Bachelorarbeit

Heuristiken in Unpacking-Frameworks

Patrick Schulz

Institut für Informatik IV
Rheinische Friedrich-Wilhelms-Universität Bonn

Prof. Dr. Peter Martini
Dr. Michael Meier

10. August 2011

Ich versichere, die vorliegende Bachelorarbeit selbstständig erstellt zu haben. Alle Zitate sind als solche gekennzeichnet. Ich habe keine anderen Hilfsmittel und Quellen, als die angegebenen, verwendet. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Bonn, 10. August 2011 Patrick Schulz

Zusammenfassung

Die Bachelorarbeit beschreibt die Evaluation von Heuristiken aus Unpacking-Frameworks. Dazu wurde ein Framework zur Anwendung und Bewertung dieser konzipiert und implementiert. Die Evaluation zeigt Stärken und Schwächen der Heuristiken hinsichtlich ihres Erfolges beim Entpacken von Programmen auf. Die so gewonnenen Erkenntnisse zeigen Möglichkeiten zur Verbesserung der Heuristiken und deren Anwendung auf.

Inhaltsverzeichnis

1 Einleitung

Malware findet sich heutzutage in allen Bereichen, in denen Computersysteme genutzt werden. Das reicht vom privaten Gebrauch bis in sensible industrielle Bereiche hinein. Die starke Vernetzung der Computersysteme bietet der Malware dabei ein Medium, um sich schnell verbreiten zu können. Die professionalisierte Entwicklung der Malware führt zu einer steigenden Anzahl. Der dadurch entstehende finanzielle Schaden macht entsprechende Gegenmaßnahmen zur Bekämpfung notwendig. Um dieser Entwicklung entgegenzuwirken werden Virenscanner eingesetzt. Diese können das Problem jedoch nur lindern und nicht lösen, da auch die Entwickler von Malware neue Verfahren entwickelt haben.

Dabei nutzt Malware verschiedene Techniken zur Verschleierung, um sich zum einen vor Virenscannern und zum anderen vor Analysen zu schützen. Denn Malware lebt davon möglichst lange von Virenscannern unerkannt zu bleiben. Die Virenscanner könnten versuchen die Nutzung solcher Techniken bei Programmen zu erkennen und die Ausführung zu verhindern, doch würde dieses Vorgehen nicht nur Malware treffen. Diese Techniken werden unter anderem auch in kommerziellen Softwareprodukten eingesetzt, um diese vor Analysen zu schützen. Dabei liegt das Interesse meist im Schutz der enthaltenden Algorithmen, die nicht kopiert werden sollen. Daher bedeutet die Benutzung der Techniken nicht, dass es sich um Malware handeln muss.
Diese Techniken werden als Packen von Programmen bezeichnet. Dies beinhaltet, im Kontext von Malware, neben der Komprimierung meist auch die Verschlüsselung, sowie die Verschleierung des Programmcodes. Eine genauere Beschreibung findet sich im Kapitel 2.2. Das Packen soll die Erstellung von zuverlässigen Signaturen der Malware verhindern, die unter anderem von Virenscannern zur Erkennung der Malware genutzt werden. Das Packen wird aber auch eingesetzt, um die Programmgröße zu verringern und so zum Beispiel die Verbreitung zu vereinfachen.
Um diese Programme dennoch untersuchen zu können wurden verschiedene Verfahren entwickelt, um den Schutz wieder automatisch zu entfernen. Diese basieren meist auf Heuristiken, die im Kontext von Unpacking-Frameworks eingesetzt werden. Die Unpacking-Frameworks und die darin genutzten Heuristiken sind im Kapitel 3 beschrieben. Dies führt zu einer ständigen Weiterentwicklung auf beiden Seiten. Sowohl die Entwickler der Packer, als auch die der Unpacking-Frameworks, müssen ihre Methoden und Verfahren immer weiter anpassen und verbessern.

Im Fokus dieser Arbeit stehen die Heuristiken dieser Frameworks, da das erfolgreiche Entpacken stark von diesen abhängt. Daher ist es wichtig die Effektivität der Heuristiken einschätzen zu können, um so eine gute Auswahl für die Nutzung in Unpacking-Frameworks treffen zu können. In dieser Arbeit werden die Heuristiken aus den bekannten Frameworks vorgestellt, untereinander verglichen und bewertet. Die Ergebnisse sollen Stärken und Schwächen der Heuristiken aufzeigen und eine Grundlage bieten effektivere Heuristiken zu entwickeln. Dabei werden folgende Teilziele erreicht:

- Ein Framework zur Anwendung der Heuristiken wird entwickelt.
- Geeignete Heuristiken werden ausgewählt und in dem Framework implementiert.
- Metriken zur Bewertung der Heuristiken werden ausgewählt und implementiert.

- Die Evaluation der Heuristiken gegen ausgewählte Packer wird durchgeführt, welches das Kernziel dieser Arbeit ist.

Im Kapitel 2 werden die grundlegenden Begrifflichkeiten, die für das Verständnis dieser Arbeit notwendig sind, erklärt. Dabei werden unter anderem im Kapitel 2.1 die für diese Arbeit relevanten Aspekte der Computerarchitektur erklärt sowie die technischen Aspekte des Packens im Kapitel 2.2. Des Weiteren werden Verfahren zum Entpacken der Programme im Kapitel 2.3 und die Binäranalyse im Kapitel 2.4 vorgestellt. Im Kapitel 2.5 wird die in dieser Arbeit eingesetzten Technik der Instrumentation erklärt.

Das Kapitel 3 schafft eine Übersicht der vorhandenen Arbeiten in diesem Themengebiet. Dies beinhaltet insbesondere die bekannten Unpacking-Frameworks sowie die hier zu vergleichenden Heuristiken.

Im 4. Kapitel wird die Evaluationsumgebung vorgestellt. Dies beinhaltet die Annahmen, die den Heuristiken zugrunde liegen sowie den Anforderungen die an das zu entwickelnde Framework gestellt werden. Das Framework, mit seinem Aufbau und den Abläufen darin, wird im Kapitel 4.3 erarbeitet. Die zu bewertenden Heuristiken werden im Kapitel 4.4 beschrieben. Die zur Bewertungen notwendigen Metriken werden im Kapitel 4.5 diskutiert.

Das Kapitel 5 stellt die Bewertung der Heuristiken vor und zeigt Vor- und Nachteile dieser auf. Das Fazit ist im Kapitel 6 beschrieben. Im letzten Kapitel werden mögliche Fortführungen dieser Arbeit aufgezeigt.

2 Grundlagen

Dieses Kapitel gibt eine Einführung in die grundlegenden Begrifflichkeiten, die zum Verständnis dieser Arbeit notwendig sind. Dabei wird das Konzept der weit verbreiteten x86-Architektur im Kapitel 2.1 vorgestellt. Darauf aufbauend werden im Kapitel 2.2 die grundlegenden Konzepte von Packern erklärt. Im Kontext der Binäranalyse, die im Kapitel 2.4 beschrieben wird, ist die Umkehrung des Packvorgangs notwendig. Anhand von zwei Methoden wird dieser Entpackvorgang im Kapitel 2.3 vorgestellt. Die dafür hilfreiche Technik der Instrumentation von Programmen wird im Kapitel 2.5 erklärt.

2.1 Computerarchitektur

In diesem Kapitel werden die Aspekte der Computerarchitektur erklärt, die für das Verständnis dieser Arbeit notwendig sind. Darüber hinaus bilden diese Aspekte die Grundlagen für die Funktionsweisen der Heuristiken und Metriken und nehmen daher in dieser Arbeit eine besondere Rolle ein.

Von-Neumann-Architektur

Die meisten Computersysteme basieren heutzutage auf der x86-Architektur. Dies ist eine Von-Neumann-Architektur, was bedeutet, dass Programmcode und Nutzdaten sich gemeinsam im selben Speicher befinden. Deswegen können Speicherinhalte nicht ohne weitere Informationen als Programmcode oder Nutzdaten klassifiziert werden. Die Zuordnung hängt von der Interpretation ab, die sich meist erst zur Laufzeit entscheidet, auch

eine Mehrfachbenutzung als Programmcode und Nutzdaten ist in der Von-Neumann-Architektur möglich.

Diese Tatsache nimmt an mehreren Stellen dieser Arbeit Einfluss, insbesondere bei der Konzeptionierung und Implementation der Metriken.

Basic Block

Ein Basic Block ist ein Tupel von Instruktionen, wobei deren Reihenfolge durch die Ausführungszeitpunkte der Instruktionen definiert ist. Für einen Basic Block gilt immer, wenn die erste Instruktion ausgeführt wird, werden auch alle anderen Instruktionen ausgeführt. Basic Blocks werden meist durch Instruktionen beendet, die eine Kontrollflussänderung erzeugen, wie zum Beispiel "CALL" und "JMP".

Import Address Table

Beim Starten eines Programmes werden die Sektionen durch den Dynamischen Linker in den Arbeitsspeicher geladen. Darüber hinaus wird die Import Address Table (IAT) in Teilen befüllt. Dazu wird die Import Table genutzt, die im Programmheader enthalten ist und angibt, welche Funktionen aus den Dynamic-Link Librarys (DLLs) benötigt werden. Nachdem die Kopien der benötigten DLLs in den Speicher geladen wurden, werden die daraus resultierenden, konkreten Speicheradressen der gewünschten Funktionen vom Dynamischen Linker in die IAT eingetragen, damit das Programm diese Funktionen während der Ausführung nutzen kann. Dieses Konzept wurde entwickelt, damit verschiedene Programme auf eine Sammlung von Funktionen zugreifen können. Dies hat den Vorteil, dass diese Funktionen nicht in jedem Programm neu implementiert werden müssen. Im Anschluss kann das Programm gestartet werden. Dazu führt der Prozessor das Programm beginnend am Entry Point aus. Der Entry Point ist der Einsprungpunkt eines Programmes, der im Header des Programmes gespeichert ist.

2.2 Packer

Packer werden auf Programme angewandt, um diese zu verkleinern und vor Analysen zu schützen. Dabei wird die Codestruktur des Programmes verändert, jedoch nicht dessen Funktionalität. Die Veränderung kann dabei eine Komprimierung, Verschlüsselung und Verschleierung des Programmes umfassen. Diese Aspekte sind insbesondere im Kontext von Malware bedeutsam.

Das Packen bestand anfänglich meist nur aus einer Komprimierung, die die Größe des Programmes verringern sollte. Dies kann bei der Verbreitung von Programmen hilfreich sein. Heutzutage wird die Komprimierung oft durch eine Verschlüsselung ergänzt, um die Erstellung von zuverlässigen Signaturen des Programmes zu verhindern. Im Kontext von Malware kommt dies stark zum Tragen, da solche Signaturen unter anderem von Virenscannern zur Erkennung von Malware eingesetzt werden. Darüber hinaus werden auch Verschleierungsmethoden eingesetzt, um die Analyse des Programmes zu erschweren. Es können beim Packen auch die drei Methoden kombiniert werden. Das ursprüngliche Programm kann nach dem Packen nicht mehr direkt ausgeführt werden, da der Programmcode nur gepackt vorliegt. Deswegen wird dem Programm weiterer Programmcode hinzugefügt, der Entpackerstub. Dieser Stub besteht aus Programmcode, der die Ausführung

des gepackten Programmcodes vorbereitet. Das bedeutet, dass der Entpackerstub das Programm so in den Speicher legen muss, als wäre es nicht gepackt. Dazu muss der Entpackerstub insbesondere die Entpackfunktion enthalten. Des Weiteren muss die IAT, deren Nutzen im Kapitel 2.1 erläutert wurde, vorbereitet werden. Dies übernimmt bei einem nicht gepackten Programm der Dynamische Linker. Die dafür benötigten Informationen werden vom Packer jedoch meist mit gepackt, so dass der Dynamische Linker diese Aufgabe nicht mehr übernehmen kann. Damit soll verhindert werden, dass diese Informationen analysiert werden, um Rückschlüsse auf die Funktionalität des gepackten Programmes zu ziehen. Der Entpackerstub muss dementsprechend vor dem ursprünglichen Programmcode ausgeführt werden.

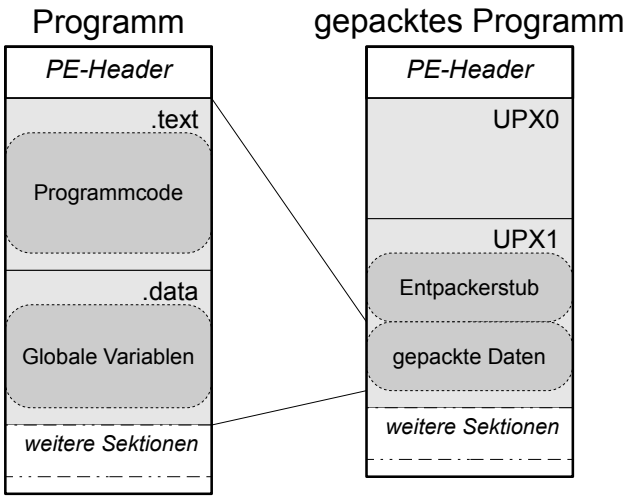

Abbildung 1: Der Aufbau eines gepackten Programmes im Vergleich zu der nicht gepackten Version am Beispiel von UPX [19]

Abbildung 1 zeigt, am Beispiel des Packers UPX [19], die Unterschiede der Programmdateien zwischen einem nicht gepackten und einem gepackten Programm. Dabei besteht die Datei des gepackten Programmes aus dem Header sowie aus den Sektionen. Die im ursprünglichen Programm auf mehrere Sektionen verteilte Daten, in dem Fall auf die Sektionen ".data" und ".text", werden durch das Packen komprimiert und in eine Sektion geschrieben. Im diesem Fall in die Sektion "UPX1". Der Entpackerstub wird ebenfalls in diese Sektion geschrieben. Eine weitere Sektion "UPX0" ist dabei leer und wird durch den Entpackerstub zur Laufzeit mit den entpackten Programmcode befüllt. Wenn die Vorbereitungen durch den Entpackerstub abgeschlossen sind, wird der Kontrollfluss an den entpackten Programmcode übergeben und die ursprüngliche Funktionalität des Programmes ausgeführt. Diese Stelle im Programm wird der Original Entry Point (OEP) genannt, da hier der ursprüngliche Einsprungpunkt des Programmes liegt.

2.3 Entpacken

Das Entpacken bezeichnet die Umkehrung des Packens, wie es im Kapitel 2.2 beschrieben wurde. Ziel ist es, das ursprüngliche Programm wieder zu extrahieren, um es in seiner ursprünglichen Form analysieren zu können. Dabei stellt das Entpacken von Programmen eine Herausforderung dar, da die Entscheidung, ob ein Programm gepackt ist, unentscheidbar ist. Das bedeutet, dass es kein Algorithmus gibt, der für alle möglichen Programme entscheiden kann, ob das Programm gepackt ist oder nicht. Diese Entscheidung lässt sich auf das Halteproblem reduzieren [13].

Die Ausgangslage beim Entpacken ist meistens das Vorhandensein eines gepackten Programmes, welches wieder extrahiert werden soll, um es analysieren zu können. Dies ist insbesondere bei Malware oft der Fall, da Malware meist gepackt ist und eine Analyse für die Bekämpfung notwendig ist. Davon ausgehend, dass das vorliegende Programm gepackt ist, besteht es aus zwei Hauptkomponenten, dem Entpackerstub und dem Programm, welches in gepackter Form enthalten ist. Der Entpackerstub ist dahingehend interessant, als dass er den Algorithmus zum Entpacken des Programmes enthält. Der Fokus beim Entpacken liegt in dem enthaltenen Programm, welches durch die Anwendung des Algorithmus entpackt und extrahiert werden soll. Eine erweiterte Definition von Entpacken beinhaltet darüber hinaus auch das Überführen des extrahierten Programmes in eine ausführbare Datei.
Hierfür sind zum einen Methoden zur Binäranalyse notwendig, um Informationen über das vorliegende Programm zu erarbeiten. Zum anderen werden Vorgehensweisen zum Entpacken von Programmen benötigt. Im Folgenden wird die Binäranalyse vorgestellt sowie zwei Verfahren beschrieben, um gepackte Programme entpacken zu können.

2.4 Binäranalyse

Im Kontext des Entpackens werden Binäranalyseverfahren benötigt, um die Programme zu analysieren. Dabei sind diese generell auf Programme anwendbar und nicht auf die Menge der gepackten Programme beschränkt. Die Binäranalyse wird genutzt, um Informationen, Algorithmen und Fehler in Programmen zu finden und zu analysieren. Es gibt zwei grundsätzliche Ausrichtungen der Binäranalyse: die statische und die dynamische Analyse. Diese unterscheiden sich insbesondere in dem Merkmal, ob das zu untersuchende Programm zu Analysezwecken ausgeführt wird oder nicht.

Statische Analyse

Die statische Analyse zeichnet sich dadurch aus, dass das zu analysierende Programm nicht ausgeführt wird, um die gewünschten Informationen zu bekommen. Dazu gehört die Analyse der Datei sowie deren Format, aber auch die Inhalte des Dateiheaders. Hieraus lassen sich wichtige Informationen gewinnen, die Aufschluss auf die Funktionsweisen des Programmes geben. Dies schließt insbesondere die Analyse der Import Table ein, die Informationen darüber enthält, welche Funktionen in DLLs in dem Programm sehr wahrscheinlich genutzt werden. Die Aussage darüber, ob eine solche Funktion genutzt wird, kann nur durch eine Ausführung des Programmes getroffen werden. Dies fällt in den Kontext der dynamischen Analyse, die im nächsten Abschnitt erklärt wird.

