technische universität
dortmund

Christian Blichmann

Automatisierte Signaturgenerierung für Malware-Stämme

Diplomarbeit

03.06.2008

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl Informatik VI

Ls6

Betreuer:
Prof. Dr. Joachim Biskup
Dr. Michael Meier

GERMANY · D-44221 DORTMUND

Erklärung

Hiermit erkläre ich, Christian Blichmann, dass ich die Diplomarbeit mit dem Titel

„Automatisierte Signaturgenerierung für Malware-Stämme“

selbstständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt habe.

Dortmund, 03.06.2008

Danksagung

An dieser Stelle möchte ich mich bei Prof. Dr. Joachim Biskup für die Ermöglichung dieser Diplomarbeit bedanken. Besonderer Dank gilt Dr. Michael Meier für die gute Betreuung und den fachlichen Diskussionen und Anregungen. Außerdem Danke ich Thomas Dullien von der Firma Zynamics GmbH für den Themenvorschlag und die gute Zusammenarbeit, sowie der Bereitstellung der für diese Arbeit notwendigen Softwarewerkzeuge.

Weiterer Dank gilt meinen Eltern für ihre – nicht nur finanzielle – Unterstützung und der Ermöglichung des Studiums.

Melanie Sellmann danke ich für das Korrekturlesen und ihre Geduld und Alexander Burris für fachliche Anmerkungen.

Zusammenfassung

Eine ständig wachsende Zahl von Schadprogrammen für Rechensysteme, auch als Malware bezeichnet, stellt Antivirenprogramme vor die Herausforderung, diese möglichst schnell und mit niedrigen Fehlerraten zu erkennen. Eine große Anzahl zu überprüfender Signaturen führt hierbei zu einem Trade-Off zwischen hoher Erkennungsrate auf der einen und möglichst guten Antwortzeiten des Rechensystems auf der anderen Seite.
Aktuelle Signaturen sind für die Sicherheit moderner Rechensysteme jedoch unverzichtbar. Im Rahmen dieser Arbeit wird eine Methode vorgestellt, mit der herkömmliche Scanner von – für Echtzeiteinsatz häufig nicht geeigneten – automatischen Klassifikationsmethoden für Malware profitieren können. Hierbei wird für eine gegebene Familie von Schadprogrammen eine Byte-basierte Signatur für den Scanner ClamAV generiert, mit der zuverlässig die gesamte Familie erkannt werden kann. Die so erzeugte Signatur wird anschließend in Bezug auf falsche Positive und falsche Negative getestet.

Abstract

An ever growing number of malicious software for computer systems – or malware for short – challenges anti-virus software to detect these as fast as possible and with minimal error rates. A large number of signatures to check leads to a trade-off between high detection rates on the one hand and best possible response times on the other.
Signatures that are up-to-date are essential for the security of modern computer systems, however. Within the scope of this work, a method is presented to allow conventional scanners to profit from automated classification methods for malware that are otherwise unsuitable for use with on-access scanners. At this, a byte-based signature for use with the ClamAV scanner is generated from a given family of malware, allowing to reliably detect the family as a whole. The generated signature is tested regarding false positives and false negatives.

Inhaltsverzeichnis

1 Einleitung 1

2 Begriffe, Konzepte und Verfahren 5

2.1 Allgemeines . . . 5
2.2 Malware . . . 6
2.2.1 Arten von Malware . . . 7
2.2.2 Tarntechniken . . . 8
2.3 Antivirenprogramme und -Techniken . . . 9
2.3.1 Scanner . . . 10
2.3.2 Signaturen . . . 11
2.4 Signaturbeschreibungssprachen . . . 12
2.4.1 Semantische Aspekte . . . 12
2.4.2 SHEDEL . . . 13
2.4.3 EDL . . . 16
2.4.4 Reguläre Ausdrücke . . . 17
2.4.5 Online-/Offline-Signaturen . . . 20
2.4.6 ClamAV-Signaturen . . . 21
2.5 Analyse von Malware . . . 26
2.5.1 IDA Pro . . . 26
2.5.2 BinDiff . . . 30
2.5.3 Graph-basiertes Vergleichen von Objektcode . . . 31
2.6 Automatische Klassifizierung von Malware . . . 36
2.6.1 VxClass . . . 36

