber seine Funktionalität verändert, und damit bleibt das Programm direkt ausführbar. Einige Packer wenden diese Technik zusätzlich vorab auf das zu packende Programm an. Da die Anwendung dieser Technik auch unabhängig von einem Packvorgang erfolgen kann, handelt es sich hier um ein separates Problem, welches im Folgenden kurz beschrieben wird.

---

[5] Siehe Abschnitt 4.5.

[6] Siehe Abschnitt 2.3.3.

[7] Siehe [Kanzaki u. a., 2003], [Eilam, 2005, Seite 344ff] und [Christodorescu u. a., 2005b].

In dem von [Kanzaki u. a., 2003] vorgestellten Code-Obfuscation-Verfahren wird ein Teil der Anweisungen des Originalprogramms beliebig verändert. Zur Laufzeit werden diese Anweisungen, kurz bevor sie ausgeführt werden, wieder durch ihre ursprüngliche Form ersetzt. Da hierbei zuvor geschriebener Code ausgeführt wird, kann die Verwendung dieser Methode durch eine Laufzeitüberprüfung (Emulation) erkannt werden. Durch Speicherung aller modifizierten und ausgeführten Bereiche des Programms kann der Originalcode ggf. weit genug wiederhergestellt werden, um eine Erkennung anhand von Signaturen zu ermöglichen.

Die von [Eilam, 2005, Seite 344ff] beschriebenen Transformationen benötigen hingegen keine Code-Modifikationen zur Laufzeit. Hierdurch ist eine Erkennung der Transformation sowie eine Wiederherstellung des Originalprogramms schwieriger. Die von [Christodorescu u. a., 2005b] und [Kruegel u. a., 2004] vorgestellten Techniken können sich eignen, um diese Transformationen rückgängig zu machen oder eine größtenteils von der Code Obfuscation unabhängige Repräsentation des Programms zu erstellen. Ein Teil der beim Umgang mit so transformierten Programmen auftretenden Probleme lässt sich nach [Appel, 2002] auf das Halteproblem reduzieren. Wenn das verwendete Code-Obfuscation-Verfahren genau bekannt ist, wird aus einer reversen Transformation jedoch ein entscheidbares Problem [Appel, 2002]. Eine Erkennung von durch Code-Obfuscation-Verfahren veränderter Malware könnte somit ermöglicht werden.

Sollte es nicht möglich sein, eine Repräsentation des Originalprogramms zu erhalten, die für eine Erkennung ausreicht, so könnte das Programm ggf. noch anhand seines Verhaltens als Malware eingestuft werden. [Natvig, 2002] beschreibt hierfür einen entsprechenden Ansatz. Allerdings kann eine Logic Bomb[8] möglicherweise nicht am Verhalten erkannt werden. Hier wäre also eine signaturbasierte Erkennung – nach einem erfolgreichen Entpackvorgang – oder ein anderer Erkennungsansatz notwendig. Zeitabhängiges Verhalten kann mit dem von [Crandall u. a., 2006] entwickelten Verfahren in vielen Fällen erkannt werden.

7.3.3 Schlusswort

Falls ein Malware-Scanner über keine der vorgestellten Techniken, transformierte Schadprogramme zu erkennen, verfügt, können Schadprogramme dennoch anhand einer nachträglich für die transformierte Datei erstellten Signatur – eines einfachen MD5-Hashwerts – von diesem erkannt werden. Dies ist möglich, da Würmer im Normalfall nur auf andere Systeme repliziert werden und präparierte Web-Seiten im Normalfall die gleiche Programmdatei auf alle Zielsysteme kopieren.

Sollte sich dies ändern und jedes Zielsystem eine andere Instanz eines mit einem polymorphen Laufzeitpackers gepackten Schadprogramms erhalten, so wäre diese einfache nachträgliche Erkennung der Gesamtmenge nicht mehr möglich. In diesem Fall müssten Malware-Scanner, sofern keine statische Entpackroutine erstellt wird oder werden kann, generische Techniken anwenden, um eine nennenswerte Erkennungsrate zu erreichen.

[8] Ein Programm, welches erst nach Eintreten von zuvor festgelegten Bedingungen, im einfachsten Fall dem Erreichen eines bestimmten Datums, eine Schadroutine ausführt.

8 Literaturverzeichnis

[Cla 2007] *Creating signatures for ClamAV*. : *Creating signatures for ClamAV*, 02 2007. http://www.clamav.net/doc/latest/signatures.pdf

[Appel 2002] APPEL, Andrew: *Deobfuscation is in NP*. Princeton University. http://www.cs.princeton.edu/~appel/papers/deobfus.pdf. Version: 08 2002

[Bayer 2005] BAYER, Ulrich: *TTAnalyze: A Tool for Analyzing Malware*, Technical University of Vienna, Information Systems Institute, Diplomarbeit, 12 2005. http://www.seclab.tuwien.ac.at/people/ulli/TTAnalyze_A_Tool_for_Analyzing_Malware.pdf

[Bayer u. a. 2006] BAYER, Ulrich ; KRUEGEL, Christopher ; KIRDA, Engin: TTAnalyze: A Tool for Analyzing Malware In Proceedings of the 15th Annual Conference of the European Institute for Computer Antivirus Research, 2006