Ein Großteil der Informationen über das Programm werden aus dem Programmcode gewonnen. Dieser ist als Maschinencode in der Datei enthalten und ist die binäre Repräsentation der Instruktionen, die von dem Prozessor verstanden und ausgeführt werden. Da die Interpretation des Maschinencodes für den Menschen oft zu aufwendig ist, wird der Code durch einen Disassembler in eine textuelle Form gebracht. Diese Form, die Assemblersprache, ist trotz ihrer Systemnähe deutlich zugänglicher als der Maschinencode und erlaubt eine bessere Analyse des Codes.
Im Kontext von gepackten Programmen stößt die statische Analyse an ihre Grenzen, da der interessante Codeteil gepackt ist und ohne Kenntnis über den Entpackalgorithmus dieser nicht untersucht werden kann.

Dynamische Analyse

Bei der dynamische Analyse wird das zu analysierende Programm ausgeführt, um die gewünschten Informationen zu bekommen. Dabei wird das Programm und insbesondere dessen Verhalten beobachtet. Es können auch Methoden der statischen Analyse genutzt werden, um ein Informationsgrundlage zu schaffen. Diese Informationen werden durch die dynamische Analyse ergänzt und ergeben so ein detaillierteres Bild, aus dem die gewünschten Informationen extrahiert werden können. So lassen sich durch eine statische Analyse zum Beispiel Funktionsaufrufe finden. Jedoch kann die Reihenfolge, in der diese aufgerufen werden, nicht immer bestimmt werden. Ein Grund hierfür kann die Abhängigkeit von der Laufzeitumgebung oder der Benutzerinteraktion sein, die in der statischen Analyse nicht beachtet werden kann. Diese dynamische Analyse kann genau diese Informationen beisteuern. Die dynamische Analyse bietet im Kontext von gepackten Programmen den Vorteil, dass auch der gepackte Codeteil analysiert werden kann, da dieser vor der Ausführung entpackt wird.

2.4.1 Signaturbasierte Verfahren

Das Entpacken von Programmen mithilfe von signaturbasierten Verfahren ist eine einfach anzuwendende Methode, die meistens bei einer statischen Analyse eingesetzt wird. Dabei wird der verwendete Packalgorithmus anhand von vorher definierten Signaturen erkannt, die unter anderem aus signifikanten Codeteilen des Entpackerstubs bestehen kann. Nach der erfolgreichen Erkennung des Algorithmus kann ein vorher entwickelter Entpackalgorithmus angewandt werden, um das Programm zu entpacken. Dieses Vorgehen bedingt, dass die verwendeten Algorithmen bekannt und insbesondere vorher analysiert wurden. Des Weiteren müssen Signaturen vorhanden sein und die genutzte Signaturdatenbank immer aktuell gehalten werden. Daher sind dieses Verfahren nicht generisch anwendbar. Darüber hinaus ist das Verfahren nur bedingt einsetzbar, wenn der Entpacker polymorph ist. Dies bedeutet, dass der Entpacker seine eigene Codestruktur nach der Zeit verändert. Daher lassen sich in diesem Fall nur sehr schwer zuverlässige Signaturen erstellen.

Ein sehr verbreitetes Programm, welches so ein signaturbasiertes Verfahren nutzt, um den verwendeten Packer zu identifizieren, ist PEiD [9]. Dieser kann das gepackte Programm jedoch nicht entpacken. Dieses Projekt wurde im Jahr 2011 beendet, was das Problem des Aktualisierens der Signaturdatenbank verdeutlicht.

2.4.2 Heuristikbasierte Verfahren

Eine generische Methode ist die Anwendung von Heuristiken, die den Ausführungszeitpunkt des Entpackerstubs erkennen. Dabei wird insbesondere versucht das Ende des Entpackerstubs zu finden und damit auch den Anfang des eigentlichen Programmes. Die Heuristiken finden meist bei der dynamischen Analyse Anwendung. Es wird dabei das gepackte Programm ausgeführt und zur Laufzeit analysiert. Die so gewonnenen Informationen werden von der Heuristik verarbeitet und dann in einer vorher definierter Art und Weise reagiert.

Dieser generische Ansatz hat den Vorteil, dass auch unbekannte Packalgorithmen damit verarbeitet werden können. Jedoch ist die Methode anfällig für Falschmeldungen und bedarf daher wesentlich mehr Aufwand bei der Anwendung und Auswertung. Ein weiterer Nachteil ist, dass die bekannten Heuristiken, die im Kapitel 4.4 vorgestellt werden, teils sehr starke Annahmen machen müssen, damit sie anwendbar sind. Im Kontext von Malware hat dies zur Folge, dass Malwareautoren versuchen bei der Erstellung genau diese Annahmen aus Kapitel 4.1 nicht zu erfüllen und damit die Heuristiken fehlschlagen lassen.

2.5 Programminstrumentierung

Die Programminstrumentation bezeichnet die Methode den Code eines Programmes zur Laufzeit nach vordefinierten Algorithmen beliebig modifizieren zu können. Damit kann die Funktionalität des Programmes verändert und insbesondere erweitert werden.

Im Kontext dieser Arbeit besitzt die Instrumentation eine weitere wichtige Einschränkung. Bei der Instrumentation eines Programmes darf dieses zur Laufzeit verändert werden, jedoch darf diese Änderung keinen Einfluss auf die Funktionalität des Programmes haben. Dies wird durch die Forderung verschärft, dass der Eingriff vom instrumentierten Programm nicht erkennbar sein darf.

Bei der Binäranalyse liegt der Kernaspekt der Instrumentation im Hinzufügen von Funktionen an beliebigen Stellen des ursprünglichen Programmes. Dabei darf das ursprüngliche Programm und seine Funktionalität nicht beeinträchtigt werden. Die Anreicherung des Programmes mit eigener Funktionalität dient dabei zum Beispiel der Problemanalyse eines Programmes, indem an verschiedenen Stellen Laufzeitinformationen gesammelt und ausgewertet werden.
Die Instrumentierung eines Programmes kann dabei auf verschiedene Techniken aufsetzen, der Hardwarevirtualisierung, der Hardwareemulation oder der Binary Translation. Diese Techniken werden benötigt, um die Kontrolle über das jeweilige Programm zu behalten und so die gewünschte Funktionalität einbauen zu können.

Hardwarevirtualisierung

Die Hardwarevirtualisierung bietet eine Technik, um mehrere virtuelle Computersysteme gleichzeitig auf der Hardware bereit zu stellen. Dabei muss die verwendete Hardware diese Technik unterstützen. Damit ist es möglich, mehrere Systeme zu starten, die nicht zwangsweise von der Existenz der anderen Systeme wissen. Die System und deren Prozesse werden dabei direkt auf dem Prozessor ausgeführt.

Im Kontext der Instrumentation ist der Hypervisor ein weiterer wichtiger Aspekt der Hardwarevirtualisierung. Der Hypervisor ist eine Softwarekomponente, der die Vir-

tualisierung kontrolliert. Damit kann der Hypervisor Einfluss auf die virtuelle Maschinen nehmen. Dieser Einfluss ist für die Programminstrumentierung notwendig, um bei Bedarf die Ausführung des Programmes zu pausieren und die einzufügende Funktion aufzurufen. So ist es möglich mit der Hardwarevirtualisierung eine Programminstrumentierung zu erreichen.

Hardwareemulation

Die Hardwareemulation bietet, wie die Hardwarevirtualisierung, eine Technik, um mehrere virtuelle Computersysteme gleichzeitig bereit zustellen. Dabei ist diese Technik unabhängig von der darunterliegenden Hardware. Die virtuellen Systeme sind hier ebenfalls unabhängig von einander und beeinflussen sich gegenseitig nicht. Die virtuelle Hardware, die den Systemen zur Verfügung gestellt wird, ist bei der Emulation nicht als Hardware existent, sondern wird durch die Software emuliert. Dies bietet den Vorteil, dass den virtuellen Systemen Hardwarekomponenten zur Verfügung gestellt werden können, ohne dass diese außerhalb der Emulationsumgebung als Hardware zur Verfügung stehen. Da die Hardwarekomponenten in den virtuellen Systemen durch Software abgebildet werden, lassen sich deren Funktionalitäten und damit auch die Verhaltensweisen einfach verändern.

Die Hardwareemulation besitzt im Kontext der Programminstrumentation den Vorteil, dass insbesondere der Prozessor durch Software emuliert wird und damit die benötigte Funktionalität hinzugefügt werden kann, die für die Programminstrumentation notwendig ist. Dies ermöglicht es Programme zu instrumentieren, da der emulierte Prozessor mit dem zu analysierende Programm angehalten werden kann und die benötigten Laufzeitinformationen extrahiert werden können. Darüber hinaus ist zu jedem Zeitpunkt der gesamte Zustand des virtuellen Systems bekannt.

Binary Translation

Die Binary Translation bezeichnet die Überführung eines Programmcodeabschnittes von einem Befehlssatz in ein anderen Befehlssatz. Dabei können Quell- und Zielbefehlssatz auch der Selbe sein. Um mit dieser Technik eine Programminstrumentation zu erreichen werden beim Überführungsprozess weitere Instruktionen eingebaut. Diese rufen die gewünschte Funktion auf und ermöglichen so, die gewünschten Laufzeitinformationen zu sammeln.

2.6 Zusammenfassung

In diesem Kapitel wurden die Grundlagen besprochen, die im Kontext des Entpackens von Programmen relevant sind. Dies beinhaltet insbesondere die Herausforderungen und die Lösungsansätze, die das Entpacken mit sich bringt. Die in den Lösungsansätzen genutzten signaturbasierten und heuristikbasierten Verfahren wurden erklärt und die drin genutzten angewandten Methoden der statischen und der dynamischen Analyse erarbeitet. Eine weitere Grundlage für das Verständnis dieser Arbeit ist die Programminstrumentierung, die anhand von drei Techniken vorgestellt wurde und unter anderem in den Unpacking-Frameworks eingesetzt wird.

3 Verwandte Arbeiten

Das Entpacken von Malware stellt, wie im Kapitel 2.3 beschrieben, eine Herausforderung dar. Die bekannten Lösungsansätze basieren dabei auf nur wenigen grundlegenden Methoden. Zur Evaluation sind Unpacking-Frameworks entstanden, von denen ein Teil im Folgenden Kapitel vorgestellt werden. Dabei werden insbesondere die verwendeten Heuristiken herausgearbeitet, da diese die Grundlage für die Evaluation in dieser Arbeit bieten. Weitere Heuristiken werden darüber hinaus im Kapitel 3.2 vorgestellt.

3.1 Unpacking-Frameworks

Unpacking-Frameworks werden oft genutzt, um neue Heuristiken zu implementieren und um ihre Funktionalität zu evaluieren. Dabei ist das Ziel eines solchen Frameworks, die Malware zu entpacken und dem Analysten Informationen über den Packer und die Malware zu liefern. Im Folgenden werden Frameworks vorgestellt, die heuristikbasierte Verfahren, wie im Kapitel 2.4 beschrieben, einsetzen. Zu den Unterscheidungsmerkmalen der Frameworks zählt, neben dessen Umfang und der Auswahl der eingesetzten Heuristiken, insbesondere auch die Architektur des Frameworks. Dies schließt die gesamte Laufzeitumgebung mit ein, wozu unter anderem der Einsatz von Hardwarevirtualisierung, Hardwareemulation und anderen Techniken gehört.

Pandora's Bochs

Pandora's Bochs [3] ist ein Framework basierend auf dem Emulator Bochs [2]. Bochs emuliert ein komplettes x86-System. Dies bedingt, dass das Framework mit dem Betriebssystem im emulierten System interagieren muss, damit ein darin laufender Prozess instrumentiert werden kann. Hierzu bietet Bochs eine Schnittstelle an, um auftretende Ereignisse verarbeiten zu können und so eine Instrumentation der Prozesse zu ermöglichen. Dieser Aufbau bietet den Vorteil, dass das Framework nicht in dem Betriebssystem läuft, welches den zu instrumentierenden Prozess verwaltet. Somit hat die beobachtete Malware keinen Zugriff auf das Framework und kann dieses daher nur schwer erkennen. Damit bietet Bochs eine gute Grundlage um Heuristiken anzuwenden.

Die hier verwendete Heuristik "Write or eXecute" (WˆX) basiert auf der Idee, dass jede Ausführung von vorher modifiziertem Code ein potenzieller Beginn der Malware ist. Im Kapitel 4.4.5 wird diese Heuristik genauer beschrieben.
In Pandora's Bochs werden darüber hinaus noch weitere Kriterien genutzt, um einzelne Randfälle und Sonderheiten von Packern besser nutzen zu können und damit die Fehlerrate zu senken [3]. Diese werden in dieser Arbeit nicht betrachtet.

Justin

Justin [6] verfolgt einen anderen Ansatz als die restlichen Unpacking-Frameworks. Das Framework versucht gepackte Malware zu erkennen und nutzt dabei Konzepte aus anderen Unpacking-Frameworks, um die Malware im ersten Schritt zu entpacken. Dabei werden Virenscanner genutzt, um den Prozess zu verschiedenen Zeitpunkten zu scannen und damit die Malware zu erkennen. Als unterliegende Technik wird ein Microsoft Windows [17] verwendet, in dem sowohl die Malware als auch das Framework läuft. In

dem Malwareprozess und im Windowskernel werden Eventhandler registriert, wodurch die Instrumentation realisiert wird, die für die Implementation der Heuristiken notwendig ist.

Die Herangehensweise von Justin folgt einer einfachen Regel. Der Programmcode darf erst ausgeführt werden, nachdem er durch einen Virenscanner gescannt wurde [6]. Dieses Verfahren wird iterativ angewendet. Zum Beginn wird der gesamte Code gescannt und dann die Ausführung gestartet. Sobald sich Änderungen im Codebereich ergeben, werden diese Stellen markiert und im Fall einer Ausführung das gesamte Speicherbild neu gescannt. Falls kein Virenscanner eine Malware erkennt, werden Heuristiken herangezogen, um zu erkennen ob der Entpackvorgang abgeschlossen ist. Dabei wird eine Kombination aus drei Heuristiken genutzt. Die erste Heuristik nutzt den Sachverhalt, dass die Malware erst in den Speicher geschrieben und dann ausgeführt wird, wie im Kapitel 4.4.5 genauer beschrieben. Die zweite Heuristik basiert auf der Annahme, dass der Entpacker vor der Kontrollflussübergabe an die Malware den Stack wieder in den Ursprungszustand versetzt, "unwind the stack" [6]. Eine detailliertere Beschreibung findet sich in Kapitel 4.4.6. Die letzte Heuristik basiert ebenfalls auf der Annahme, dass der Entpacker vor dem Start den Ursprungszustand wiederherstellt. Als Indiz wird diesmal das Vorhandensein der Kommandozeilenparameter beim Kontrollflussübergang herangezogen. Diese Heuristik wird im Kapitel 4.4.7 genauer beschrieben.

Ether

Ether [4] ist ein Unpacking-Framework auf der Basis der Hardwarevirtualisierungstechnik Xen [30]. Die Technik wird dazu genutzt, um mehrere Systeme gleichzeitig auf der Hardware laufen zulassen. Diese sind in dem Anwendungsfall von Ether zum einen das System, in dem das zu analysierende Programm ausgeführt wird, und zum anderen ein weiteres System, das zur Analyse und Auswertung genutzt wird. Der modifizierte Xen Hypervisor ist dafür zuständig, dass nach interessanten Ereignissen in dem virtuellen System die Analysefunktion aufgerufen wird. Die Hardwarevirtualisierung bietet zwei wichtige Vorteile. Zum einen ist es für das Programm nur schwer zu erkennen, dass es analysiert wird und zum anderen bringt die Hardwarevirtualisierung einen Geschwindigkeitsvorteil mit sich, da das System direkt auf dem Prozessor läuft.
Die verwendete Heuristik ist vergleichbar mit der, die auch in dem Framework Pandora's Bochs eingesetzt wird. Es werden Speicherstellen, die zuvor durch den Prozess verändert wurden, als OEP betrachtet sobald sie ausgeführt werden, wie im Kapitel 4.4.5 beschrieben.

Eureka

Das Eureka-Framework [22] nutzt keine weitere Hardwareemulation oder Hardwarevirtualisierung, sondern führt das zu analysierende Programm direkt in dem nativen Betriebssystem aus. Dabei wird die Instrumentierung mittels eines Kerneltreibers realisiert, der die von dem Programm aufgerufenen System Calls abfängt.
Es werden in Eureka verschiedene Heuristiken genutzt, um einen abgeschlossenen Entpackvorgang zu erkennen. Zum einen wird der Aufruf der Application-Programming-Interface-Funktion, kurz API-Funktion, NtTerminateProcess als Indiz genutzt, wie im Kapitel 4.4.2 beschrieben. Dabei wird davon ausgegangen, dass vor der Benutzung dieser

API-Funktion das Programm entpackt wurde. Zum anderen wird der API-Aufruf von NtCreateProcess aus dem Kapitel 4.4.3 herangezogen. Der Aufruf wird unter anderem von Malware genutzt, um einen an den Prozess angehängte Debugger zu entfernen. Ein weiteres Fallbeispiel ist das Starten eines entpackten Programmes, das zuvor in eine Datei kopiert wurde. Beide Heuristiken lösen das Sichern eines Speicherabbildes aus, das im Anschluss durch das Framework weiter analysiert wird.