3 Automatisierte Signaturgenerierung 41

3.1 Überblick . . . 42
3.1.1 Ziele . . . 42
3.1.2 Idee für ein Signaturgenerierungsverfahren . . . 43

3.2 Signaturgenerierung mittels BinDiff und k-LCS 44
3.2.1 Definitionen 44
3.2.2 Formale Beschreibung 46
3.3 Längste gemeinsame Teilsequenzen. 49
3.3.1 Dynamische Programmierung 50
3.3.2 Weitere Verfahren 51
3.4 Längste gemeinsame Teilsequenzen von Permutationen 52
3.4.1 Korrektheit 54
3.4.2 Laufzeiteigenschaften 54
3.5 Eine Heuristik für das *k*-LCS-Problem 56
3.6 Kürzungsstrategien 57
3.7 Konstruktion eines regulären Ausdrucks aus einer gemeinsamen Teilsequenz 57

4 Implementierung **61**
4.1 Überblick 61
4.2 Verwendete Programmiersprachen 63
4.2.1 Java 63
4.2.2 C++. 64
4.3 Schnittstelle zu IDA Pro und BinDiff 64
4.3.1 IDAJava 65
4.3.2 SWIG 68
4.4 Programmbeschreibung 69
4.4.1 Implementierung des Rahmenprogramms. 70
4.4.2 Datenstrukturen und Optimierungen 71
4.4.3 Einschränkungen von ClamAV-Signaturen. 73
4.4.4 Unterstützte Plattformen 73
4.4.5 Benutzerschnittstelle. 73
4.5 Probleme 75

5 Bewertung **77**
5.1 Erreichte Ziele. 77
5.2 Falsche Positive 78
5.2.1 Testverfahren. 79
5.3 Falsche Negative 81
5.4 Experimente. 82
5.4.1 Unbekannte Mitglieder einer Malware-Familie 82

5.4.2 Einfluss der Kürzungsstrategien 83
5.5 Performanz des Verfahrens. 83

6 Schlussbetrachtungen **85**

Abkürzungsverzeichnis **87**

Abbildungsverzeichnis **89**

Tabellenverzeichnis **91**

Listings **93**

Literaturverzeichnis **95**

1 Einleitung

Schadprogramme für Rechensysteme, im Allgemeinen als Malware bezeichnet, stellen für die Sicherheit der Infrastruktur einer Informationsgesellschaft eine ernstzunehmende Bedrohung dar. Malware ist hierbei ein Sammelbegriff für unerwünschte Programme auf einem Rechensystem. Hierzu zählen unter anderem als Würmer, Viren und Trojanische Pferde, sowie Spyware bezeichnete Programme und Programmfragmente. Laut einer Studie von Computer Economics [13] belaufen sich die geschätzten direkten Kosten für die Analyse und die Behebung von entstandenen Schäden an infizierten Systemen sowie für Produktionsausfälle und Einnahmeverluste als Folge von Malware-Attacken allein für das Jahr 2006 weltweit auf 13,4 Mrd. US-Dollar.
Eine Standard-Antwort auf diese Bedrohung sind Antivirenprogramme, die mittels Signaturen Arbeitsplatzcomputer in Echtzeit auf das Vorhandensein von Schadprogrammen überprüfen.