[Bellard 2005] BELLARD, Fabrice: QEMU, a Fast and Portable Dynamic Translator. In: *ATEC'05: Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, 2005, 41–46

[Brulez 2006] BRULEZ, Nicolas: *Crimeware - Anti-Reverse Engineering Uncovered*. Presentation at the Anti-Phishing Working Group (APWG) Meeting 2006. http://www.websense.com/securitylabs/images/alerts/apwg_crimeware_antireverse.pdf. Version: 06 2006

[Brulez u. Russel 2005] BRULEZ, Nicolas ; RUSSEL, Ryan: *Analyzing Malicious Code*. Presentation at the Reverse Engineering Conference (RECON) 2005. http://www.secure-software-engineering.com/downloads/recon2005/RECON2005_Brulez_Analyzing_Malicious_Code.pdf. Version: 07 2005

[Brunnstein 1999] BRUNNSTEIN, Dr. K.: From AntiVirus to AntiMalware Software and Beyond: Another Approach to the Protection of Customers from Dysfunctional System Behaviour In Proceedings of the 22nd National Information Systems Security Conference, National Information Systems Security Conference, 1999

[Carrera u. Erdélyi 2004] CARRERA, Ero ; ERDÉLYI, Gergely: Digital genome mapping: Advanced binary malware analysis in Proceedings of 15th Virus Bulletin International Conference, 2004

[Christodorescu u. a. 2005a] CHRISTODORESCU, Mihai ; JHA, Somesh ; SESHIA, Sanjit A. ; SONG, Dawn ; BRYANT, Randal E.: Semantics-Aware Malware Detection. In: *Proceedings of the 2005 IEEE Symposium on Security and Privacy (Oakland 2005)*. Oakland, CA, USA : ACM Press, 05 2005. – ISBN 1–58113–820–2, 32–46

[Christodorescu u. a. 2005b] CHRISTODORESCU, Mihai ; KINDER, Johannes ; JHA, Somesh ; KATZENBEISSER, Stefan ; VEITH, Helmut: Malware Normalization / University of Wisconsin, Madison. Version: November 2005. http://www.cs.wisc.edu/wisa/papers/tr1539/tr1539.pdf. Wisconsin, USA, November 2005 (1539). – Forschungsbericht

[Cifuentes 1994] CIFUENTES, Cristina: *Reverse Compilation Techniques*, Queensland University of Technology, School of Computing Science, Diss., 07 1994. http://cse.unl.edu/~jricha/re/documents/x86/decompilation_thesis.pdf

[Cifuentes u. a. 2001] CIFUENTES, Cristina ; WADDINGTON, Trent ; EMMERIK, Mike V.: Computer Security Analysis through Decompilation and High-Level Debugging. In: *Working Conference on Reverse Engineering*, 2001, pp 375–380

[Crandall u. a. 2006] CRANDALL, Jedidiah R. ; WASSERMANN, Gary ; OLIVEIRA, Daniela A. S. ; SU, Zhendong ; WU, S. F. ; CHONG, Frederic T.: Temporal search: detecting hidden malware timebombs with virtual machines. In: *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA : ACM Press, 2006. – ISBN 1–59593–451–0, 25–36

[Eckert 2001] ECKERT, Claudia: *IT-Sicherheit*. Oldenbourg, 2001

[Eggert u. a. 2003] EGGERT, Bodo ; MESSERSCHMIDT, Michel ; SEEDORF, Jan: Klassifikation von bösartiger Software und aktuelle Testergebnisse des Virus Test Centers von AntiMalware-Software unter Linux Linuxtag 2003, 2003

[Eilam 2005] EILAM, Eldad: *Reversing - Secrets of Reverse Engineering*. Wiley Publishing, Inc., 2005 http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0764574817.html

[Ferrie 2006] FERRIE, Peter: Attacks on Virtual Machine Emulators. Auckland, New Zealand, 12 2006

[Freitag 2000] FREITAG, Sönke: *Webbasiertes Auffinden maliziöser Software mit fortschrittlichen heuristischen Verfahren*, Universität Hamburg, Anwendungen der Informatik in Geistes- und Naturwissenschaften, Diplomarbeit, 07 2000. http://agn-www.informatik.uni-hamburg.de/papers/doc/diparb_soenke_freitag.pdf

[Giloi 1984] GILOI, Wolfgang: Die Entwicklung der Rechnerarchitektur von der von-Neumann-Maschine bis zu den Rechnern der „fünften Generation“. In: *Elektronische Rechenanlagen* 26 (1984), Nr. 2, S. pp 55–70

[Graf 2005] GRAF, Tobias: Generic Unpacking In Proceedings of the 2005 Virus Bulletin International Conference, 2005

[Gryaznov 2006] GRYAZNOV, Dmitry: Malware in popular networks In Proceedings of the 2006 Virus Bulletin International Conference, 2006

[Hirschi 2007] HIRSCHI, Ashwin: Travelling light, the Lua way. In: TRATT, Laurence (Hrsg.) ; WUYTS, Roel (Hrsg.): *Rapid Application Development with Dynamically Typed Languages*, IEEE Software, 2007

[Hohensee u. a. 1996] HOHENSEE, Paul ; MYSZEWSKI, Mathew ; REESE, David: Wabi Cpu Emulation Proceedings of the 8th HotChips Symposium, 1996, pp. 47–65