Renovo

Renovo [10] basiert auf TEMU, ein BitBlaze [26] Modul. BitBlaze ist eine Plattform, die von der Berkeley University of California genutzt und entwickelt wird, um Binäranalysen aus verschiedenen Perspektiven zu erstellen. Die dynamische Komponente der Analyse wird mit dem TEMU-Modul abgedeckt. TEMU nutzt dabei den System-Emulator QEMU [21]. Durch die Emulation des Systems hat TEMU eine große Kontrolle über das System. Die Heuristik wird in Renovo durch einen Schattenspeicher für jeden Prozess realisiert. In diesem werden die Zustände des Prozessspeichers gesichert. Bei Schreibzugriffen wird die modifizierte Stelle markiert und bei einer späteren Ausführung hier der OEP vermutet, wie im Kapitel 4.4.5 beschrieben.

3.2 Weitere Heuristiken

Neben den schon im vorherigen Kapitel 3.1 genannten Heuristiken existieren noch weitere Heuristiken, wie zum Beispiel Hump-and-dump [27].

Hump-and-dump basiert auf der Annahme, dass die Entpackfunktion in einer Schleife implementiert wurde. Die Annahme leitet sich von dem Aspekt ab, dass Packer unterschiedlich große Programme verarbeiten können und deswegen unterschiedlich viele Daten verarbeiten werden müssen. Dies ist durch eine Schleife realisierbar. Die Heuristik versucht die Schleife und das Verlassen dieser zu erkennen. Dabei wird die Mehrfachausführung der Instruktionen in der Schleife genutzt, um diese zu erkennen. Nach dem Verlassen der Schleife wird davon ausgegangen, dass der Entpackvorgang abgeschlossen ist. Eine detaillierte Beschreibung der Heuristik findet sich in Kapitel 4.4.1.

Eine weitere Heuristik nutzt als Indiz den Aufruf von LoadLibrary und GetProcAddress und wird im Kapitel 4.4.4 genauer beschrieben. Diese Heuristik wird in der "Universal PE Unpacker"-Erweiterung verwendet, die im IDA Pro Disassembler und Debugger enthalten ist [7]. Die zwei Windows API-Funktionen werden verwendet, um dynamisch weitere DLLs in den Prozess zu laden. Dies ist notwendig, um die Ausführung des Programmes vorzubereiten, da diese eine befüllte IAT, wie im Kapitel 2.3 beschrieben, benötigt. Die IAT beinhaltet Referenzen auf Funktionen in DLLs. Eine statische Analyse kann diesen Informationen nutzen, um damit Rückschlüsse auf die Funktionalität des gepackten Programmes zu ziehen. Um dies zu verhindern wird die Import Table durch Packer gelöscht und die IAT muss zur Laufzeit wieder hergestellt werden.

3.3 Fazit

Die existierenden Unpacking-Frameworks unterscheiden sich grundlegend voneinander. Dabei kommen unterschiedliche Techniken und Heuristiken zum Einsatz. Es wird unter anderem die Virtualisierung und Emulation eines Computers verwendet, aber auch

Kerneltreiber kommen dabei zum Einsatz. Mit den existierenden Unpacking-Frameworks lassen sich die eingesetzten Techniken untereinander Vergleichen, sofern ähnliche Heuristiken genutzt werden. Die Heuristiken sind jedoch so nur mit Einschränkungen untereinander vergleichbar, da ein erfolgreicher Entpackvorgang von der Technik und der Heuristik abhängt und der Einsatz unterschiedlicher Techniken daher bei einem Vergleich nicht vernachlässigt werden darf.

Viele Unpacking-Frameworks setzen die WˆX-Heuristik, die im Kapitel 4.4.5 im Detail beschrieben wird, ein. Darüber hinaus werden noch andere Heuristiken, wie die NtCreateProcess-Heuristik, nur in manchen Frameworks eingesetzt. Im Kapitel 3.2 wurden weitere Heuristiken erarbeitet, die in keinem dieser Unpacking-Frameworks genutzt werden.

Auf Grund dieser Unterschiede lassen sich die Heuristiken mit den zur Verfügung stehenden Frameworks nicht vergleichen, weshalb in dieser Arbeit ein Framework zum Vergleichen der Heuristiken geschaffen wird. Eine genaue Beschreibung der Heuristiken sowie das erstellte Framework findet sich im Kapitel 4.

4 Evaluationsumgebung

In dieser Arbeit sollen verschiedene Heuristiken, die in Unpacking-Frameworks genutzt werden, hinsichtlich ihrer Effektivität bewertet werden. Dabei soll analysiert werden, welche Heuristiken genutzt werden können, um gepackte Programme erfolgreich entpacken zu können. Dazu ist es notwendig die Heuristiken zu implementieren und anzuwenden. Die Ergebnisse, die durch die Anwendung geliefert werden, müssen bewertet und interpretiert werden, um Rückschlüsse auf die Effektivität der Heuristiken ziehen zu können. Um dieser Aufgabe gerecht zu werden wird ein Framework, wie in der Abbildung 2 skizziert, erstellt.

Das Framework soll diesen Ablauf mithilfe der Kontroll- und Steuerkomponente automa-

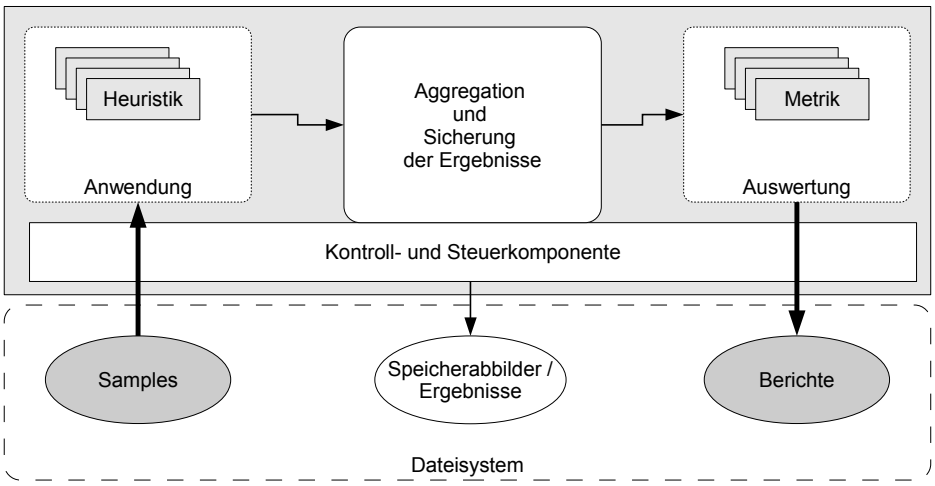

Abbildung 2: Das skizzierte Frameworkkonzept zur Anwendung von Heuristiken aus Unpacking-Frameworks und Auswertung deren Ergebnisse. Die in dem Bericht zusammengefassten Ergebnisse der Auswertung bieten die Grundlage zur Bewertung der Heuristiken.

tisieren, um die Auswertung einfach nachvollziehbar und die Ergebnisse leicht reproduzierbar zu machen. Wie im linken Teil der Abbildung 2, zu sehen werden die Heuristiken auf die Samples, in diesem Fall gepackte Programme, angewandt. Um dies zu realisieren müssen diese Programme instrumentiert werden. Das bedeutet, dass eigene Funktionen in den Programmfluss eingebaut werden müssen, ohne das Programm selbst oder dessen Funktionalität zu verändern. Die sich daraus ergebenden Anforderungen an die Instrumentation werden im Kapitel 4.2 erarbeitet. Die Ergebnisse der Heuristiken werden aggregiert und gesichert, wie in der Mitte der Abbildung 2 zu sehen ist. Dies hat den Vorteil, dass die Ergebnisse zu einem späteren Zeitpunkt analysiert und die Auswertung nachvollzogen werden kann. Die dabei ebenfalls gesicherten Speicherabbilder des analysierten Programmes werden der Auswertungskomponente zur Verfügung gestellt. Darin kommen Metriken zum Einsatz, die anhand der Speicherabbilder eine Aussage darüber treffen, ob der Entpackvorgang erfolgreich abgeschlossen wurde. Diese Auswertung wird im letzten Schritt in einem Bericht zusammengefasst, um die Grundlage zur Bewertung der Heuristiken zu schaffen.

Um die Heuristiken anwenden zu können müssen Annahmen gemacht werden, die im Kapitel 4.1 erarbeitet werden. Daraus ergeben sich Anforderungen an das Framework, insbesondere an die verwendete Instrumentationstechnik. Diese Anforderungen werden im Kapitel 4.2 diskutiert. Die darauf basierende Ausarbeitung des Frameworks sowie der zeitliche Ablauf einer Analyse wird im Kapitel 4.3 beschrieben. Die Heuristiken, die in dieser Arbeit angewandt und bewertet werden, sind im Kapitel 4.4 beschrieben. Die für die Auswertung benötigten Metriken werden im Kapitel 4.5 erarbeitet. Für die Evaluation wird ein Testdatensatz von gepackten Programmen benötigt. Die Auswahl dieser wird im Kapitel 4.6 besprochen.

4.1 Annahmen

Das Problem zu entscheiden, ob ein Programm gepackt ist oder nicht, ist nach dem Kapitel 2.3 nicht entscheidbar [13]. Um für eine Teilmenge der Programme dies lösen zu können, müssen Annahmen gemacht werden, um die Heuristiken anwenden zu können. Die Spezialfälle decken den Großteil ab, der für die praktische Anwendung und damit für diese Arbeit relevant ist. In dieser Arbeit werden die folgenden Annahmen gemacht:

- Die grundlegende Annahme ist, dass das gepackte Programm mindestens einen Entpackerstub besitzt. In diesem Kontext wird weiter angenommen, dass das ursprüngliche Programm zunächst vollständig entpackt wird, bevor der Kontrollfluss an das Programm übergeben wird.
  Dies ist oft der Fall, da die Programme in der Regel unabhängig vom Packer entwickelt werden. Der Packer muss demnach bei der Anwendung generisch vorgehen und fertige Programme verarbeiten können. Daher ist es für diesen einfacher, das Programm als Ganzes zu packen und den entsprechenden Entpackerstub voran zu schreiben, wie im Kapitel 2.3 beschrieben. Eine kombinierte Entwicklung ist möglich, bedarf jedoch mehr Aufwand und wird daher seltener genutzt.

- Das gepackte Programm muss darüber hinaus ohne weitere Programme ausführbar sein und mindestens die selbe Funktionalität wie das ursprüngliche Programm bieten.
  In dem Kontext der Malwareentwicklung versucht der Malwareautor in der Regel eine große Verbreitung zu erreichen, weswegen er möglichst wenig Anforderungen an das zu infizierende System stellen möchte. Deswegen werden für diesen Anwendungsfall Packer gewählt die keine weiteren Programme auf dem Zielsystem benötigen. Des Weiteren soll die Funktionalität der Malware erhalten bleiben. Diese Eigenschaft wird ebenfalls bei der Wahl des Packers bedacht.

- Das Programm muss in eine oder mehrere Sektionen entpackt werden. Diese Einschränkung ist nicht konzeptbedingt, musste jedoch, um der zeitlichen Anforderung an die Arbeit gerecht zu werden, gemacht werden. Es ist ebenfalls möglich, dass der Entpacker das Programm in einen, mittels der API-Funktion "VirtualAlloc", reservierten Speicherbereich entpackt. Diese Fälle können in dieser Arbeit nicht betrachtet werden.

Die gemachten Annahmen sind notwendig, um die Heuristiken anwenden zu können und werden ebenfalls in den Unpacking-Frameworks aus dem Kapitel 3.1 gemacht, da in die-

sen die selben Heuristiken verwendet werden. Die Annahmen schränken die Menge der entpackbaren Programme dahingehend ein, dass diese noch mit vertretbarem Aufwand entpackt werden können. Eine Ausweitung dieser Menge würde eine steigende Komplexität der Heuristiken mit sich bringen. Daher sind die gemachten Annahmen ein Mittelweg zwischen der Einschränkung und der Komplexität.

4.2 Anforderungen

Das in dieser Arbeit entwickelte Framework zur Evaluation von Heuristiken in Unpacking-Frameworks stellt einige Anforderungen an die Techniken, die es nutzt um Programme zu instrumentieren. Wie aus dem Kapitel 3.1 bekannt, stehen mehrere Techniken zur Verfügung, die anhand folgender Kriterien untersucht werden:

- **Instrumentation** Um die zu testenden Heuristiken anwenden zu können, muss die Technik in der Lage sein, Programme zu instrumentieren. Die Instrumentation muss dabei verschiedene Funktionalitäten bieten. Zum einen muss eigener Programmcode an beliebigen Stellen des instrumentierten Prozesses eingefügt werden können, ohne dabei die Funktionalität des Prozesses zu beeinträchtigen. Zum anderen muss die Technik eine Kontrollmöglichkeit über den Prozess bieten, um diesen zum Beispiel anzuhalten, beenden oder Speicherbereiche auslesen zu können.

- **Betriebssystem** Auf Grund der Tatsache, dass der Großteil der Packer für Microsoft Windows entwickelt werden, muss die eingesetzte Technik Windowsprogramme ausführen und instrumentieren können. Auch werden Packer oft im Kontext von Malware verwendet, dessen Verbreitung ebenfalls unter Microsoft Windows stärker ist.

- **API** Das Framework soll die Möglichkeit bieten Heuristiken anzupassen und neue implementieren zu können. Deshalb ist es wichtig, dass die zugrundeliegende Technik eine reichhaltige API bietet. Darüber hinaus sollte die API Zugriff auf viele Laufzeitaspekte bieten, da die verschiedenen Heuristiken unterschiedliche Aspekte betrachten.

- **Granularität** Eine effiziente Ausführung ist ein weiterer wichtiger Punkt. Deshalb ist es notwendig, dass die Instrumentationstechnik verschiedene Granularitäten bietet, um nur so oft wie nötig die Ausführung zu unterbrechen. Sinnvoll sind dabei die Abstufungen Instruktionsebene, Basic-Blockebene und Funktionsebene, die jeweils mögliche Stellen im Programmfluss bieten, um den eigenen Code einbauen zu können.

Es stehen mehrere Instrumentationstechniken zur Auswahl, darunter Bochs [2], Ether [4], Pin [12] und die Option einer kompletten Neuentwicklung. Diese Möglichkeiten sind anhand der Anforderungen zu bewerten, um eine geeignete Instrumentationstechnik für die Evaluation zu wählen.

Bochs bietet durch die vollständige Hardwareemulation der Maschine viele Möglichkeiten, Prozesse zu instrumentieren. Des Weiteren gibt es kaum Einschränkungen, bei der Wahl des Betriebssystems. Es kann insbesondere das für diese Analyse notwendige Microsoft Windows genutzt werden. Eine reichhaltige API, um die Heuristiken zu implementieren,

fehlt in Bochs. Diese ist jedoch als Grundlage für das Framework notwendig und muss zuvor entwickelt werden. Diese Instrumentationtechnik bietet kaum Granularität und fokussiert sich auf Basic-Blockebene. Dies kann bei intensiveren Berechnungen, wie der Entschlüsselung, von Nachteilen sein.

Das Framework Ether kann durch die Hardwarevirtualisierung Performancevorteile bringen, da die Prozesse direkt von dem Prozessor ausgeführt werden. Die Hardwarevirtualisierung mit der Erweiterung durch Ether bietet die grundlegende Möglichkeit zur Instrumentation, dabei können beliebige Betriebssysteme genutzt werden. Die bereit gestellte API bietet jedoch zu wenig Möglichkeiten, um die Heuristiken effizient implementieren zu können. Es wird von Ether die Möglichkeit geboten, auf Instruktionsebene und System-Call-Ebene zu instrumentieren, jedoch fehlen die Basic Block- und Funktionsebenen. Dies bringt trotz der Hardwarevirtualisierung große Performanceeinbußen mit sich, da jede durch die Instrumentation eingefügte Funktion das Pausieren des gesamten analysierten Betriebssystem bedeutet. Eine feinere Granularität würde diesen Nachteil mindern.

Eine komplette Neuentwicklung hat den Vorteil, dass die benötigte API zur Verfügung gestellt werden könnte. Bei der Wahl des Betriebssystems ist man dabei nicht eingeschränkt und kann insbesondere das benötigte Microsoft Windows unterstützen. Es kann auch die benötigte Granularität der Instrumentation geschaffen werden, in dem die API dahingehend ausgebaut wird. Jedoch wird eine komplette Neuentwicklung einer Instrumentationstechnik der zeitlichen Anforderung an diese Arbeit nicht gerecht.