In der Praxis ist eine ständig wachsende Zahl von Schadprogrammen zu beobachten. Dabei wird häufig das syntaktische Erscheinungsbild eines bereits existierenden Schadprogramms (zum Beispiel mit Techniken wie Polymorphie oder Metamorphie, siehe Abschnitt 2.2.2) variiert, um eine Entdeckung zu erschweren. Damit einhergehend steigt die Zahl der von Antivirenprogrammen zu überprüfenden Signaturen. Dies führt zu einem Trade-Off zwischen hoher Erkennungsrate auf der einen und möglichst guten Antwortzeiten des Rechensystems auf der anderen Seite.
Ein weiteres Problem ist die zunehmende Zahl von Duplikaten in den Malware-Datenbanken der Hersteller von Antivirenprogrammen, die ebenfalls zu einer größer werdenden Anzahl von Signaturen beiträgt. Laut einem Bericht des Anbieters McAfee [22] wurden 2007 von ca. 2,8 Mio. gesammelten Malware-Exemplaren 1,7 Mio. als Duplikate erkannt, insgesamt wird der Anteil der Duplikate in der Malware-Datenbank auf etwa 36 % geschätzt.
Seitens der Industrie bedeutet eine zahlenmäßige Zunahme von Malware einen immensen personellen Aufwand bei der Analyse der jeweiligen Schadprogramme und der –

meist manuellen – Erstellung von Signaturen. Ebenfalls nicht zu vernachlässigen ist der finanzielle Aufwand, der für die Verteilung der Signaturen und der dafür benötigten Netzwerk-Ressourcen anfällt.

Da die Verfügbarkeit von tagesaktuellen Signaturen für die Sicherheit moderner Rechensysteme unverzichtbar ist, verspricht eine weitgehende Automatisierung des Signaturerstellungsprozesses eine drastische Reduzierung von Aufwand und Kosten. Daher wäre es von Vorteil, die Anzahl der Signaturen signifikant zu senken und gleichzeitig die Erkennungsrate durch Antivirenprogramme beizubehalten oder sogar zu steigern.
Die Varianten einer Malware sind – trotz unterschiedlichen syntaktischen Erscheinungsbildes – semantisch meist nahezu identisch. Hieraus lässt sich ein Ähnlichkeitsmaß ableiten, das die Zerlegung einer gegebenen Menge von Schadprogrammen in disjunkte Teilmengen erlaubt. In Anlehnung an die Biologie werden die Schadprogramme in diesen Teilmengen im Folgenden als *Stamm* oder *Familie* bezeichnet[1].
Eine solche automatische Klassifizierung von Malware in Familien kann beispielsweise mittels Clusteranalyse der paarweisen Ähnlichkeiten erfolgen[2], und Methoden wie in [16], [31] oder [29] eignen sich für die Definition eines Ähnlichkeitsmaßes, sind jedoch im Allgemeinen nicht für den Echtzeiteinsatz auf Arbeitsplatz-Computern geeignet. Hier sind die Byte-basierten Signatursysteme aktueller Antivirenprogramme mittelfristig nicht zu ersetzen.
Um dennoch von den Resultaten der automatischen Klassifikation zu profitieren, benötigt man eine Methode, welche aus einer gegebenen Familie von Schadprogrammen eine Byte-basierte Signatur generiert, mit der zuverlässig die gesamte Familie erkannt werden kann. Neue und noch unbekannte Varianten einer Malware können so von den verfügbaren Antivirenprogrammen mit der Signatur für einen bereits bekannten Stamm erkannt werden (siehe Abschnitt 5.4.1).

Im Rahmen der vorliegenden Arbeit sollen für Stämme von Malware, die durch eine geeignete Methode (darunter fällt auch die manuelle Analyse) vorab klassifiziert wurden, eine – möglichst kompakte – Signatur für alle diese Varianten der jeweiligen Malware zur Verwendung durch das Antivirenprogramm ClamAV generiert werden ([11], [35]).

Kapitel 2 führt zunächst die dieser Arbeit zugrunde liegenden Konzepte ein und liefert dafür notwendige Begriffsdefinitionen. Die semantischen Aspekte von Signaturen werden diskutiert und einige Signaturbeschreibungssprachen kurz vorgestellt, bevor

[1] Die klassische Systematik der Biologie nach Linné ordnet die Begriffe streng hierarchisch, sie werden hier synonym verwendet.

[2] Eine Implementierung des soeben beschriebenen Verfahrens findet sich in [65] und wird in Abschnitt 2.6.1 näher erläutert.

der Schwerpunkt auf eine genauere Betrachtung von ClamAV-Signaturen gelegt wird. Es folgt eine Beschreibung der graphentheoretischen Analyse von Objektcode und ein Überblick über die Architektur der dafür verwendeten Softwarewerkzeuge.