[Horspool u. Marovac 1980] HORSPOOL, R. N. ; MAROVAC, Nenad: An Approach to the Problem of Detranslation of Computer Programs. In: *Compututer Journal* 23 (1980), Nr. 3, S. 223–229

[Intel 2003] INTEL: *IA-32 Intel Architecture Software Developers Manual Volume 2: Instruction Set Reference*, 2003. – Order Number XXX.XXX *Eintragen*

[Johansen 2005] JOHANSEN, Eric: Anti-Virus in the Wild Proceedings of the 2005 International Virus Bulletin Conference, 2005

[Kanzaki u. a. 2003] KANZAKI, Yuichiro ; MONDEN, Akito ; NAKAMURA, Masahide ; MATSUMOTO, Ken ichi: Exploiting Self-Modification Mechanism for Program Protection. In: *Proceedings of the 27th Annual International Computer Software and Applications Conference*, 2003

[Kaspersky u. a. 2003] KASPERSKY, Kris ; TARKOVA, Natalia ; LAING, Julie: *Hacker Disassembling Uncovered*. A-List Publishing, 2003 http://www.alistpublishing.com/html/Programming/book2&book_home. – ISBN 1931769222

[Kephart u. a. 1997] KEPHART, Jeffrey O. ; SORKIN, Gregory B. ; SWIMMER, Morton ; WHITE, Steve R.: Blueprint for a Computer Immune System. In: *Proceedings of the Seventh International Virus Bulletin Conference*, 1997

[Krüger 2003] KRÜGER, Silvio: *Superwürmer - Entdeckung und Gegenmaßnahmen*, Universität Hamburg, Anwendungen der Informatik in Geistes- und Naturwissenschaften, Diplomarbeit, 08 2003. http://agn-www.informatik.uni-hamburg.de/papers/doc/Krueger.pdf

[Kruegel u. a. 2004] KRUEGEL, C. ; ROBERTSON, W. ; VALEUR, F. ; VIGNA, G.: Static disassembly of obfuscated binaries. San Diego, CA, 08 2004, pp 255–270

[Lawton 2002] LAWTON, George: Virus Wars: Fewer Attacks, New Threats. In: *Computer* 35 (2002), Nr. 12, S. 22–24. http://dx.doi.org/http://dx.doi.org/10.1109/MC.2002.1106172. – DOI http://dx.doi.org/10.1109/MC.2002.1106172. – ISSN 0018–9162

[Lawton 1999] LAWTON, Kevin: *Running multiple operating systems concurrently on an IA32 PC using virtualization techniques*. http://denali.cs.washington.edu/relwork/papers/plex86.txt. Version: 11 1999

[Magnusson 1993] MAGNUSSON, Peter: Partial Translation. Version: 10 1993. ftp://ftp.sics.se/pub/SICS-reports/Reports/SICS-T--93-05--SE.ps.Z. European Research Consortium for Informatics and Mathematics at SICS, 10 1993. – Forschungsbericht

[Magnusson u. a. 2002] MAGNUSSON, Peter ; CHRISTENSSON, Magnus ; ESKILSON, Jesper ; FORSGREN, Daniel ; HALLBERG, Gustav ; HOGBERG, Johan ; LARSSON, Fredrik ; MOESTEDT, Andreas ; WERNER, Bengt: Simics: A Full System Simulation Platform. In: *Computer* 35 (2002), 02, Nr. 2, S. pp 50–58. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/2.982916. – DOI http://doi.ieeecomputersociety.org/10.1109/2.982916

[Marx 2004] MARX, Andreas: Outbreak Response Times: Putting AV to the Test In Proceedings of the 2005 Virus Bulletin International Conference, 2004

[Microsoft 2006] MICROSOFT: *Microsoft Portable Executable and Common Object File Format Specification*. v8.0, 05 2006. http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

[Miller 2005] MILLER, Matt: Implementing a Custom X86 Encoder. In: *Uninformed Journal* 5 (2005), 07. http://www.uninformed.org/?v=5&a=3&t=pdf

[Miller 2007] MILLER, Matt: Locreate: An Anagram for Relocate. In: *Uninformed Journal* 6 (2007), 01. http://www.uninformed.org/?v=6&a=3&t=pdf

[Morgenstern u. Brosch 2006] MORGENSTERN, Maik ; BROSCH, Tom: *Runtime Packers: The Hidden Problem?* Black Hat USA. http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf. Version: 07 2006. – Presentation

[Muttik 2000] MUTTIK, Igor: Stripping Down an AV Engine In Proceedings of the 2000 Virus Bulletin Conference, 2000

[Natvig 2002] NATVIG, Kurt: Sandbox II: Internet Proceedings of the 2002 International Virus Bulletin Conference, 2002

[Nerche 2000] NERCHE, Jens: *Virtualisierung eines x86 PC*, Technische Universität Dresden, Diplomarbeit, 09 2000. http://os.inf.tu-dresden.de/papers_ps/nerche-diplom.ps

[Norman 2003] NORMAN: *Norman SandBox Whitepaper*. http://sandbox.norman.no/pdf/03_sandbox%20whitepaper.pdf. Version: 2003

[Ohl 2002] OHL, Bruno: *Der VMware-Ansatz: Virtualisierung von Rechnern*. Technische Universität München, Vortrag. http://www13.in.tum.de/lehre/seminare/WS0203/hauptsem/Vortrag9_VM_Ausarbeitung_Verbessert.pdf. Version: 11 2002