Eine nutzbare Alternative stellt Pin [12] dar. Es bietet ein Framework um Programme zu instrumentieren, dabei ist es sowohl unter Microsoft Windows wie unter Linux lauffähig. Pin nutzt keine Virtualisierung, sondern wird direkt in dem nativem Betriebssystem zusammen mit dem zu analysierenden Programm gestartet. Die Instrumentation wird dabei mithilfe der Binary Translation realisiert, die im Kapitel 2.5 beschrieben wurde. "*Dabei wird das Programm nicht statisch umgeschrieben, sondern es wird dynamisch während der Ausführung des Programmes Code hinzugefügt* [12]". Der hinzuzufügende Programmcode und der Algorithmus, der zur Laufzeit bestimmt an welchen Stellen der Code eingefügt werden soll, werden in einem Pintool gespeichert. Die Pintools werden in der Programmiersprache C/C++ geschrieben und werden zur Laufzeit von Pin in den zu analysierenden Prozess eingefügt.
Dabei wird in dem jeweiligen Pintool festgelegt, bei welchen Ereignissen der implementierte Algorithmus aufgerufen werden soll. Dieser entscheidet dann, ob das Ereignis relevant ist und somit der Programmcode an dieser Stelle eingefügt werden soll. Bei der Auswahl der Ereignisse bietet Pin die oben geforderte Granularität. Ein mögliches Ereignis ist zum Beispiel die Ausführung einer Instruktion. Der Algorithmus kann dann entscheiden, ob diese Instruktion interessant ist und dementsprechend der eigenen Programmcode einfügt wird. Darüber hinaus kann der Programmcode vor oder hinter der Instruktion eingefügt werden. Weitere Abstufungen, die von Pin geboten werden, sind die Basic-Block- und Funktionsebenen. Damit ist die Anforderung an die Granularität von Pin erfüllt.
Der in dem Pintool implementierte Algorithmus wie auch der Programmcode kann dabei auf eine reichhaltige API von Pin zurückgreifen. Dies bedeutet zum Beispiel, dass sich Instruktionen einfach kategorisieren lassen, was eine effiziente Implementierung der Heuristiken ermöglicht. Darüber hinaus bietet Pin eine einfache Möglichkeit, auf den Speicher des Prozesses zuzugreifen, was für das Erstellen des Speicherabbildes, wie im Kapitel 4.3.3

beschrieben, nötig ist.
Pin erfüllt alle in diesem Kapitel erarbeiteten Anforderungen und kann somit als Instrumentationstechnik in dem Framework genutzt werden.

4.3 Framework

Das in dieser Arbeit erstellte Framework soll die Grundlage bieten, um einerseits Heuristiken zum Entpacken anzuwenden und zum anderen die Möglichkeit bieten diese hinsichtlich ihres Erfolges bewerten und untereinander vergleichen zu können. Die grundlegenden Funktionalitäten, die das Framework bieten muss, sind:

- Die Möglichkeit der Programminstrumentation, um die Heuristiken anzuwenden.
- Die Kontrolle über das zu analysierende Programm behalten.
- Die Ergebnisse der Heuristiken und die Speicherabbilder zu sichern.
- Die Metriken auf die Speicherabbilder anwenden und in der Auswertung aufzubereiten.

Um dies zu erreichen wurde das in diesem Kapitel beschriebene Framework implementiert. Damit ist die Möglichkeit geschaffen verschiedene Heuristiken effizient bewerten zu können.

Die Anwendung der Heuristiken bedarf einer Instrumentationstechnik, die im Kontext des Frameworks genutzte wird und im Kapitel 4.3.1 beschrieben wird. Die Architektur des Frameworks und damit das Zusammenspiel der einzelnen Komponenten wird im Kapitel 4.3.2 beschrieben. Die Komponente zur Steuerung der Abläufe befindet sich im Kapitel 4.3.3. Der Aufbau der Analyseergebnisse, die Grundlage für die spätere Auswertung sind, wird im Kapitel 4.3.4 beschrieben.

4.3.1 Instrumentation

Die Kernfunktionalität des Framework ist die Anwendung der Heuristiken. Dazu wird das Instrumentationsframework Pin [12] eingesetzt, wie in Kapitel 4.2 diskutiert. Das Betriebssystem, in dem sowohl das Framework als auch Pin läuft, ist ein Microsoft Windows XP SP3 [17]. Es wird keine Hardwarevirtualisierung oder Hardwareemulation eingesetzt, sondern die Just-In-Time-Translation von Pin verwendet, wie in der Abbildung 3 zu sehen. Dabei wird eine Binary Translation auf das Programm angewandt, um so die Instrumentation zu erreichen. Die Heuristiken, die in Pintools implementiert werden, nutzen die Instrumentations-API von Pin, siehe Abbildung 3, um unter anderem Einfluss auf die Binary Translation zu nehmen. Der gesamte Analyseablauf wird durch das Framework konfiguriert und kontrolliert. Die Heuristiken sind in C++ geschrieben, liegen kompiliert als DLL vor und werden beim Starten durch Pin in den Adressraum des Programmes geladen. Eine genaue Beschreibung dieser Heuristiken findet sich in Kapitel 4.4.

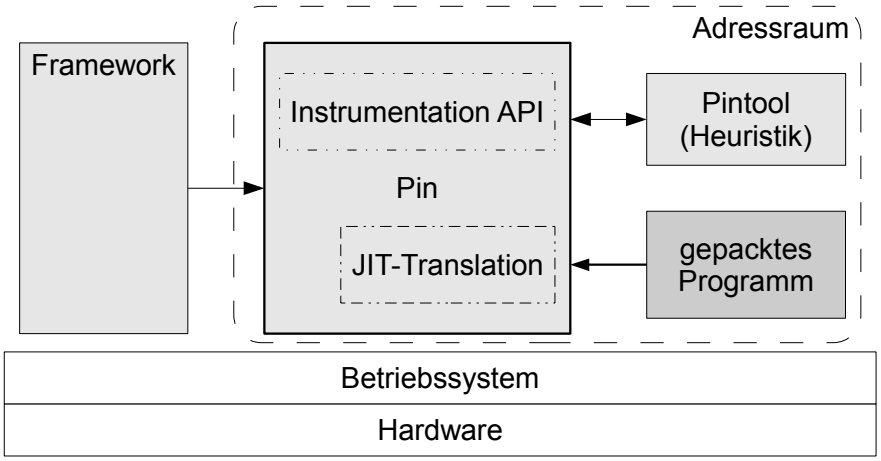

Abbildung 3: Zusammenarbeit des Frameworks mit Pin

4.3.2 Architektur

Die Architektur des Frameworks lässt sich in zwei logische Teile aufteilen. Der erste Teil ist für die Anwendung und Verwaltung der Heuristiken verantwortlich. Darüber hinaus sichert dieser Teil die Ergebnisse der Heuristiken und die Speicherabbilder der analysierten Prozesse. Der andere Teil verarbeitet die Speicherabbilder und wendet die Metriken an. Im Anschluss werden diese Analyseergebnisse aufbereitet und in einem Bericht zusammengefasst, um sie später zu interpretieren.

Das Framework ist in der Programmiersprache Python [20] geschrieben. An dieser Stelle wurde diese Programmiersprache gewählt, da sie einen großen Funktionsumfang mitbringt und sich sehr flexibel gestalten lässt. Darüber hinaus lassen sich auch größere Konzepte, wie das hier entwickelte Framework, schnell umsetzen. In dem Framework wird unter anderem die Konfiguration vorgenommen. Diese bestimmt die Auswahl der Programme und mit welchen Parametern die Heuristiken auf diese angewandt werden. Um dies zu erreichen sind drei funktionale Bestandteile notwendig: dem Userinterface, einem Projekt und den Heuristiken, siehe Abbildung 4.

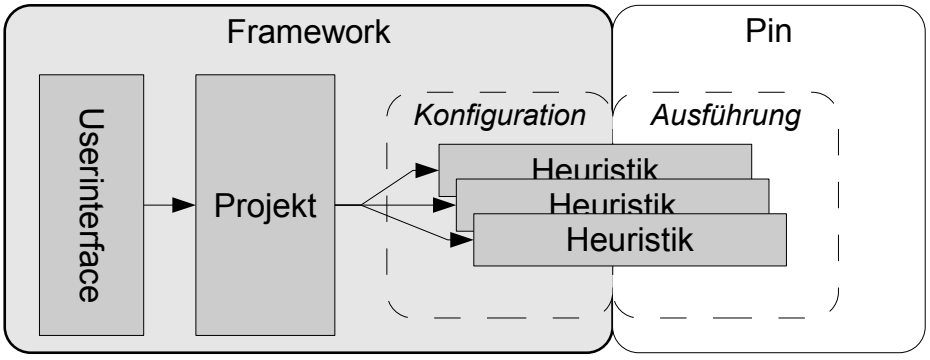

Abbildung 4: Der Aufbau der Frameworkkomponenten, die zur Konfiguration und Anwendung der Heuristiken notwendig sind.

Das Userinterface bietet die Möglichkeit eine neue Analyse, in der Abbildung Pro-

jekt genannt, zu konfigurieren und zu starten. Die spätere Durchführung erfolgt dabei vollständig autonom und endet mit der Sicherung der Ergebnisse und den Speicherabbilder. Diese werden bei der späteren Auswertung benötigt. Die Konfiguration der Analyse wird in dem Projekt vorgenommen und beinhaltet unter anderem die Auswahl der gewünschten Heuristiken. Darüber hinaus wird darin festgelegt, auf welches Programm die Heuristiken angewendet werden sollen. Die Heuristiken sind funktional in zwei Teile aufgeteilt, siehe Abbildung 4. Die Konfiguration der Heuristik, das Festlegen der Parameter, wird im Framework vorgenommen. Die eigentliche Ausführung geschieht in einem Pintool, also im Kontext des Pin-Frameworks.

Die Anwendung der Metriken auf die Analyseergebnisse findet in einem anderen Teil des Frameworks statt. Dabei besitzt dieser Teil zwei funktionale Kernaspekte, siehe Abbildung 5, die den Aufbau bestimmen. Dies ist zum einen die Aufbereitung der Analyseergebnisse und zum anderen die Bewertung der Speicherabbilder durch die Metriken. Das

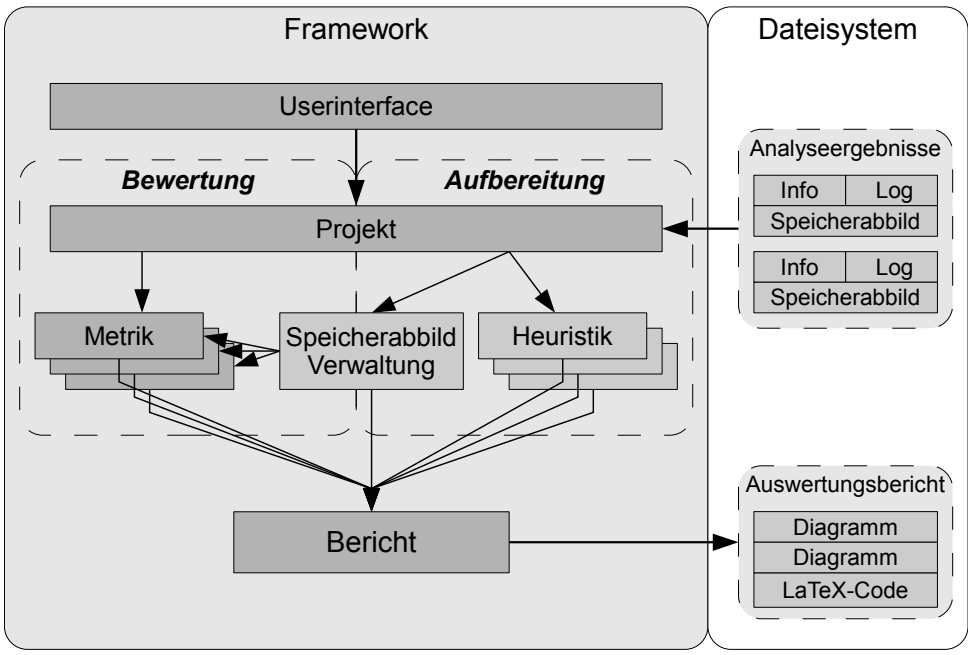

Abbildung 5: Der Aufbau des Frameworks zur Bewertung der Heuristiken

Userinterface bietet die Möglichkeit das Projekt zu konfigurieren. Dabei wird insbesondere festgelegt, welches Analyseergebnis zur Bewertung herangezogen wird und welche Metriken angewandt werden sollen. Der Aufbau des Analyseergebnisses wird im Kapitel 4.3.4 beschrieben. Das Analyseergebnis legt implizit fest, welche Heuristiken durch das Projekt instantiiert werden, um die Ergebnisse aufzubereiten, da jedem Analyseergebnis die entsprechenden Heuristiken zugeordnet sind. Die Speicherabbild-Verwaltung bereitet die bei der Analyse gesicherten Speicherabbilder auf und bietet den Metriken eine Schnittstelle auf diese. Somit lassen sich die Metriken effizient anwenden und die Speicherabbilder auswerten.

Die Ergebnisse und Bewertungen der Metriken sowie Informationen, die von der Speicherabbild-Verwaltung und den Heuristiken generiert wurden, werden an die Bericht-Klasse weitergeleitet. Der darin zusammengestellte Auswertungsbericht wird daraufhin

gesichert. Ebenfalls werden die durch die Heuristiken und Metriken generierten Diagramme gesichert, um sie im Auswertungsbericht zu integrieren.

Die Nutzung der beiden beschriebenen Teile, den zur Anwendung der Heuristiken und den zur Bewertung durch die Metriken, sind zeitlich nachgelagert. Eine Beschreibung des Ablaufs einer Analyse findet sich im Kapitel 4.3.3.

4.3.3 Ablauf

Das Framework bietet eine einfache und flexible Schnittstelle um verschiedene Heuristiken auf Programme anzuwenden. Dabei ist das Framework so aufgebaut, dass der hier beschriebene Analyseablauf vollautomatisch stattfindet.

Vorbereitungphase

In der Vorbereitungsphase wird mithilfe des Userinterfaces das Projekt konfiguriert. Dabei werden dem Projekt die Heuristiken sowie deren Konfigurationen übergeben. Darüber hinaus wird das Programm festgelegt, auf dem die Heuristiken angewandt werden sollen. Diese Einstellungen werden von dem Projekt in einer Logdatei gesichert, um die Analyse zu einem späteren Zeitpunkt nachvollziehen zu können. Daraufhin werden die benötigten Heuristiken vorbereitet und mit den jeweiligen Parametern konfiguriert. Damit ist die Vorbereitungsphase abgeschlossen.

Analysephase

In der Analysephase werden die Heuristiken angewendet. Dazu wird nacheinander für jeder Heuristik, ein eigener Prozess gestartet. Dieser Prozess wird durch den Teil der Heuristik konfiguriert und kontrolliert, der zum Framework gehört, siehe Abbildung 4. Der andere Teil der Heuristik, der als Pintool implementiert ist, übernimmt dabei die Kernfunktionalität der Heuristik. Dabei ist dieser Teil für die Beschaffung der benötigten Informationen über den Prozess sowie die Verarbeitung dieser Informationen zuständig. Darüber hinaus werden die Speicherabbilder durch diesen erstellt und gesichert, um im Anschluss die Auswertung durchführen zu können. Die Speicherabbilder werden für jeden Prozess zu zwei Zeitpunkten angefertigt. Zum einen beim Starten des Prozesses und zum anderen zu dem durch die Heuristik ermittelten Zeitpunkt, an dem das Programm erfolgreich entpackt sein soll. Mit der Sicherung der Konfigurationen, der Speicherabbilder und der Ergebnisse der Heuristiken ist die Analysephase abgeschlossen.

Die darauf folgende Auswertungsphase ist zeitlich entkoppelt und kann zu einem beliebigen späteren Zeitpunkt begonnen werden, da alle relevanten Informationen gesichert wurden.

Auswertungsphase

Zu Beginn der Auswertungsphase werden die gesicherten Speicherabbilder den einzelnen Heuristiken zugeordnet und für die Nutzung durch die Metriken aufbereitet. Dazu werden die Analyseergebnisse eingelesen, die unter anderem auch Informationen wie den vermuteten OEP enthalten. Diese Informationen werden dem Bericht ebenfalls beigefügt, um die spätere Interpretation zu unterstützen. Die Auswertung geschieht mithilfe der

Metriken. Diese werden nacheinander auf die Speicherabbilder angewandt. Die Metriken nutzen verschiedene Aspekte um festzustellen, ob das Programm entpackt in den Speicherabbildern vorliegt. Eine genaue Beschreibung der Metriken und welche Aspekte dabei genutzt werden, wird im Kapitel 4.5 beschrieben. Die Ergebnisse der Metriken werden zusammengefasst und in den Auswertungsbericht eingefügt.

Damit können die Ergebnisse der Metriken untereinander verglichen werden und Rückschlüsse auf die Effektivität der Heuristik gezogen werden können.

4.3.4 Analyseergebnis

Das Ergebnis der Heuristiken sowie die Speicherabbilder sind nach einem Analysedurchlauf in dem Log-Verzeichnis des Frameworks zu finden. Dabei verteilt sich das Ergebnis auf mehrere Dateien, siehe Abbildung 6.