[Popek 1974] POPEK, Goldberg: Formal requirements for virtualizable third generation architectures. In: *Communications of the ACM* 17 (1974), 7

[Probst 2001] PROBST, Mark: Fast Machine-Adaptable Dynamic Binary Translation. In: *SIGARCH Comput. Archit. News* 29 (2001), 09, Nr. 5. http://www.complang.tuwien.ac.at/schani/papers/bintrans.ps.gz

[Quist u. Smith 2006] QUIST, Daniel ; SMITH, Val: *Detecting the Presence of Virtual Machines Using the Local Data Table*. http://www.offensivecomputing.net/?q=filemanager/active&fid=164. Version: 03 2006

[Robin u. Irvine 2000] ROBIN, John ; IRVINE, Cynthia: Analysis of the Intel Pentium's Ability to Support a Secure Virtual Machine Monitor. Denver, CO, 08 2000

[Royal u. a. 2006] ROYAL, Paul ; HALPIN, Mitch ; DAGON, David ; EDMONDS, Robert ; LEE, Wenke: PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In: *ACSAC '06: Proceedings of the 22nd Annual Computer Security Applications Conference on Annual Computer Security Applications Conference*. Washington, DC, USA : IEEE Computer Society, 2006. – ISBN 0–7695–2716–7, S. 289–300

[Russinovich u. Solomon 2004] RUSSINOVICH, Mark E. ; SOLOMON, David A.: *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Redmond, WA, USA : Microsoft Press, 2004 http://www.microsoft.com/mspress/books/6710.aspx. – ISBN 0735619174

[SabreSecurity 2006] SABRESECURITY: *VxClass - Automatische Klassifikation von Malware und Trojanern in „Familien“*. http://www.sabre-security.com/vxclass_rub.pdf. Version: 11 2006. – Deutscher IT Sicherheitspreis, Horst Görtz Stiftung

[Santamarta 2006] SANTAMARTA, Rubén: *Generic Detection and Classification of Polymorphic Malware Using Neural Pattern Recognition*. http://www.reversemode.com/index.php?option=com_remository&Itemid=2&func=fileinfo&id=22. Version: 10 2006

[Schipka 2006] SCHIPKA, Maksym: Prevalence of PE packers in e-mail traffic. Auckland, New Zealand, 12 2006

[Schultz u. a. 2001] SCHULTZ, M. G. ; ESKIN, E. ; ZADOK, E. ; STOLFO, S. J.: Data mining methods for detection of new malicious executables. In: *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*. Washington, DC, USA : IEEE Computer Society, 2001, pp 38–49

[Shaffer 2004] SHAFFER, Joshua: *A Performance Evaluation of Operating System Emulators*. Bucknell University. http://www.eg.bucknell.edu/~perrone/students/JoshShaffer.pdf. Version: 05 2004

[Shealy u. a. 1997] SHEALY, A.R. ; MALLOY, B.A. ; SYKES, D.A.: SIMx86: An extensible simulator for the Intel 80x86 processor family. In: *Simulation Symposium* 00 (1997), 157. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/SIMSYM.1997.586518. – DOI http://doi.ieeecomputersociety.org/10.1109/SIMSYM.1997.586518. – ISSN 1080–241X

[Shirey 2000] SHIREY, R.: *rfc2828: Internet Security Glossary*. http://www.rfc-editor.org/rfc/rfc2828.txt. Version: 5 2000

[Stepan 2005] STEPAN, Adrian: Defeating Polymorphism: Beyond Emulation In Proceedings of the 2005 Virus Bulletin International Conference, 2005, pp 40–48

[Surauer 2002] SURAUER, Christian: *Ansätze zur Virtualisierung von Rechnern*. Technische Universität München, Presentation. http://wwwspies.informatik.tu-muenchen.de/lehre/seminare/SS02/hauptsem/Vortrag1_VirtualisierungFinal.pdf. Version: 04 2002

[Szor 2000] SZOR, Peter: Attacks on Win32 - Part II Proceedings of the 2000 International Virus Bulletin Conference, 2000, pp 47–68

[Szor 2001] SZOR, Peter: Drill Seeker in Proceedings of 2001 Virus Bulletin International Conference, 2001

[Szor 2005] SZOR, Peter: *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, 2005 http://www.awprofessional.com/title/0321304543. – ISBN 0321304543

[Tijms 2000] TIJMS, Arjan: *Binary translation : Classification of emulators*. University Leiden, Leiden Institute for Advanced Computer Science. http://www.liacs.nl/~atijms/bintrans.pdf. Version: 2000

[Veldman 1993] VELDMAN, Frans: Combating Viruses Heuristically. In: *Proceedings of the 3rd International Virus Bulletin Conference*, 1993, S. pp. 67–76

[Vnuk u. Návrat 2006] VNUK, Miroslav ; NÁVRAT, Pavol: Decompression of Run-Time Compressed PE-Files. In: *Studies in Informatics and Control* 15 (2006), 06, Nr. 2. http://www.ici.ro/ici/revista/sic2006_2/art03.html

[Witchel u. Rosenblum 1996] WITCHEL, Emmett ; ROSENBLUM, Mendel: Embra: Fast and Flexible Machine Simulation. In: *Measurement and Modeling of Computer Systems*, 1996, 68-79

A Anhang

A.1 Testdetails

Dieser Abschnitt enthält Details zu den in Kapitel 6 durchgeführten Tests.