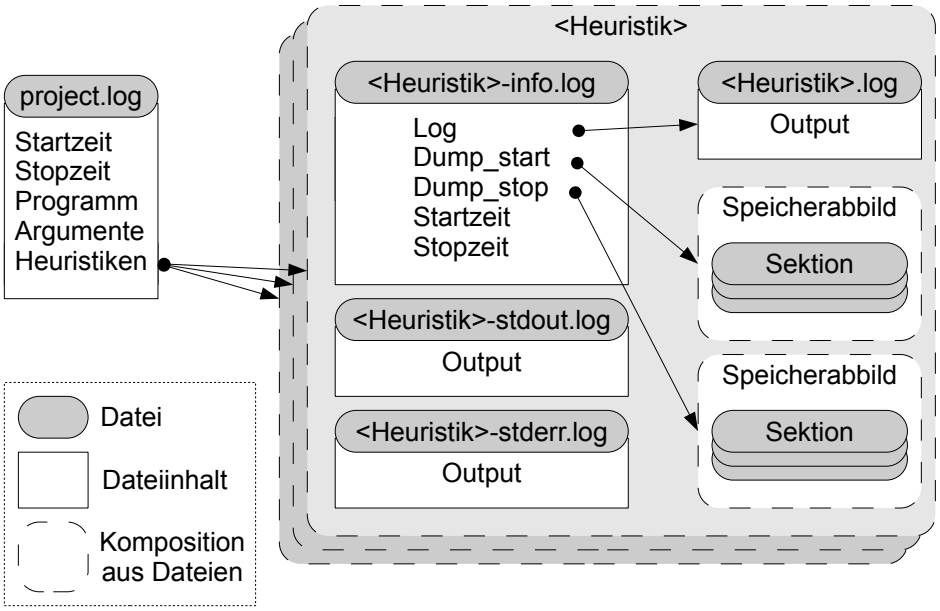

Abbildung 6: Der Aufbau der Analyseergebnisse, die nach der Anwendung der Heuristiken auf ein Programm gesichert werden.

In dem Ergebnis werden insbesondere die verwendete Heuristiken mit deren Konfigurationen in der "project.log"-Datei festgehalten. Dies ist notwendig, um in der Auswertungsphase das jeweilige Ergebnis der Heuristik zuordnen zu können. Darüber hinaus werden die Konsolenausgaben der Prozesse in den Dateien "$\langle heuristik \rangle$-stdout.log" und "$\langle heuristik \rangle$-errout.log" festgehalten, da diese in der Evaluation benötigt werden. Die Ausgaben der Heuristiken werden in der "$\langle heuristik \rangle$.log"-Datei gesichert. Darin werden die Informationen festgehalten, die für die jeweilige Heuristik ausschlaggebend sind. Des Weiteren werden die Speicherabbilder der instrumentierten Prozesse gesichert. Die Speicherabbilder enthalten den Inhalt der im instrumentierten Prozess genutzten Sektionen, wobei jede Sektion in einer eigenen Datei gesichert wird. Darin ist, sofern die Heuristik richtig entschieden hat, das entpackte Programm enthalten.

Es gibt für jeden erfolgreich beendeten Prozess zwei solcher Speicherabbilder. Das erste wird direkt beim Starten erstellt. Das zweite zu dem Zeitpunkt, an dem die jeweilige

Heuristik das Ende des Entpackerstubs vermutet und der Prozess damit beendet wird. Die im Anschluss durchgeführte Auswertung durch die Metriken basiert nur auf den Informationen aus den Speicherabbildern.

4.4 Heuristiken

Die Heuristiken sind ein entscheidender Bestandteil eines Unpacking-Frameworks. Diese versuchen anhand von Informationen, die durch die statische und dynamische Analyse gewonnen werden, den Ausführungszeitpunkt zu bestimmen, an dem der Entpackerstub das Entpacken abgeschlossen hat und damit das original Programm entpackt im Speicher vorliegt. Da dieses Problem im Allgemeinen, nach Kapitel 2.3, nicht entscheidbar ist [3], müssen Annahmen, wie im Kapitel 4.1 aufgestellt, gemacht werden, um Spezialfälle lösen zu können.

Die Heuristiken finden sich in dieser Arbeit an zwei Stellen wieder, siehe Abbildung 4 auf Seite 18. Zum einen als Implementation in einem Pintool, damit direkt in dem Prozess des verarbeiteten Programmes gearbeitet werden kann. Dies ist notwendig, um die benötigten Laufzeitinformationen performant erzeugen und verarbeiten zu können. Dazu gehört unter anderem die Erstellung eines Speicherabbildes zum Beginn des Prozesses. Zum anderen wird jede Heuristik als eigene Klasse im Framework abgebildet. Konfiguriert werden die Heuristiken durch ein Projekt, wie im Kapitel 4.3.3 beschrieben.

Im Folgenden finden sich die Heuristiken, die in dieser Arbeit mit Hilfe des Frameworks angewandt und bewertet werden.

4.4.1 Hump-and-Dump

Hump-and-Dump ist eine Heuristik die darauf basiert, dass die Entpackfunktionalität in einer Schleife implementiert wurde. Diese Schleife wird anhand der mehrfachen Ausführung einer Speicherstelle erkannt. Dazu wird ein Schwellenwert der notwendigen Ausführungen festgelegt, der bei einer Überschreitung die jeweilige Speicherstelle als Teil einer Schleife klassifiziert. Der Schwellenwert ist dabei ein Parameter der Heuristik. Als Schwellenwert wurde in dieser Arbeit der Wert 30000 gewählt, da dieser auch von den Autoren des Papers "Hump-and-dump: efficient generic unpacking using an ordered address execution histogram" [27] gewählt wurde. Sobald zur Laufzeit eine Schleife gefunden wird, versucht die Heuristik das Ende der Schleife zu erkennen. Dazu wird ein weiterer Schwellenwert genutzt, der ebenfalls ein Parameter der Heuristik ist. Dieser gibt eine Anzahl von hintereinander nur einmal ausgeführten Speicherstellen an. Hier wurde der Wert 20, wie auch in dem Paper [27], verwendet. Falls beide Bedingungen erfüllt werden, wird diese Stelle als das Ende des Entpackvorganges markiert.

4.4.2 NtTerminateProcess Aufruf

Diese Heuristik macht den vollständigen Entpackvorgang anhand des Windows API-Funktionsaufrufs von "NtTerminateProcess" [16] fest. Es wird dabei davon ausgegangen, dass sich das eigentliche Programm mit diesem Aufruf beendet [22]. Das heißt insbesondere, dass das Programm entpackt wurde. Im Kontext der Malwareanalyse bedeutet dies jedoch auch, dass die Malware komplett ausgeführt wurde und sehr wahrschein-

lich Schadfunktionen ausgeführt hat. Darüber hinaus ist eine Malwareanalyse dadurch benachteiligt, dass eine Vielzahl an Malware sich nicht selbstständig beendet.

Obwohl diese Heuristik einfach anzuwenden ist, kann sich die Implementation wesentlich umfangreicher gestalten, da einige Packer Methoden verwenden um API-Funktionsaufrufe so zu gestalten, dass sie nicht erkannt werden [22]. Wenn die Implementation dieser Heuristik von der Verschleierungstaktik des Packers betroffen ist, werden die jeweiligen Aufrufe nicht erkannt und die Heuristik schlägt fehl.

4.4.3 NtCreateProcess Aufruf

In einer Vielzahl an Malware findet sich nach der Entpackfunktion die Ausführung der API-Funktion NtCreateProcess [22]. Damit wird oft versucht möglicherweise angehängte Debugger von der Malware zu trennen. Dieser Aufruf kann genutzt werden, um das Ende des Entpackens zu bestimmen. Da der Aufruf jedoch nicht zwangsweise am Anfang bzw. überhaupt notwendig ist, kann diese Heuristik nicht immer erfolgreich sein.

Darüber hinaus muss bei der Implementation beachtet werden, dass einige Packer API-Funktionsaufrufe versuchen zu verstecken [22]. Wenn die Verschleierungsmethode des Packers nicht durch die Implementation der Heuristik abgedeckt ist, werden die jeweiligen Aufrufe nicht erkannt und die Heuristik schlägt fehl.

4.4.4 LoadLibrary/GetProcAddress Aufruf

LoadLibrary und GetProcAddress sind zwei Windows API-Funktionen, die genutzt werden, um weitere DLLs in einen Prozess zu laden und Zugriff auf die darin exportierten Funktionen zu erhalten. Dies kann zum einen von dem gepackten Programm selbst genutzt werden, um den eigenen Funktionsumfang zu erweitern oder vom Entpackerstub, um die IAT wieder aufzubauen [31]. Die IAT beinhaltet die tatsächlichen Speicheradressen, an die Funktionen aus den DLLs geladen wurden, um deren Nutzung durch das Programm zu ermöglichen. Das Befüllen dieser geschieht in der Regel durch das Betriebssystem. Viele Packer nehmen dies jedoch selber vor, um die statische Analyse zu erschweren, da somit nicht direkt erkennbar ist, welche DLLs geladen werden. Dies bedingt, dass der Entpackerstub die DLLs selber lädt und die IAT befüllt, wofür die beiden API-Funktionen genutzt werden. Daher kann der Aufruf dieser beiden Funktionen als Heuristik herangezogen werden.

In dieser Arbeit wurde für jede dieser Funktionen eine eigene Heuristik erstellt, da der Aufruf dieser nicht zwangsweise in Kombination erfolgen muss. Beide Heuristiken nutzen einen Schwellenwert, der die Anzahl an Aufrufen festlegt. Dieser wird als Parameter durch das Framework übergeben. In dieser Arbeit wird der Schwellenwert für die LoadLibrary-Heuristiken auf zwei Aufrufe festgelegt, da alle verwendeten Samples mindestens zwei DLLs nutzen. Für die GetProcAddress-Heuristik wird der Wert auf drei Aufrufe festgelegt, da alle verwendeten Samples mehr als drei Funktionen aus externen DLLs nutzen.

Bei der Implementation der Heuristiken muss darauf geachtet werden, dass einige Packer API-Funktionsaufrufe versuchen zu verstecken [22]. Wenn die Verschleierungsmethode des Packers nicht durch die Implementation der Heuristik abgedeckt ist, werden die jeweiligen Aufrufe nicht erkannt und die Heuristik schlägt fehl. Ergänzend zu der

LoadLibrary-Funktion werden ebenfalls GetModuleHandle-Aufrufe beachtet. Diese Funktion wird genutzt um Referenzen auf DLLs zu erhalten, die schon in den Prozess geladen wurden.

4.4.5 WˆX Speicher

Die WˆX-Heuristik, "Write or eXecute", ist eine sehr einfache aber solide Methode. Daher wird sie in viele Unpacking-Frameworks, die im Kapitel 3.1 vorgestellt wurden, verwendet. Sie basiert auf der Tatsache, dass ein gepacktes Programm beim Entpacken in den Speicher geschrieben werden muss und bei der späteren Ausführung diese Speicherstellen ausgeführt werden. Dieser Zustandswandel lässt sich in einer Heuristik beobachten. Beim Prozessstart werden alle Speicherbereiche als "sauber" betrachtet. Jegliche Schreibzugriffe markieren die Zielspeicherstelle als "neu beschrieben". Wenn bei einer dieser als "neu beschrieben" markierten Speicherstellen ausgeführt wird, ist diese Stelle als Beginn des entpackten Programmes anzusehen. Dies bedeutet, nach den Annahmen in Kapitel 4.1, dass der Entpackerstub komplett durchlaufen wurde und das Programm entpackt im Speicher vorliegt.
Die Heuristik kann unter anderem durch folgende zwei Methoden nachteilig beeinträchtigt werden. Zum einen kann ein Speicherbereich neu beschrieben und schon vor der Entpackfunktion ausgeführt werden, was Falschmeldungen hervorrufen würde. Zum anderen kann derselbe physikalische Speicher an zwei unterschiedlichen virtuellen Adressen geladen werden, wobei der eine Bereich zum Schreiben und der andere zum Ausführen genutzt wird [24]. Da der Prozess und damit auch die Heuristik nur auf den virtuellen Adressen arbeitet, kann diese Verfahren nur schwer erkannt werden. Dieses Verfahren wird exemplarisch im Kapitel 5 beschrieben.

4.4.6 Stackpointer

Diese Heuristik nutzt den Stackpointer als Informationsgrundlage dafür, ob der OEP des original Programmes erreicht wurde und damit der Entpackerstub komplett durchlaufen wurde. Dabei basiert die Heuristik auf der Annahme, dass der Entpacker versucht dem Programm möglichst die selben Umgebungsbedingungen zu bieten, wie sie bei einer nativen Ausführung herrschen. Dies ist darauf zurückzuführen, dass der Packer sonst möglicherweise die Funktionsweise des Programmes beeinträchtigt [6].
Die Heuristik sieht vor, beim Starten des Prozesses den Stackpointer zu sichern. Dieser befindet sich in dem Register "%esp" bzw. "%rsp", einem Speicherbereich auf der CPU, und kann daher durch die Heuristik erfasst werden. Wenn zu einem späteren Zeitpunkt der Stackpointer wieder den gesicherten Wert annimmt, wird an dieser Stelle der OEP erwartet. Aus Performancegründen wird der Stackpointer nur am Ende eines Basic Block verglichen. Als Parameter erwartet die Heuristik noch eine Anzahl an Basic Blocks, bei denen zum Beginn kein Vergleich durchgeführt wird, um die Rate der Falschmeldungen an dieser Stelle zu senken. Der hier verwendete Schwellenwert wurde auf 10 festgelegt, da davon ausgegangen wird, dass der Entpackerstub innerhalb von 10 Basic Blocks mit seiner Funktionalität begonnen hat und kein Entpackerstub unter einer Länge von 10 Basic Blocks implementiert ist.

Die Heuristik kann dahingehend nachteilig beeinflusst werden, wenn der Entpackerstub schon vor der Ausführung der Entpackfunktion den Stackpointer wiederherstellt

oder der Stack gar nicht genutzt wird.

4.4.7 Kommandozeilen-Argumente

Die Kommandozeilen-Argumente werden genutzt, um beim Starten von Programmen Informationen zu übergeben, die zum Beispiel das Verhalten spezifizieren. Das Vorhandensein von Referenzen auf die Kommandozeilen-Argumente deutet auf den Beginn der Hauptfunktion eines Programmes hin, wenn diese als Funktionsparameter auf dem Stack übergeben werden. Daher kann das Vorhandensein der Referenzen in einer Heuristik genutzt werden, um den Beginn der Hauptfunktion eines neuen Programmes zu erkennen. Dabei geht die Heuristik davon aus, dass diese Kommandozeilen-Argumente bei einem gepackten Programm unmittelbar vor dem Starten des original Programmes als Parameter einer Funktion auf dem Stack zu finden sind [6]. Da die Argumente durch das Framework beim Start des Prozesses festgelegt werden, sind diese der Heuristik bekannt.

Aus Performancegründen wird der Stack nur am Ende jedes Basic Blocks auf das Vorhandensein der Kommandozeilen-Argumente geprüft. Bei der Implementation muss darauf geachtet werden, dass nicht die Referenzen, die sich auf dem Stack befinden, sondern die referenzierten Daten verglichen werden müssen.

4.4.8 Stacktiefe

Die Stacktiefe ist eine Eigenschaft eines Prozesses, die Anhaltspunkte darüber liefert, wie viele verschachtelte Funktionsaufrufe gemacht und wie stark der Stack genutzt wurde. Unter der Annahme, dass der Entpackerstub als Schleife implementiert wurde und damit nur eine geringe Stacktiefe erreicht, kann diese Eigenschaft als Heuristik herangezogen werden. Das Ende des Entpackvorganges wird dabei durch ein Überschreiten eines Schwellenwertes festgemacht. Dieser gibt die zu überschreitende Anzahl an Stackframes an. Der Schwellenwert wird in dieser Arbeit auf 10 festgelegt, da Entpackerstubs nur wenige Funktionalitäten benötigen und damit nur wenige verschachtelte Funktionsaufrufe machen.

Die Implementation dieser Heuristik berechnet die Stacktiefe anhand der vorhandenen Stackframes. Ein neuer Stackframe wird dabei durch die Ausführung der Instruktion "CALL" hinzugefügt und durch die Ausführung von "RET" wieder abgebaut.

4.5 Metriken

Die Metriken, die in der Auswertungsphase Verwendung finden, bewerten die Speicherabbilder aus den Analyseergebnissen dahingehend, ob das Speicherabbild das entpackte Programm enthält. Mit diesen Bewertungen als Grundlage lassen sich Rückschlüsse auf die Effektivität der Heuristiken schließen. Die Metriken bekommen bei der Anwendung jeweils zwei Speicherabbilder des Prozesses. Ein Abbild wurde zu Beginn des Prozesses erstellt, das zweite Abbild wurde jeweils zu dem Zeitpunkt erstellt, an dem die jeweilige Heuristik das Ende des Entpackerstubs vermutet. Die Auswertungen der Metriken werden in einem Bericht zusammengefasst. Im folgenden werden die verwendeten Metriken beschrieben.

4.5.1 Code-Daten Verhältnis

Das Speicherabbild eines Prozesses beinhaltet drei Arten von Speicher. Ein Teil des Speicherabbildes besteht aus nicht initialisierten Speicher. Der Rest des Abbildes besteht entweder aus Programmcode oder Nutzdaten. Das Verhältnis von Programmcode zu den Nutzdaten ändert sich beim Entpacken des Programmes dahingehend, dass sich im Speicherabbild mehr Programmcode befindet. Das ist darauf zurückzuführen, dass das gepackte Programm im Allgemeinen kein valider Programmcode ist und beim Entpacken daraus neuer valider Programmcode entsteht. Damit steigt der Anteil an Programmcode im Speicherabbild [22]. In dem Fall, dass dieser neu erzeugte Programmcode an die Stelle des gepackten Programmcodes, der als Nutzdaten zu klassifizieren ist, geschrieben wird, sinkt gleichzeitig auch der absolute Anteil an Nutzdaten in dem Speicherabbild. Mithilfe des Verhältnisses, insbesondere der Änderungsstärke zwischen den beiden Speicherabbilder, lassen sich Rückschlüsse darauf ziehen, ob und welche Bereiche des Programmes entpackt wurde.
Bei der Implementation der Heuristik wird eine Methode benötigt, die bei einem gegebenen Speicherbereich entscheiden kann, ob es sich um Nutzdaten, Programmcode oder nicht initialisierten Speicher handelt. Da das System auf einer Von-Neumann-Architektur basiert, lässt sich die Entscheidung nur anhand des Speicherinhaltes treffen. In dieser Arbeit wird die Entscheidung anhand des prozentualen Vorkommens dieser drei Opcodes "PUSH","MOV" und "CALL" getroffen. Diese Opcodes wurden gewählt, da sie prozentual am häufigsten vorkommen [1]. Darüber hinaus ist "MOV" ein Opcode, der sehr universell ist und daher in sehr vielen Bereichen eines Programmes genutzt wird. Auch die beiden anderen Opcodes finden sich in vielen Funktionen wieder und sind damit gut über den Adressraum des Programmes verteilt. Bei der Analyse wird in 1024 Byte Schritten über das Speicherabbild iteriert und der Abschnitt disassembliert. Sofern alle drei Vorkommen in einem jeweils vorher definierten Bereich liegen, wird dieser Speicherbereich als Programmcode markiert. Die Bereiche stammen aus einer vorher durchgeführten statistischen Erhebung über 200, mit Microsoft Windows XP SP3 mitgelieferten, Programme und liegen für "PUSH" und "MOV" bei 5%-25% und für "CALL" bei 5%-15%.
Diese Bewertung wird für beide Speicherabbilder durchgeführt und im Anschluss der prozentuale Anteil des Programmcodes in den Bericht eingefügt, der nicht initialisierte Speicher wird dabei nicht mitberechnet.

4.5.2 Entropieanalyse

Die Entropieanalyse berechnet für ein Speicherabbild oder für Teile davon einen Wert, der den mittleren Informationsgehalt widerspiegelt. Je höher die Informationsdichte in dem analysierten Bereich ist, umso höher ist dieser Entropiewert. Dabei wird zur Berechnung die von Shannon definierte Formel genutzt:

$$H(x) := -\sum_{i=0}^{255} p(i) * \log_{10}(p(i))$$

Dabei ist $p(i)$ gleich der Anzahl der Vorkommnisse des Symbols i geteilt durch die Größe des betrachteten Speicherbereichs.

Eine Komprimierung von vorher unkomprimierten Daten erzeugt, bis auf Sonderfälle, eine höhere Informationsdichte [8], wenn sie zum Beispiel zum Packen von Programmen

genutzt wird. Eine gute Verschlüsselung zeichnet sich dadurch aus, dass das verschlüsselte Ergebnis, auch Ciphertext genannt, einen hohen Entropiewert aufweist. Ein gepacktes Programm lässt sich daher oft anhand eines hohen Entropiewertes erkennen, da es vorher komprimiert oder verschlüsselt wurde. Wenn das Programm durch den Entpackerstub entpackt wird, lässt sich im Speicher eine Änderung des Entropiewertes feststellen [8].

Die Metrik erhält jeweils zwei Speicherabbilder und kann daher für beide den Entropiewert und somit die Veränderung berechnen. Dabei lassen sich die Entpacker in zwei Klassen von Typen einteilen. Der Typ 1 zeichnet sich durch einen fallenden Entropiewert beim Entpacken ab. Der Typ 2 weist einen steigenden Entropiewerte beim Entpacken auf. Die ist unter anderem davon abhängig, wohin der Entpacker die Daten schreibt und was an dieser Stelle vorher gespeichert war.

Diese Metrik führt eine Entropieanalyse auf zwei unterschiedliche Bereiche aus. Zum einen wird das gesamte Speicherabbild betrachtet. Hierbei ist zu beachten, dass möglicherweise nachgeladene DLLs sich nur im zweiten Speicherabbild befinden, da diese beim Starten des Prozesses noch nicht geladen waren. Das hat zur Folge, dass die Analyse verfälscht werden könnte. Daher wird zum anderen noch eine Analyse ausschließlich auf die Sektionen, die in der Programmdatei verfügbar waren, ausgeführt.

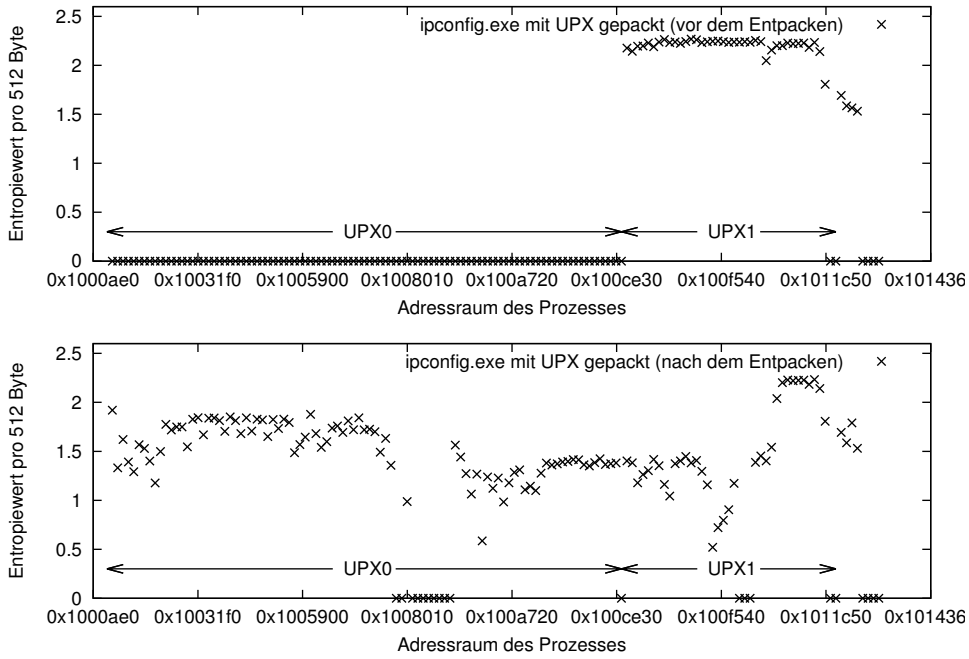

Abbildung 7: Die Entropieanalyse über die Sektionen UPX0 und UPX1 des Programmes "ipconfig.exe", das mit UPX [19] gepackt wurde. Die Sektionen wurden den Speicherabbilder entnommen, die vor und nach dem Entpacken erstellt wurden. Die Veränderung der Entropiewerte zeigt die Entstehung neuer Daten im Speicher.

Die Visualisierung der Entropieveränderung dient der besseren Interpretation, siehe Abbildung 7. Dabei wird der zu betrachtende Adressraum in jeweils 512 Byte große Bereiche unterteilt, für die jeweils der Entropiewert berechnet und im Diagramm eingetragen wird. Die Abbildung zeigt die Entropieanalyse für das Programm "ipconfig", welches zuvor mit dem Packer UPX [19] gepackt wurde. Dabei wurden nur die Sektionen betrachtet,

die aus diesem Programm stammen, da die geladenen DLLs in diesen Fall nicht relevant sind. Das obere Diagramm zeigt den Zeitpunkt vor dem Entpacken. Dabei ist zu erkennen, dass sich im vorderen Adressraum keine Daten befinden, da der Entropiewert dort 0 beträgt. Im unteren Diagramm, welches den Zustand nach dem Entpacken darstellt, wurden an diese Stelle Daten entpackt, da die Entropiewerte hier gestiegen sind.

4.5.3 Import Address Table - Größe

Die IAT ist ein Speicherbereich des Prozesses, in dem Funktionsadressen gespeichert werden. Sie wird genutzt um Funktionen aus DLLs aufzulösen und zu benutzen. Die Größe der IAT gibt Aufschluss darüber, wie viele Funktionen aus DLLs genutzt werden.

Die Import Table wird von den meisten Packern entfernt, um die Hinweise auf das enthaltende gepackte Programm zu verschleiern. Daher kann der dynamische Linker die IAT nicht befüllen, da dafür die Informationen aus der Import Table benötigt werden. Deshalb muss beim Entpacken die IAT wieder rekonstruiert werden, damit das enthaltene Programm die Funktionen, die in der IAT referenziert werden, nutzen kann. Daraus folgt, dass ein Änderung der IAT während des Entpackvorganges zu erkennen ist.

Um die Größe der IAT bestimmen zu können, muss diese zuvor in dem Speicherabbild lokalisiert werden. Dazu wird eine Methode angewandt, die den typischen Aufbau einer IAT nutzt. Diese Methode basiert darauf, dass in der IAT Funktionsadressen gespeichert werden, wobei die Adressen der Funktionen, die aus der selben DLL stammen, in der IAT meist direkt hintereinander liegen. Die jeweiligen Mengen an Funktionen, die aus den selben DLLs kommen, liegen jeweils in den selben Sektionen. Weshalb die Adressen der jeweiligen Funktionen mit einer sehr hohen Wahrscheinlichkeit mit dem selbe Byte beginnen. Dazu kommt, dass die Funktionsadressen, unter Annahme einer typischen 32-Bit Windows-Architektur, vier Byte lang sind und in der IAT direkt aufeinander folgen. Das führt zu einem Muster, bei dem in vier Byte Schritten mehrfach hintereinander das selbe Byte auftritt. Dies ist in der Abbildung 8 zu erkennen. Die entsprechenden Bytes sind unterlegt.

Die Metrik setzt diese Methode ein, um Sequenzen dieser Art zu finden, woraus sich ableiten lässt, inwiefern sich die IAT verändert hat. Weiterhin werden mit dieser Methode auch Switch-Case-Konstrukte gefunden, je nachdem wie diese vom Compiler übersetzt wurden. Das Differenzieren muss später manuell vorgenommen werden. Dabei können die möglichen Funde von Switch-Case-Konstrukten als Indizien genutzt werden, um neu entstandenen Code zu erkennen.

4.5.4 Import Address Table - Nutzung

Bei dieser Metrik kommen ähnliche Aspekte wie bei der in Kapitel 4.5.3 vorgestellten Metrik zum tragen. Die IAT beinhaltet Funktionsadressen, um die Funktionen aus den DLLs zu nutzen. Eine gestiegene potentielle Nutzung der IAT weist auf mehr Funktionalität und damit auf das Vorhandensein des entpackten Programmes hin.

Im vorhergehenden Unterkapitel 4.5.3 wurde versucht die IAT zu finden und dessen Größe zu bestimmen, um eine mögliche Veränderung als Indiz zu nutzen. Wenn die IAT von Beginn an vollständig ist, wird eine solche Veränderung nicht feststellbar sein. In beiden Fällen ist aber eine gesteigerte Häufigkeit der potentiellen Nutzung dieser festzustellen. Da insbesondere das entpackte Programm die IAT nutzt, finden sich hier viele

Adresse	Funktionsadresse	Funktionsname
0x01001008	0x 77 da189a	...
0x0100100c	0x 00 000000	...
0x01001010	0x 77 c41e2e	GDI32.dll : SetBkColor
0x01001014	0x 77 c41d83	GDI32.dll : SetTextColor
0x01001018	0x 77 c41eff	GDI32.dll : SetBkMode
0x0100101c	0x 00 000000	
0x01001020	0x 77 e59f93	Kernel32.dll : GetModuleHandleA
0x01001024	0x 77 e605d8	Kernel32.dll : LoadLibraryA
0x01001028	0x 77 e5a5fd	Kernel32.dll : GetProcAddress
0x0100102c	0x 77 e7a9ad	Kernel32.dll : GlobalCompact
0x01001030	0x 77 e536a3	Kernel32.dll : GlobalAlloc
0x01001034	0x 77 e53803	Kernel32.dll : GlobalFree
0x01001038	0x 77 e4e341	Kernel32.dll : GlobalReAlloc
0x0100103c	0x 77 e58d60	...
0x01001040	0x 77 e41be6	...

Abbildung 8: Der Aufbau einer IAT am Beispiel von calc.exe. Die selten wechselnde führende Byte bildet ein markantes Merkmal für die IAT.

Verweise auf die IAT, die im gepackten Zustand nicht zu erkennen sind. Eine stärkere potenzielle Nutzung ist daher ein Indiz dafür, dass das Programm entpackt vorliegt.

Die potenzielle Nutzung der IAT wird anhand von gefundenen Verweisen in dem Speicherabbild festgemacht. Dabei wird das Abbild disassembliert und indirekte "JMP"- sowie indirekte "CALL"-Instruktionen ausgewählt und die darin genutzten Referenzwerte aufgelistet, siehe Abbildung 9. Die Referenzwerte sind Adressen, die auf IAT-Einträge zeigen, wie in der Mitte der Abbildung zu erkennen. Die Einträge wiederum beinhalten Referenzen auf die auszuführenden Funktionen in den DLLs. In dieser sortierten Liste der Referenzwerte werden Häufungspunkte gesucht, bei denen die benachbarten Adressen nicht weiter als ein festgelegter Schwellenwert, in diesem Fall 20 Byte, voneinander entfernt liegen. Die Anzahl und die Häufigkeit der genutzten IAT-Einträge werden durch diese Metrik aufbereitet und ausgewertet.

4.5.5 Funktionsanzahl

Mit einer steigenden Größe eines Programmes und der damit zusätzlichen Funktionalität steigt auch die Anzahl an enthaltenden Funktionen. Eine Funktion ist eine zusammengehörige Menge von Instruktionen, die eine bestimmte Funktionalität abbilden. Dabei ist diese durch eine definierte Art und Weise, die "Calling-Convention", einfach zu nutzten. Daher besitzen diese Funktionen am Anfang sowie am Ende ein Codeteil, um dieser "Calling-Convention" zu entsprechen.

Diese Metrik nutzt diese Codeteile um Funktionen zu erkennen. Dabei lässt sich das entpackte Programm daran erkennen, dass neue Funktionen im Speicherabbild gefunden werden. Es werden dafür zwei einfache Signaturen von Instruktionen verwendet, die in diesen Codeteilen in der Regel vorkommen. Der Funktionsanfang wird dabei an der Instruktion "PUSH %ebp" erkannt. Diese Instruktion wird zum Sichern des alten Funk-

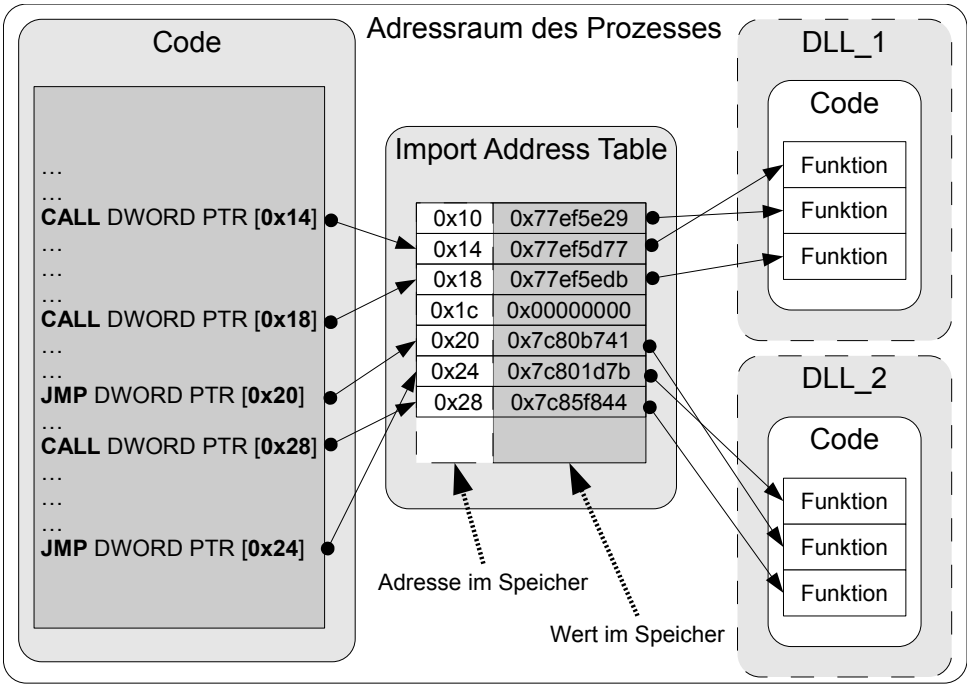

Abbildung 9: Die Verweise im Programmcode auf die IAT zeigen die potentielle Nutzung dieser an.

tionskontextes verwendet, damit dieser am Ende der Funktion wieder hergestellt werden kann. Dies ist notwendig, damit die aufrufende Funktion weiter ausgeführt werden kann. Das Ende einer Funktion wird an den Instruktionen "RET", "RETN", "RETF" festgemacht. Diese Instruktionen beenden eine Funktion und springen in die vorherige Funktion zurück. Die Anzahl der Vorkommnisse dieser beiden Signaturen wird durch die Metrik ermittelt und ausgewertet.

4.6 Testdatensatz

Die Evaluation der Heuristiken wird auf der Basis von selbst erstellten Samples durchgeführt. Dabei wird das im Kapitel 4.3 konzipierte Framework eingesetzt. Die abschließende Bewertung der Ausgaben wird manuell durchgeführt und die Ergebnisse der einzelnen Durchläufe werden aggregiert.