A.1.1 Verwendete Laufzeitpacker

Tabelle A.1 enthält alle verwendeten Laufzeitpacker sowie die hiermit in den Tests erzielten Ergebnisse. Die Überschriften der einzelnen Spalten haben folgende Bedeutung:

Packer Name und Version des verwendeten Laufzeitpackers.

SM Mindestens eines der kleineren Programme konnte gepackt werden.

BG Mindestens eines der größeren Programme konnte gepackt werden.

OK Bei mindestens einer gepackten Datei konnte der OEP durch den Emulator erreicht werden.

Err Bei mindestens einer gepackten Datei konnte der OEP nicht durch den Emulator erreicht werden.

Gr Grund aus dem der OEP nicht erreicht werden konnte.

1. Nicht implementierte Funktionen der Windows API.
2. Unzureichend emulierte Struktur der Windows DLL Dateien.
3. Keine Unterstützung für mehrere Prozesse oder Threads.
4. Keine Unterstützung für die Debug Register des Prozessors.
5. Keine Unterstützung für Speicherseitenrechte.
6. Unnötig hohe Speichernutzung bei der Sicherung des Emulatorzustands.
7. Nicht implementierte CPU Anweisungen.

SMD Der OEP mindestens eines der kleineren gepackten Programme konnte direkt gefunden werden.

SMN Der OEP mindestens eines der kleineren gepackten Programme konnte erst nachträglich gefunden werden.

SMO Der OEP mindestens eines der kleineren gepackten Programme konnte erst nach einer erfolgreichen Falsifikation eines OEP korrekt ermittelt werden.

BGD Der OEP mindestens eines der größeren gepackten Programme konnte direkt gefunden werden.

BGN Der OEP mindestens eines der größeren gepackten Programme konnte erst nachträglich gefunden werden.

BGO Der OEP mindestens eines der größeren gepackten Programme konnte erst nach einer erfolgreichen Falsifikation eines OEP korrekt ermittelt werden.

Packer	SM	BG	OK	Err	Gr	SMD	SMN	SMO	BGD	BGN	BGO
ACProtect 2.0	o	X	X	X	(6)				X		X
Armadillo 1.82	X	X		X	(1,3)						
Armadillo 4.64	X	X		X	(1,3)						
ASPack 1.0b	X	X	X				X		X		
ASPack 2.12	X	X	X				X		X		
ASProtect 1.23 rc4	X	X		X	(2)						
BeRoEXEPacker 1.0	X	X	X	X	(2)	X	X		X		
EmbedPE 1.3	X	X		X	(4)						
EXE32pack 1.37	X	X	X			X				X	
EXE32pack 1.42	X		X			X	X				
EXECryptor 2.3.9	X	X		X	(2)						
ExeStealth 2.73		X	X						X		
eXPressor 1.451	X	X	X				X		X		
FSG 1.2	X	X	X			X			X		
FSG 1.31	X	X	X			X			X		
FSG 1.33	X	X	X			X			X		
FSG 2.0	X	X	X			X	X		X		
Hmimys 1.0	X	X	X				X		X		
JDPack 1.01	X	X	X			X			X		
KByS Packer 0.28b	X	X	X			X			X		
Krypton 0.2	X	X		X	(2)						
MEW 11	X	X	X			X	X		X		
Molebox 2.61		X		X	(1)						
Morphine 1.2b	X	X	X				X		X		
NeoLite 1.01	X	X		X	(1)						
NeoLite 2.0	X	X		X	(1)						
NoodleCrypt 2	X	X		X	(7)						
nPack 1.1b	X	X	X				X		X		
NSPack 3.7	X	X	X				X		X		
Obsidium 1.3	X	X		X	(2)						
Packman 1.0	X	X	X			X			X		
PE Pack 1.0	X	X	X				X		X		
PE-crypt 1.02	X	X		X	(3)						
PECompact 2.40	X	X	X			X			X		
PECompact 2.78a	X	X	X			X			X		
ped 0.1	X	X	X			X			X		

Packer	SM	BG	OK	Err	Gr	SMD	SMN	SMO	BGD	BGN	BGO
PeLockNT 2.04	X	X	X				X		X		
PEncrypt 4	X	X	X				X		X		
PESpin 0.3	X	X		X	(1)						
petite 2.2	X	X		X	(5)						
petite 2.3	X	X		X	(5)						
PeX 0.99	X	X	X				X	X	X		X
polyene 0.01 plus	X	X	X				X		X		
RLPack 1.0b	X	X	X			X			X		
RLPack 1.17	X	X	X				X		X		
Shrinkwrap 1.4	X	X	X			X	X				
SimplePack 1.0	X	X	X			X			X		
SimplePack 1.21	X	X	X			X	X		X		
tElock 0.42	X	X	X			X			X		
tElock 0.98	X	X		X	(4)						
Upack 0.25b	X	X	X			X			X		
Upack 3.99	X	X	X				X		X		
UpolyX 0.5	X		X			X					
UPX 1.20	X	X	X			X			X		
UPX 1.91	X	X	X			X			X		
UPX 2.02	X	X	X			X			X		
UPX 2.92b	X	X	X			X			X		
UPX Scrambler 3.02	X	X	X			X			X		
UPXCrypt		X	X						X		
UPXRedir		X	X						X		
UPXshit	X	X	X			X			X		
Vgcrypt 0.75	X	X	X			X			X		
WinKript 1.0		X	X						X		
WWPack 1.20		X	X						X		
yoda’s crypter 1.2		X	X	X	(1)				X		
yoda’s prot 1.0b		X		X	(1)						
yzpack 2.0b	X	X	X				X		X		

Tabelle A.1: Detailauflistung der Entpackergebnisse

A.1.2 Erstellte Signaturen

Die in Tabelle A.2 gelisteten Signaturen wurden für die in Abschnitt 6.1 beschriebenen Tests verwendet.

Signaturname	Signatur
Testfile_HelloWorld_NoEP	`:1:*:??00E815000000{40}6A00E8`
Testfile_HelloWorld	`:EP+0:*:??00E815000000{40}6A00E8`

Testfile_HelloWorldBig_NoEP	`:1:*:??00100000CC{4096}`
	`586A0068{100-}6F6F6F6F{4092}`
	`556E7061636B6564`

Testfile_HelloWorldBig	`:1:EP+0:??00100000CC{4096}`
	`586A0068{100-}6F6F6F6F{4092}`
	`556E7061636B6564`

Testfile_Lua_NoEP	`:1:*:??8BEC6AFF68{66804}`
	`64890D000000005F5E5BC9C3-`
	`{100-} 244C75613A204C7561{7000-}`
	`257300005F414C455254`

Testfile_Lua	`:1:EP+0:??8BEC6AFF68{66804}`
	`64890D000000005F5E5BC9C3-`
	`{100-} 244C75613A204C7561{7000-}`
	`257300005F414C455254`

Testfile_Base64Tool_NoEP	`:1:*:??89E583EC08C7042401000000`
	`{128088}741C85F6740C895C2404`
	`{2000-}537431306261645F`

Testfile_Base64Tool	`:1:EP+0:??89E583EC08C7042401000000`
	`{128088}741C85F6740C895C2404`
	`{2000-}537431306261645F`

Tabelle A.2: Für die Testdateien erstelle Signaturen

A.2 Verhaltensdiagramme von Laufzeitpackern

Dieser Abschnitt enthält Lese- Schreib- und Ausführungszugriffe ausgewählter Entpackroutinen. Hierbei sind Schreib- und Ausführungszugriffe hervorgehoben.

Abbildung A.1: Direkt nach Programmstart werden Anweisungen im Bereich des entpackten Programms ausgeführt. Diese werden im Verlauf des Entpackvorgangs mit dem entpackten Programm überschrieben.

Abbildung A.2: Wenn die Entpackroutine beendet ist wird ein Programmfehler verursacht. Als Exception-Handler wurde zuvor der OEP des entpackten Programms eingetragen. Dieses wird nun durch den den Fehlerbehandlungsmechanismus des Betriebssystems angesprungen.

Abbildung A.3: Bevor die Ausführung den korrekten OEP erreicht, wird zunächst ein falscher OEP generiert und angesprungen, welcher anschließend mit dem entpackten Programm überschrieben wird.