Um diese Basis zu schaffen werden fünf Programme mit jeweils sieben verschiedenen Packern gepackt und analysiert. Die Programme sind jeweils einfache Konsolenprogramme, die sich ohne Benutzerinteraktion selbstständig beenden. Dies ermöglicht eine einfach vollautomatische Durchführung der Analyse. Die verwendeten Programme sind zum einen "ipconfig.exe", "nslookup.exe" sowie "tracerpt.exe", die mit Microsoft Windows XP SP3 mitgeliefert werden, und zum anderen "md5sum.exe" sowie "sha1sum.exe", die vom GnuPG Project [23] bereit gestellt werden. In der Tabelle 1 finden sich die verwendeten Packer. Diese werden mit der jeweiligen Standardeinstellung auf die Programme angewandt.

Die Menge der generierten Samples lässt sich wie folgt definieren:

$$S := P \times B \qquad \text{mit } P := \{\text{Packer}\}, B := \{\text{Programme}\}$$

- ASPack 2.28 (Demo Version) [11]
- FSG 2.0 [5]
- mew11 SE v1.2 [18]
- MPRESS v2.18 [25]
- PECompact 3.02.2 (Demo Version) [29]
- Upack-Optimizer 1.2 [14]
- UPX 3.07w [19]

Tabelle 1: Die Liste der verwendeten Packer, die zur Generierung der Samples genutzt wurden.

Jedes Sample wird bei der Analyse drei Mal mit jeder Heuristik ausgeführt, um mögliche Fehler erkennen zu können. Die Ergebnisse der einzelnen Durchläufe werden aggregiert und dieses Ergebnisse weiter genutzt. Es ist davon auszugehen, dass die jeweiligen Durchläufe sehr ähnliche Ergebnisse liefern, weshalb der Informationsverlust durch die Aggregation sehr gering ist. Die aggregierten Ergebnisse der einzelnen Heuristiken für ein Sample werden miteinander verglichen und dahingehend bewertet, ob das Programm vollständig entpackt wurde und damit die Heuristik einen korrekten Zeitpunkt bestimmt hat.

4.7 Zusammenfassung

Das in diesem Kapitel konzipierte Framework schafft die Möglichkeit Heuristiken aus Unpacking-Frameworks effizient anzuwenden und zu bewerten. Dabei wird die Menge der zu bewertenden Heuristiken, mithilfe des Instrumentationsframeworks Pin [12], auf gepackte Programme angewandt und die Ergebnisse gesichert. Die verschiedenen Heuristiken nutzen dabei sehr unterschiedliche Strategien, wie zum Beispiel die Analyse des Speichernutzungsverhaltens und die Beobachtung des Programmflusses. Auch der einfache Aufruf einer Funktion wird in einigen Heuristiken als Indiz genutzt, um das fertige Entpacken festzustellen. Die Anwendung der implementierten Heuristiken und die Analyse läuft dabei automatisiert ab und wird durch das Framework kontrolliert. Bei der Auswertung werden verschiedene Metriken eingesetzt, die jeweils unterschiedliche Aspekte betrachten und so eine gute Informationsgrundlage bieten, um die Effektivität der Heuristiken bewerten zu können.

5 Evaluation

In dieser Arbeit sollen die Heuristiken in Unpacking-Frameworks evaluiert werden. Dies geschieht in zwei zeitlich nachgelagerten Schritten. Im ersten Schritt werden die Heuristiken aus dem Kapitel 4.4 auf die gepackten Samples angewandt und die Ergebnisse gesichert. Im zweiten Schritt werden die Metriken aus dem Kapitel 4.5 genutzt, um diese Ergebnisse auszuwerten. Die Auswertung gibt Aufschluss darüber, ob die jeweilige Heuristik erfolgreich war und damit das ursprüngliche Programm erfolgreich entpackt wurde. Der genaue Ablauf dieses Verfahrens wurde in Kapitel 4.3.3 beschrieben. Die Evaluation wird mithilfe des Frameworks erreicht, das im Kapitel 4.3 konzipiert wurde und die automatisierte Durchführung dieser Schritte ermöglicht.

Die anschließend manuell durchgeführte Bewertung der Heuristiken basiert auf den Ausgaben der Metriken, die in der Auswertungsphase, wie im Kapitel 4.3.3 beschrieben, angewandt wurden. Dabei werden durch die Metriken unterschiedliche Aspekte betrachtet, die durch eine Interpretation zusammengefasst werden und damit eine Aussage darüber liefern, ob das Entpacken erfolgreich war. Zusätzlich können die gesicherten Speicherabbilder herangezogen und manuell auf das Vorhandensein des entpackten Programmes untersucht werden. Dies ist nur dann notwendig, wenn das Bild, das sich aus den Auswertungen der Metriken ergibt, nicht eindeutig erkennen lässt, ob das Entpacken erfolgreich war. Als Testdatensatz wird eine Menge an Samples genutzt, die sich selbstständig beenden. Die Zusammensetzung des Testdatensatzes wurde im Kapitel 4.6 beschrieben. Das selbstständige Beenden hat den Vorteil, dass die "NtTerminateProcess"-Heuristik, wie im Kapitel 4.4.2 beschrieben, in jedem Fall einen Zeitpunkt bestimmt und ein Speicherabbild erstellt. Dabei kann dieser Zeitpunkt zum einen im Ausführungszeitraum des Entpackerstubs liegen und zum anderen das Beenden des ursprünglichen Programms markieren. Welcher der beiden Fälle vorliegt, kann anhand der Ausgabe auf "stdout" beurteilt werden, da die Ausgabe des ursprünglichen Programmes bekannt ist. Die Ausgabe ist im Analyseergebnis, wie im Kapitel 4.3.4 beschrieben, enthalten. Im zweiten Fall kann das Ergebnis dieser Heuristik als Referenz für die Bewertung der anderen Heuristiken genutzt werden.

Die zusammengefassten Ergebnisse finden sich in der Abbildung 10. Darin enthalten sind die Packer-Heuristik-Kombinationen und dem zugeordnet die Anzahl der erfolgreich entpackten Samples. Da für jeden Packer fünf Samples erstellt wurden, ist die maximal zu erreichende Anzahl an erfolgreich entpackten Samples pro Packer-Heuristik-Kombination ebenfalls fünf. Dabei gibt es Einschränkungen, da einige Programme nach dem Packen nicht mehr lauffähig waren. Die betroffenen Kombinationen sind zum einen "nslookup", wenn es mit dem mew11-Packer gepackt wird, und zum anderen das Programm "ipconfig", wenn es mit dem ASPack- oder dem PECompact-Packer genutzt wird. Darüber hinaus war "md5sum" nicht mehr lauffähig, nachdem es mit PECompact gepackt wurde. Daher können für den mew11- und den ASPack-Packer maximal vier Samples durch die Heuristiken erfolgreich entpackt werden. Beim PECompact-Packer liegt die maximale Anzahl an erfolgreich entpackten Samples bei drei.

Die zusammengefasst Ergebnistabelle, siehe Abbildung 10, zeigt welche Heuristiken in Kontext von frei verfügbaren Packern gute Ergebnisse liefern. Im Folgenden werden die Ergebnisse der Heuristiken beschrieben und Rückschlüsse darauf gezogen, welche Aspekte Einfluss auf den Erfolg der jeweiligen Heuristik nehmen.

	ASPack *	FSG	mew11 *	MPRESS	PECompact **	Upack	UPX	
Hump-and-Dump	0	5	2	0	2	5	5	Σ 19
NtCreateProcess	0	0	0	0	0	0	0	Σ 0
LoadLibrary	0	0	4	2	0	2	2	Σ 10
GetProcAddress	0	5	4	5	0	5	5	Σ 24
W^X	4	5	2	5	0	5	5	Σ 26
Stackpointer	4	5	0	4	0	0	5	Σ 18
Stacktiefe	0	5	4	4	3	5	5	Σ 26
Kommandozeilen-Argumente	4	5	4	5	3	5	5	Σ 31
NtTerminateProcess	4	5	4	5	3	5	5	Σ 31

Abbildung 10: Die Aufstellung der, durch die Heuristiken (waagerecht), erfolgreich entpackten Samples die mit den verschiedenen Packern (senkrecht) gepackt wurden. Die maximale Anzahl pro Heuristik-Packer-Kombination beträgt 5.
$*$ = maximal 4 Samples $**$ = maximal 3 Samples

Hump-and-Dump

Die Hump-and-Dump-Heuristik hat 19 von 31 Samples erfolgreich entpacken können. Sie beinhaltet die Schwierigkeit, die richtigen Parameter für die Erkennung der Entpackschleife, wie im Kapitel 4.4.1 erklärt, zu finden.

Im Kontext dieser Heuristik gibt es mehrere Aspekte, die den Erfolg negativ beeinflussen können. Ein Aspekt bezieht sich unter anderem auf die Implementation der Schleifen innerhalb des Entpackerstubs. Dabei besitzt nicht jeder Entpackerstub eine Schleife, die durch eine Mehrfachausführung in dieser Art zu erkennen ist. Dies kann zum Beispiel durch die folgenden Instruktionen erreicht werden.

```
1 mov   ecx, 200
2 rep   movsw
```

Dabei wird prozessorintern 200 mal die "MOVSW"-Instruktion ausgeführt, die jeweils zwei Byte von einer Speicherstelle an eine andere kopiert. Dabei passiert die Mehrfachausführung der Kopierinstruktion implizit und wird nicht durch das mehrfache Springen an diese Stelle erreicht. Durch einen besonderen Betriebsmodus des Prozessors, der eine Einzelschrittausführung der Instruktionen zulässt ist es möglich, diese Mehrfachausführung zu erkennen, jedoch muss dies von der verwendeten Instrumentationstechnik unterstützt werden.

Eine weitere Möglichkeit, diese Heuristik zu umgehen, besteht darin schon vor dem Entpacken eine große Schleife zu durchlaufen, so dass der Schwellenwert überschritten

wird. Dies würde ein vorzeitiges Anschlagen der Heuristik bewirken.

Im Fall von dem ASPack-Packer hängt der Misserfolg dieser Heuristik mit den im Kapitel 4.1 gemachten Annahmen und der daraus abgeleiteten Implementation des Frameworks zusammen. ASPack reserviert sich anfänglich mit der API-Funktion "VirtualAlloc" neuen Speicher und entpackt in der ersten Phase das gepackte Programm in diesen Bereich. Die hierzu genutzte Schleife wird von der Heuristik erkannt. Da der mittels "VirtualAlloc" reservierte Speicher von dem Framework nicht überwacht wird, kann das entpackte Programm darin nicht gefunden werden. Erst in der zweiten Phase wird das Programm an die endgültige Speicherstelle kopiert, an der später das Programm ausgeführt wird.

NtCreateProcess

Die NtCreateProcess-Heuristik, die im Kapitel 4.4.3 vorgestellt wurde, hat kein Sample entpacken können, was darauf zurück zu führen ist, dass weder die Entpackerstubs noch die Programme diese API-Funktion nutzen. Die Heuristik basiert auf der Annahme, dass die Malware diesen Aufruf nutzt. Da in dieser Evaluation keine Malware sondern gutartige Programme genutzt wurden, ist diese Annahme nicht erfüllt. Daher kann aus dem Misserfolg der Heuristik in dieser Auswertung nicht auf einen Misserfolg im Kontext von Malware geschlossen werden. Um diesen Kontext abzudecken sind weitere Testreihen notwendig, die von dieser Arbeit nicht abgedeckt werden.

Die Auswertung zeigt dennoch, dass durch die gemachte Annahme die Heuristik nicht generisch genug ist, um gepackte Programme und Malware entpacken zu können.

LoadLibrary

Die LoadLibrary-Heuristik, die im Kapitel 4.4.4 vorgestellt wurde, konnte nur 10 der 31 Samples erfolgreich entpacken. Dabei konnten die Samples von drei Packern nicht entpackt werden. Bei drei weiteren Packern ist es auffällig, dass jeweils nur zwei Programme entpackt wurden. Dabei ist zu bemerken, dass dies jeweils das Programm "md5sum" und "sha1sum" ist. Aufgrund dieses Sachverhaltes ist der Erfolg dieser Heuristik, zumindest bei diesen drei Packern, von den gepackten Programmen abhängig. Damit ist die Heuristik nicht generell für alle Packer erfolgreich anwendbar.

Ein weiterer Aspekt, der Einfluss auf den Erfolg hat, ist die Möglichkeit, dass der Entpackerstub die Funktion schon vor dem Entpacken nutzen kann. Dies ist unter anderem bei ASPack der Fall, weshalb die Heuristik Programme nicht entpacken kann, die mit diesem gepackt sind.

GetProcAddress

Die Aufrufe der GetProcAddress-Funktion sind nach der Auswertung ein gutes Indiz dafür, dass der Entpackerstub das ursprüngliche Programm erfolgreich entpackt hat. Die genauer Beschreibung dieser Heuristik ist im Kapitel 4.4.4 zu finden. Von den 31 getesteten Samples wurden 24 erfolgreich entpackt. Dabei war die Heuristik nur bei zwei Packern, dem ASPack- und dem PECompact-Packer, nicht erfolgreich. Beim ASPack-Packer ist dies auf die frühzeitige Nutzung der API-Funktion zurückzuführen. Dies könnte in Einzelfällen durch eine Anhebung des Schwellenwertes kompensiert werden.

Diese Heuristik kann ebenfalls durch einen integrierten Nachbau der GetProcAddress-Funktion umgangen werden. Dadurch wäre der Entpackerstub nicht mehr auf den Aufruf der API-Funktion angewiesen und könnte dennoch die Funktionalität nutzen. Bei den hier verwendeten Packern wurde diese Technik nicht beobachtet.

W^X

Die W^X-Heuristik, wie im Kapitel 4.4.5 beschrieben, ist eine Variante, die auf dem Speicherzugriffsverhalten basiert. Diese Heuristik schlug bei einem Packer fehl, dennoch ist sie im Vergleich zu den anderen Heuristiken effektiv und hat 26 Samples erfolgreich entpackt. Dies bekräftigt die starke Verbreitung in den Unpacking-Frameworks, wie im Kapitel 3.1 angemerkt.

Zwei Möglichkeiten, diese Heuristiken fehlschlagen zu lassen, wurden bereits im Kapitel 4.4.5 genannt. Die Erste nutzt dabei die API-Funktionen "CreateFileMapping" und "MapViewOfFile", um den selben physikalischen Speicherbereich an zwei unterschiedliche virtuelle Speicherbereiche zu legen. Somit kann ein Bereich zum Schreiben und der Andere zum Ausführen genutzt werden [24]. Die Heuristik müsste, um dies abzufangen, diese beiden Funktionsaufrufe beachten und die beiden Speicherbereiche als einen betrachten. Eine andere Methode ist es, auf dem physikalischen Speicher zu operieren und Schreibzugriffe sowie Ausführungen aufzuzeichnen. Dies ist jedoch nicht mit jeder Instrumentationstechnik möglich.

Die zweite Möglichkeit, die ein Packer nutzen kann, besteht darin schon vor dem Entpacken einen Speicherstelle auszuführen, die zuvor beschrieben wurde. Dies ist unter anderem bei selbst modifizierenden Programmen der Fall. Dieser Aspekt kommt ebenfalls dann zum tragen, wenn mehrere Packer verwendet wurden. Unpacking-Frameworks, wie zum Beispiel Renovo [10], gehen daher iterativ vor und setzten den Zustand immer wieder zurück. So ist es ihnen möglich mehrere Packschichten verarbeiten zu können. Dazu ist eine weitere Eigenschaft notwendig, die diese Wiederholung abbricht. Dies kann zum Beispiel die Laufzeit, wie bei Renovo, oder das Erkennen einer Malware im Speicher, wie bei Justin [6], sein. Dieses iterative Verfahren wird in dieser Arbeit nicht genutzt, da dies der zeitlichen Anforderung an diese Arbeit nicht gerecht wird. Dennoch zeigt sich in der Ergebnismatrix, die in Abbildung 10 zu sehen ist, dass bei einer einfachen Anwendung von Packern auf Programmen diese Heuristik erfolgreich ist.

Stackpointer

Der Stackpointer ist als Grundlage für eine Heuristik zum Entpacken von Programmen verwendbar. Diese wurde im Kapitel 4.4.6 vorgestellt. So konnten in dieser Testreihe 18 der insgesamt 31 Samples damit entpackt werden. Da die Heuristik für die jeweiligen Packer entweder alle oder kein Sample entpacken konnte, ist der Erfolg dieser Heuristik nicht von dem gepackten Programm sondern vom verwendeten Packer abhängig.

Der Stackpointer zeigt auf das untere Ende des Stackspeichers, der für jeden Prozess bereit gestellt wird. Dieser verändert sich während der Nutzung, zum Beispiel wenn Funktionen aufgerufen werden. Daher ist es untypisch, dass eine Funktion beziehungsweise ein Programm einen festen Stackpointerwert erwartet. Dies wird jedoch von der Heuristik angenommen. Dennoch liefert die Heuristik gute Ergebnisse. Ein weiteres Argument, warum

ein bestimmter Stackpointerwert zu Beginn eines Programmes nicht vorausgesetzt werden sollte, liefert die Technik "Address Space Layout Randomization" (ASLR) [28], die in vielen modernen Betriebssystemen eingesetzt wird. Diese Technik positioniert den Stack beim Starten eines Programmes zufällig, was bedeutet, das bei jedem Start des Programmes der Stackpointer auf eine andere Speicherstelle zeigt. Dadurch wird ebenfalls die Wahrscheinlichkeit gesenkt, dass diese Annahme für Prozesse erfüllt sein muss.