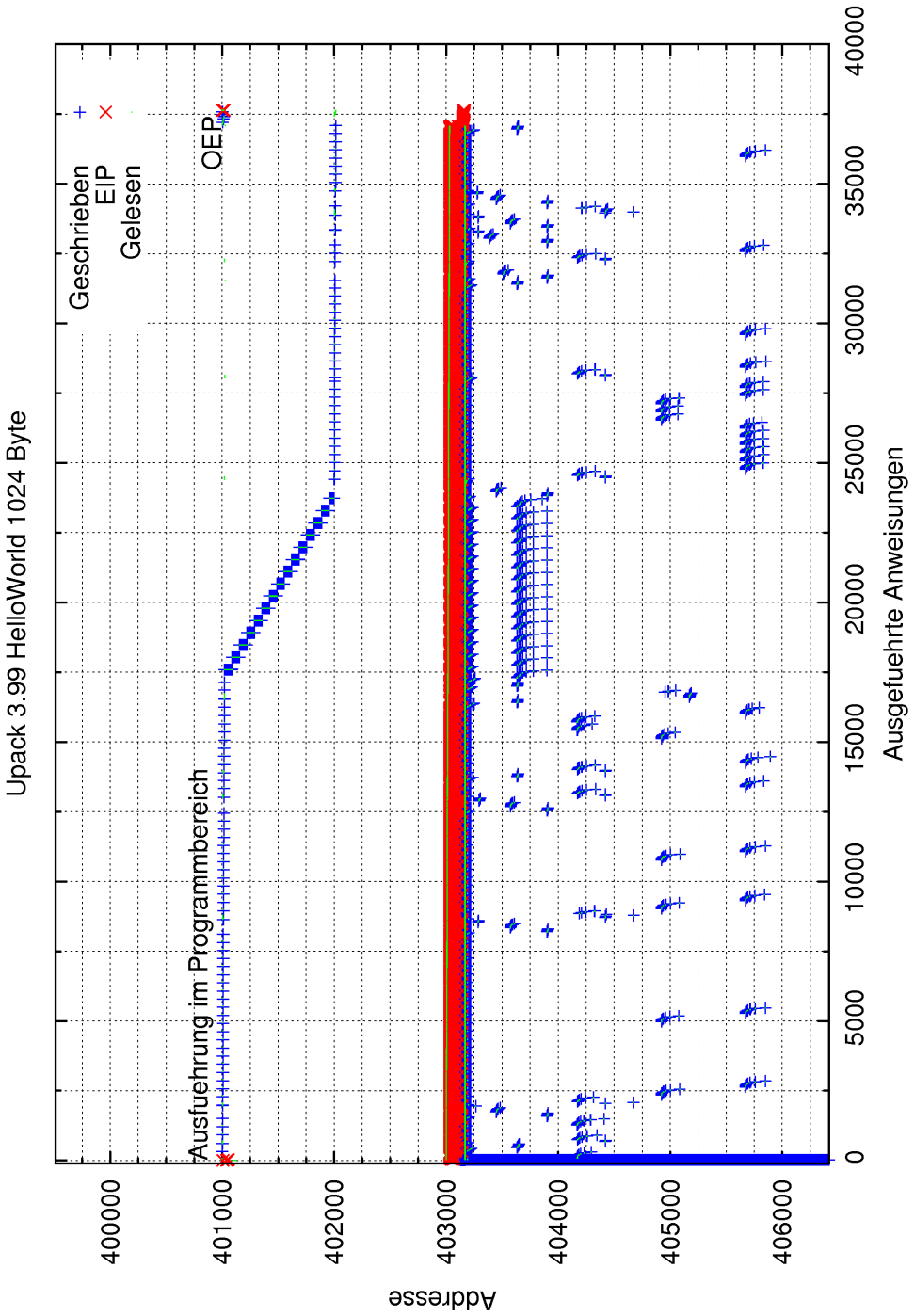

Abbildung A.1: Verhaltensdiagramm von Upack 3.99

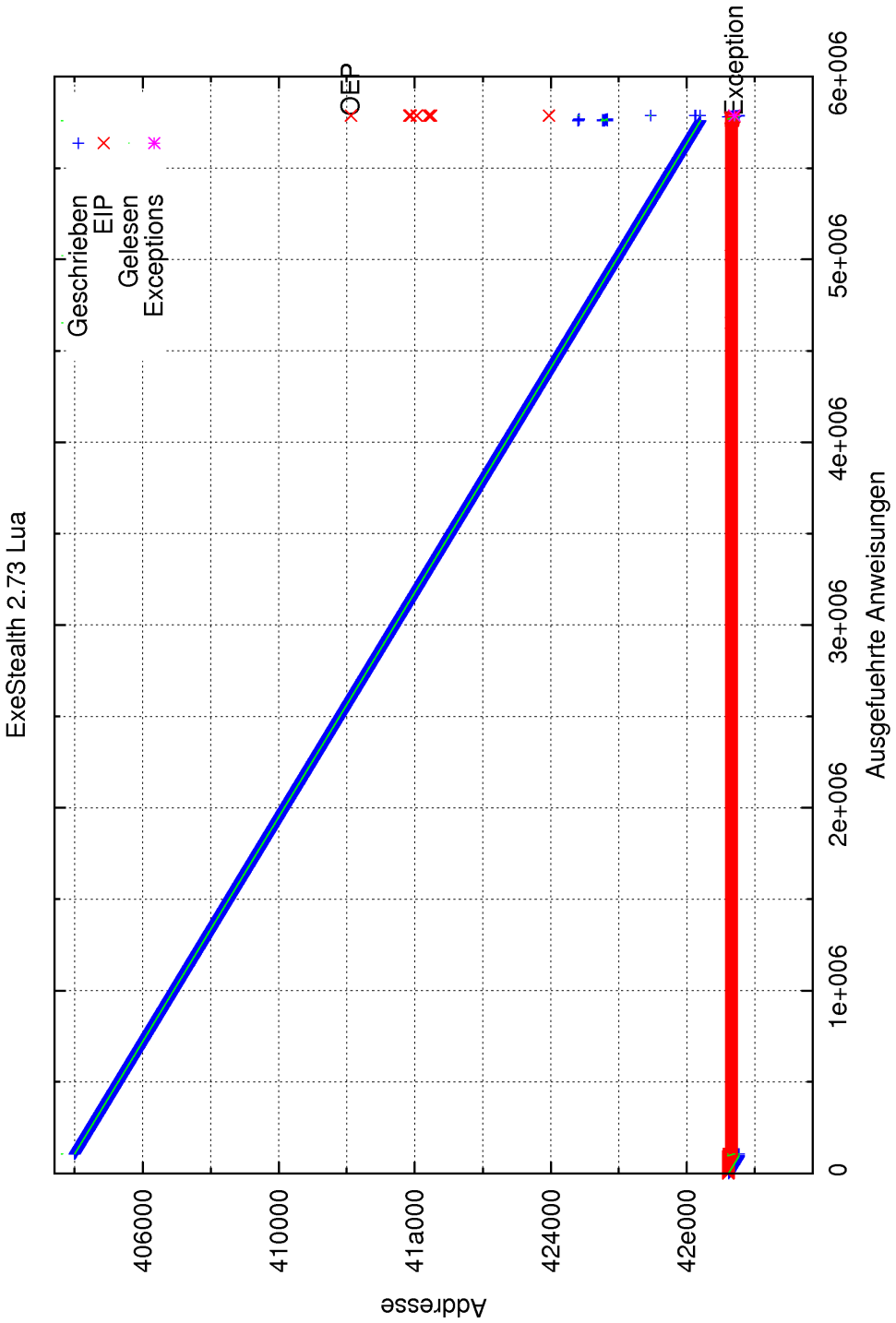

Abbildung A.2: Verhaltensdiagramm von AsPack 2.12

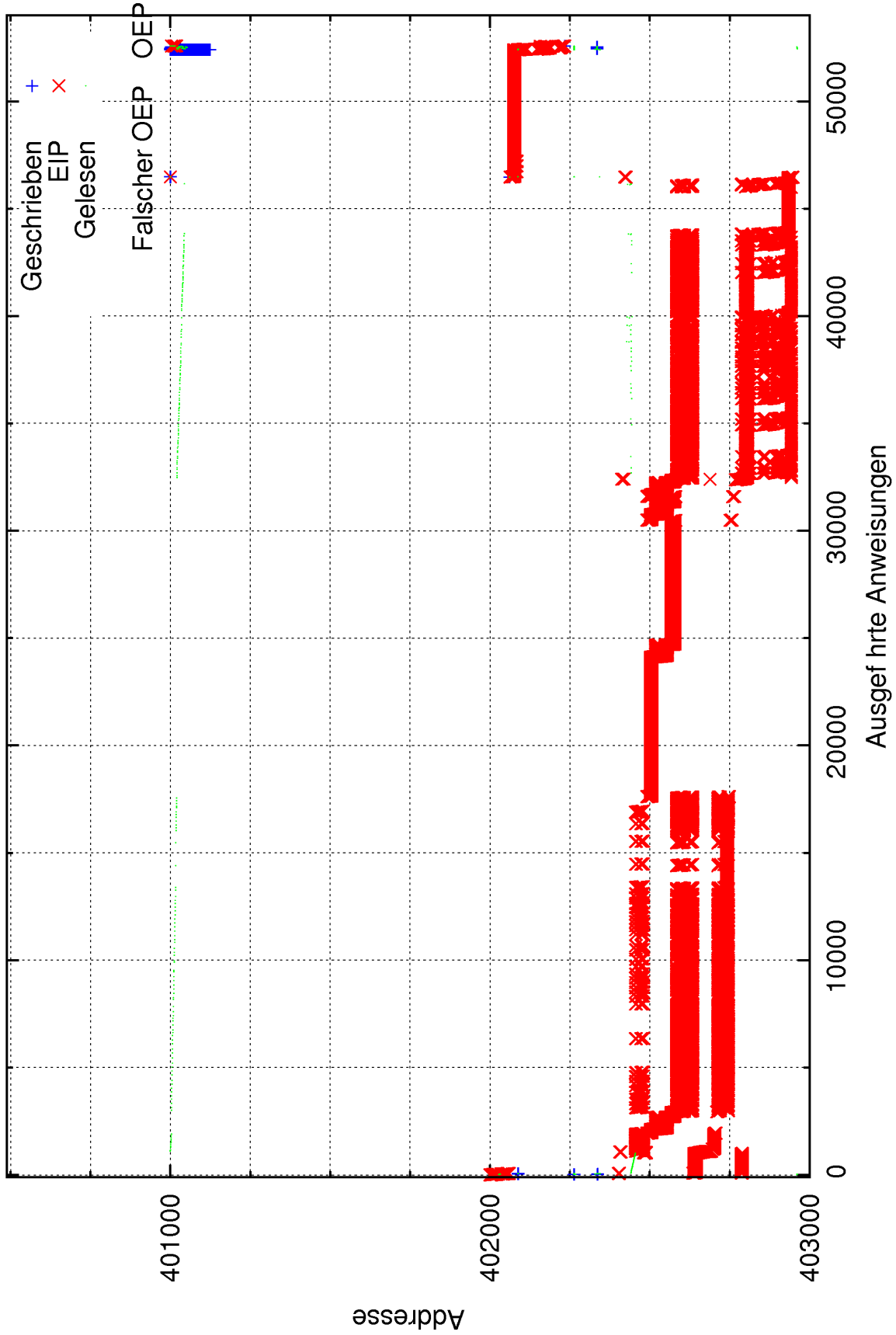

Abbildung A.3: Verhaltensdiagramm von AsPack 2.12

B Glossar

API Application Programming Interface: Eine normalerweise aus Funktionen bestehende Schnittstelle zur Nutzung vorhandener Funktionalität.

Einstiegspunkt Adresse an der die Ausführung einer ausführbaren Datei im PE-Format begonnen wird.

EP Siehe Einstiegspunkt.

EP-basierte Signatur Eine EP-basierte Signatur wird nicht an beliebigen Stellen in der Datei gesucht, sondern nur relativ zu dem EP der Datei.

Präprozessor Ein Präprozessor ist ein Programm, das einen Eingabetext ggf. anhand in vorhandener Instruktion konvertiert und das Ergebnis ausgibt.

OEP Siehe Originaler Einstiegspunkt.

Originaler Einstiegspunk Adresse an der die Ausführung des entpackten Programms beginnt nachdem es entpackt worden ist.

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 18. Juli 2007

Ort, Datum

Unterschrift