Ein Grund, wieso diese Heuristik dennoch gut funktioniert, liegt in der Verwendung der Instruktionen "PUSHA" und "POPA" im Entpackerstub. Die Instruktion "PUSHA" sichert dabei eine Reihe von Registern, darunter auch den Stackpointer, der im "esp"-Register gespeichert ist. Die Instruktion "POPA" kehrt dies um, setzt also insbesondere auch den Stackpointer wieder auf den vorher gesicherten Wert. Diese Instruktionen werden unter anderem von dem Packer ASPack genutzt, weshalb die Heuristik die damit gepackten Samples entpacken konnte.

Der Erfolg dieser Heuristik kann dennoch explizit durch den Entpackerstub beeinflusst werden. Zum einen kann der Stackpointer vor der Entpackfunktion wieder rekonstruieren werden und zum anderen ist es möglich, dass der Entpackerstub den Stack nicht benutzt und deshalb der Stackpointer nicht verändert wird. In beiden Fällen würde die Heuristik vorzeitig anschlagen.

Stacktiefe

Die Heuristik, die die Stacktiefe als Indiz nimmt und im Kapitel 4.4.8 besprochen wurde, gehört ebenfalls zu den Besseren mit 26 erfolgreich entpackten Samples. Die Stacktiefe wird dabei anhand der gemachten "CALL"- und "RET"-Instruktionen errechnet. Die Heuristik konnte auch die PECompact-Samples entpacken, bei denen die meisten Heuristiken nicht erfolgreich waren. Die Heuristik basiert auf der Annahme, dass vor der Entpackfunktionalität nur wenige Funktionen verschachtelt aufgerufen werden. Der Erfolg der Heuristik in dieser Testreihe untermauert diese Annahme. Jedoch konnte die Heuristik die Samples des ASPack-Packer nicht entpacken. ASPack stellt im Vergleich zu den anderen hier genutzten Packer eine Ausnahme dar, da dieser schon mehr als 10 Funktionen verschachtelt aufruft.

In diesem Fall ist der ausbleibende Erfolg darauf zurückzuführen, dass vor dem Entpacken anderen Vorbereitungen von dem Entpackerstub getroffen werden und deshalb viele Funktionsaufrufe notwendig sind. Eine weitere Möglichkeit besteht darin, gezielt eine Vielzahl an Funktionsaufrufen zu simulieren, ohne dass dies viel Programmcode und damit Platz benötigt, da der Entpackerstub möglichst klein gehalten werden soll:

```
1 mov     ecx, esp
2 add     ecx, -0x40
3 LOOP:
4  call   FUNC
5 FUNC:
6 cmp     ecx, esp
7 jne     LOOP
8 add     esp, 0x40
```

Diese Instruktionen bilden eine Schleifen ab, die 16 "CALL"-Instruktionen ausführt und im Anschluss den Stack wieder aufräumt. Solch ein Konstrukt kann im Entpackerstub frühzeitig aufgerufen werden, damit die Heuristik hier anschlägt.

Kommandozeilen-Argumente

Die Heuristik, die das Vorhandensein der Kommandozeilen-Argumente nutzt, um den fertigen Entpackvorgang zu erkennen, spielt in dieser Evaluation eine Sonderrolle. Diese Heuristik wurde im Kapitel 4.4.7 vorgestellt.

Die Heuristik basiert auf der Annahme, dass der Entpackerstub die Kommandozeilen-Argumente auf dem Stack platziert, bevor er den Kontrollfluss an das entpackte Programm übergibt. Diese Annahme ist dahingehend problematisch, als dass die Programme in der Regel diese Kommandozeilen-Argumente selbst auf den Stack legen, weshalb es nicht notwendig ist, dass dies der Entpackerstub auch macht. Die Funktion, die dafür verwendet werden kann ist die "__getmainargs"-Funktion [15].

Dies bedeutet, dass die Heuristik eine Funktionalität des ursprünglichen Programmes erkennt und nicht die Vorbereitung zum Start des Programmes. Dies ist dahingehend problematisch, als dass der Erfolg diese Heuristik von den gepackten Programmen abhängt. Die Programme insbesondere Malware ist nicht darauf angewiesen die Kommandozeilen-Argumente auf den Stack zu legen und zu verarbeiten, wenn diese nicht benötigt werden. In solchen Fällen würde die Heuristik fehlschlagen. In diesem Testverfahren wurden einfache Konsolenanwendungen als zu packende Programme verwendet. Diese sind auf Kommandozeilen-Argumente angewiesen und legen diese auch selbst auf den Stack, damit der Endanwender das Programm sinnvoll nutzen kann. Daher konnte die Heuristik alle Samples erfolgreich entpacken. Diese Tatsache lässt jedoch keine Rückschlüsse auf den Erfolg bei der Anwendung auf Malware zu. Diese muss in einer gesonderten Testreihe beobachtet und ausgewertet werden.

NtTerminateProcess

Die NtTerminateProcess-Heuristik aus dem Kapitel 4.4.2 nimmt dahin gehend eine Sonderrolle ein, als dass sie als Referenz für die Ergebnisse der anderen Heuristiken geplant und genutzt wird. Um dies zu erreichen wurden Programme gewählt, die sich selbst ohne Interaktion beenden. Das bedeutet, dass sich die Programme in jedem Fall beenden und daher die API-Funktion NtTerminateProcess aufgerufen wird. Dabei ist die Möglichkeit zu beachten, dass auch der Entpackerstub schon vorzeitig die Ausführung beenden kann. Dies lässt sich anhand der Ausgaben, die das Programm ausgibt, erkennen. In der Testreihe wurde kein Programm frühzeitig durch den Entpackerstub beendet und es konnten die Ergebnisse der NtTerminateProcess-Heuristik als Referenz für die anderen Heuristiken genutzt werden.

Auf Grund der Sonderrolle, die diese Heuristik hier einnimmt, gibt diese Auswertung keinen Aufschluss darüber, wie effektiv diese Heuristik beim Entpacken von Malware ist.

5.1 Zusammenfassung

In der Evaluation wurden die neun Heuristiken auf die 31 Samples des Testdatensatzes angewandt. Dabei wurden die Analyseergebnisse, die mithilfe der Heuristiken erzeugt wurden, auf der Basis der Metriken ausgewertet.

Die Evaluation zeigt, dass der Großteil der Heuristiken ein gutes bis sehr gutes Ergebnis erbracht hat. Dabei konnten sieben der neun Heuristiken über die Hälfte der Samples erfolgreich entpacken. Nur zwei Heuristiken haben schlechte Ergebnisse geliefert, darunter

auch die NtCreateProcess-Heuristik, die kein Sample entpacken konnte. Die Heuristik, die die Kommandozeilen-Argumente als Indiz nutzt, und die NtTerminateProcess-Heuristik zeigen, dass sich die Ergebnisse dieser Evaluation nicht ohne weitere Auswertungen auf den Einsatz im Kontext von Malware übertragen lassen.

Bei einem Vergleich der Packer lässt sich erkennen, dass die Samples von einigen Packern mehrheitlich entpackt werden konnten. Ein Beispiel dafür ist UPX [19], der zur einfachen Komprimierung von Programmen entwickelt wurde. Andere Packer wie ASPack [11] und PECompact [29] werden kommerziell entwickelt und vertrieben. Dies zeigt sich auch in den Ergebnissen, da Samples die mit diesen gepackt wurden seltener erfolgreich entpackt wurden.

Es wurde gezeigt, dass die Heuristiken keine allgemeingültige Lösung bieten, um gepackte Programme wieder zu entpacken und insbesondere der Einsatz einzelner Heuristiken nicht reicht, um bei allen Packern erfolgreich zu sein. Denn zu jeder Heuristik wurde mindestens eine Methoden gefunden, die von Packern genutzt werden kann, um die Heuristik fehlschlagen zu lassen.

6 Fazit

Malware und Programme sind oft mithilfe von Packern gepackt und so nur schwer zu analysieren. Daher wurden Unpacking-Frameworks entwickelt, die Heuristiken nutzen um diesen Prozess umzukehren. Da der Erfolg maßgeblich von den Heuristiken abhängt, ist eine Bewertung dieser notwendig, um diese einschätzen und weiterentwickeln zu können.

In dieser Arbeit wurde ein Framework zur Anwendung und Bewertung der Heuristiken entwickelt. Eine im Kapitel 4.4 getroffene Auswahl an Heuristiken wurden Implementiert und mit einer Auswahl der Metriken bewertet. Die im Kapitel 5 vorgestellte Evaluation zeigt, dass mit dem Einsatz einzelner Heuristiken dem Problem des Entpackens nicht in allen Fällen effektiv begegnet werden kann. Die Evaluation zeigt weiterhin auf, dass sich die gewonnenen Erkenntnisse nicht ohne weitere Auswertungen auf die Effektivität der Heuristiken im Kontext von Malware übertragen lassen.

Die Heuristiken zeigen im Durchschnitt ein gutes Ergebnis. Es ist auch zu erkennen, dass Packer gezielt Methoden verwenden oder verwenden könnten, die den Erfolg der Heuristiken negativ beeinflusst. Eine Weiterentwicklung der bestehenden und eine Neuentwicklung weitere Heuristiken ist daher meiner Meinung nach notwendig, um mit der Entwicklung der Packer mitzuhalten. Die Tatsache, dass einzelne Heuristiken von Packern gezielt getäuscht werden, kann mit einer Kombination aus Heuristik begegnet werden. Dies würde den Aufwand der Gegenmaßnahme wesentlich steigern, was dazu führt, dass die Anwendung dieser unpraktikabel wird.

7 Ausblick

Neben der Verbesserung und Ausweitung der Bewertung von Heuristiken aus Unpacking-Frameworks liefert diese Arbeit Hinweise dafür, dass ein modifiziertes Konzept von Unpacking-Frameworks die Heuristiken effektiver nutzen kann.

Die Implementation des hier konzipierten Frameworks zur Anwendung und Bewertung von Heuristiken bietet noch Möglichkeiten des Ausbaus und der Verbesserung. So wird

die Beobachtung und Verarbeitung von dynamisch allozierten Speicher nicht hinreichend beachtet. Diese Erweiterung würde die Qualität der Bewertungen verbessern und die Möglichkeit bieten weitere Aspekte bei der Anwendung von Heuristiken zu nutzen.

Um eine Auswertung im Kontext von Malware effizient durchführen zu können, ist eine Erweiterung des Frameworks nötig, die eine Möglichkeit schafft das gesamte System nach einem Testlauf auf einen gesicherten Zustand zurückzusetzen. Damit wäre eine automatisierte Bewertung der Heuristiken im Kontext von Malware möglich, die in dieser Arbeit nur unzureichend betrachtet werden konnte.

Dennoch bietet das Framework jetzt schon die Möglichkeit Heuristiken zu verbessern. In der Weiterentwicklungsphase der Heuristiken kann das Framework genutzt werden, um nach einer Änderung an einer Heuristik, die Auswirkung auf die Effektivität zu messen. So kann frühzeitig erkannt werden, ob durch die gemachte Änderung die Heuristik stabiler und besser wird.

Die Erkenntnisse dieser Arbeit liefern Hinweise, dass eine Kombination aus verschiedenen Heuristiken ein weiterer Schritt in der Entwicklung von Unpacking-Frameworks sein kann. So ist ein neues Framework möglich, dass die Aussagen von mehreren Heuristiken verarbeitet und diese als Auslöser nutzt, um zum Beispiel ein Speicherabbild zu erstellen. Die Ausführung des Samples wird dabei nicht unterbrochen, sondern weiter fortgesetzt. Im Laufe der Ausführung würden so mehrere Speicherabbilder entstehen. Die Tatsache, dass mehrere Abbilder zu unterschiedlichen Zeitpunkten erstellt werden, erhöht die Chance das ursprüngliche Programm in mindestens einem Abbild gesichert zu haben. Eine Herausarbeitung der Unterschiede zwischen den einzelnen Abbildern und die Anwendung der in dieser Arbeit genutzten Metriken liefert dabei eine gute Informationsgrundlage, über das Vorhandensein des entpackten Programmes. Dieses Konzept hat den Vorteil, dass verschiedene Heuristiken genutzt werden können und somit Versuche einzelne Heuristiken fehlschlagen zu lassen scheitern. Es bietet darüber hinaus die Möglichkeit weitere Metriken zu integrieren und zu evaluieren. Dies gilt auch für weitere Heuristiken. So ist gewährleistet, dass das Framework mit der Entwicklung der Packer mithalten kann und weiterhin nutzbar bleibt.

Literatur

[1] Daniel Bilar. "Opcodes as predictor for malware". In: *Int. J. Electron. Secur. Digit. Forensic* 1 (2 2007), S. 156–168.

[2] *Bochs IA-32 Emulator Project*. Juli 2011. URL: http://bochs.sourceforge.net.

[3] Lutz Böhne. *Pandora's Bochs: Automatic unpacking of Malware*. RWTH Aachen University, 2008.

[4] Artem Dinaburg u. a. "Ether: malware analysis via hardware virtualization extensions". In: *Proceedings of the 15th ACM conference on Computer and communications security*. CCS '08. Alexandria, Virginia, USA: ACM, 2008, S. 51–62.

[5] *FSG v2.0*. Juli 2011. URL: http://www.xtreeme.prv.pl/.

[6] Fanglu Guo, Peter Ferrie und Tzi-Cker Chiueh. "A Study of the Packer Problem and Its Solutions". In: *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*. RAID '08. Cambridge, MA, USA: Springer-Verlag, 2008, S. 98–115.

[7] Hex-Rays. *IDA Pro Disassembler and Debugger*. Juli 2011. URL: http://www.hex-rays.com/idapro/.

[8] Guhyeon Jeong u. a. "Generic unpacking using entropy analysis". In: *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*. 2010, S. 98 –105.

[9] Jibz u. a. *The PEiD project*. 2008. URL: http://www.peid.info.

[10] Min Gyung Kang, Pongsin Poosankam und Heng Yin. "Renovo: a hidden code extractor for packed executables". In: *Proceedings of the 2007 ACM workshop on Recurring malcode*. WORM '07. Alexandria, Virginia, USA: ACM, 2007, S. 46–53.

[11] StarForce Technologies Ltd. *ASPack - an advanced Win32 executable file compressor*. Juli 2011. URL: http://www.aspack.com/.

[12] Chi-Keung Luk u. a. "Pin: building customized program analysis tools with dynamic instrumentation". In: *SIGPLAN Not.* 40 (6 2005), S. 190–200.

[13] Unpack executing Malware u. a. *PolyUnpack: Automating the Hidden-Code Extraction of*.

[14] MATHEsoft. *Upack-Optimizer*. Juli 2011. URL: http://upo.mathesoft.de/.

[15] Microsoft. *MSDN: __getmainargs*. Juli 2011. URL: http://msdn.microsoft.com/en-us/library/ff770599.aspx.

[16] Microsoft. *MSDN Library*. Juli 2011. URL: http://msdn.microsoft.com/en-us/library/.

[17] *Microsoft Windows*. Juli 2011. URL: http://www.microsoft.com/windows/.

[18] Northfox. *MEW - real exe packer*. Juli 2011. URL: http://northfox.uw.hu/.

[19] Markus Franz Xaver Johannes Oberhumer, László Molnár und John F. Reiser. Juli 2011. URL: http://upx.sourceforge.net/.

[20] *Python*. Juli 2011. URL: http://www.python.org.

[21] *QEMU: opensource processor emulator*. Juli 2011. URL: http://www.qemu.org/.

[22] Monirul Sharif u. a. "Eureka: A Framework for Enabling Static Malware Analysis". In: *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*. ESORICS '08. Berlin, Heidelberg: Springer-Verlag, 2008, S. 481–500.

[23] Matthew Skala u. a. *The GNU Privacy Guard*. Juli 2011. URL: ftp://ftp.gnupg.org/gcrypt/binary/.

[24] skape. "Using dual-mappings to evade automated unpackers". In: *Uninformed Journal* 10.1 (Okt. 2008). URL: http://www.uninformed.org/?v=10\&a=1\&t=sumry.

[25] MATCODE Software. *MPRESS is a free, high-performance executable packer*. Juli 2011. URL: http://www.matcode.com/mpress.htm.

[26] Dawn Song u. a. "BitBlaze: A New Approach to Computer Security via Binary Analysis". In: *Proceedings of the 4th International Conference on Information Systems Security*. ICISS '08. Hyderabad, India: Springer-Verlag, 2008, S. 1–25. URL: http://bitblaze.cs.berkeley.edu/.

[27] Li Sun, Tim Ebringer und Serdar Boztas. *Hump-and-dump: efficient generic unpacking using an ordered address execution histogram*.

[28] PaX Team. *address space layout randomization*. März 2003. URL: http://pax.grsecurity.net/docs/aslr.txt.

[29] Bitsum Technologies. *PECompact*. Juli 2011. URL: http://www.bitsum.com/pecompact.php.

[30] *The Xen® hypervisor*. Juli 2011. URL: http://www.xen.org/.

[31] Tadas Vilkeliskis. "Automated unpacking of executables using Dynamic Binary Instrumentation". In: (2009). URL: http://tadas.vilkeliskis.com/research/.