Das Verfahren der automatisierten Signaturgenerierung für Malware-Stämme wird in Kapitel 3 beschrieben. Zunächst wird das Verfahren grob skizziert und danach formal beschrieben. Anschließend folgen einige komplexitätstheoretische Betrachtungen zu den Laufzeiteigentschaften und dem Platzbedarf des vorgestellten Verfahrens.

Eine prototypische Implementierung des in Kapitel 3 vorgestellten Signaturgenerierungsverfahrens wird in Kapitel 4 beschrieben. Die verwendeten Softwarewerkzeuge und Programmiersprachen und die bei der Implementierung aufgetretenen Probleme werden erläutert.

Kapitel 5 beschäftigt sich mit dem Testen der Implementierung unter realen Bedingungen. Die verwendeten Testverfahren werden vorgestellt und die vom Prototyp generierten Signaturen auf falsche Positive und falsche Negative getestet. Es folgen einige Betrachtungen zu der Performanz des implementierten Prototyps im praktischen Einsatz.

Die Arbeit schließt in Kapitel 6 mit einem Ausblick und zeigt mögliche Verbesserungen auf.

2

Begriffe, Konzepte und Verfahren

Da die Terminologie im Bereich IT-Sicherheit nicht so etabliert ist, wie in anderen Teilbereichen der Informatik, führt dieses Kapitel zunächst grundlegende Begriffe im Zusammenhang mit Malware und Signaturgenerierung ein.
Es folgt ein Überblick über Signaturbeschreibungssprachen und deren semantische Aspekte, sowie eine Unterscheidung zwischen Signaturen für Systeme zur Einbruchs- und Missbrauchserkennung und Signaturen für Antivirenprogramme. In Abschnitt 2.4.4 werden reguläre Ausdrücke beschrieben, da sie die Grundlage für ClamAV-Signaturen bilden, die von dem in Kapitel 3 vorgestellten Verfahren generiert werden.
In Abschnitt 2.5 werden Softwarewerkzeuge zur Analyse von Schadprogrammen vorgestellt, und es wird ein Algorithmus für das strukturelle Vergleichen von Objektcode skizziert.
Im letzten Abschnitt dieses Kapitels wird ein Softwarewerkzeug zur automatischen Klassifikation von Schadprogrammen in Familien vorgestellt und dessen Architektur erläutert. Schadprogramme, die von diesem Werkzeug in Familien klassifiziert wurden, dienen dem Signaturgenerierungsverfahren aus Kapitel 3 als Eingabe.

2.1 Allgemeines

Die Programme eines Rechensystems liegen in ausführbarer Form vor oder können durch *Kompilieren* des Quelltextes in diese überführt werden. Ausführbare Programme werden in den Dateien des Rechensystems gespeichert.[1] Mit Dateien eines Rechensystems sind im Allgemeinen Dateien mit wahlfreiem Zugriff und Byte-Adressierung gemeint. Der direkt lauffähige Teil eines Programms[2] wird *Objektcode* genannt, die ausführbare Datei selbst

[1] Eine Ausnahme stellen *Single-Level-Store*-Architekturen [54] dar, in denen es nur eine implizite Speicherhierarchie gibt.
[2] nach eventueller Adressrelokation durch das Ladeprogramm des Betriebssystems

wird auch als *Binary* bezeichnet. Objektcode und Binary werden unter dem Oberbegriff *ausführbarer Code* zusammengefasst.

Das Untersuchen von Programmen ohne Kenntnis des Quelltexts mit dem Ziel, Einzelheiten über die in dem Programm implementierten Funktionen in Erfahrung zu bringen oder eine Rekonstruktion des Programmquelltexts zu erstellen, wird als *Reverse-Engineering* bezeichnet (siehe Abschnitt 2.5).

Eine *Byte-Sequenz* bezeichnet eine geordnete Menge von Bytes, also eine Sequenz (für eine formale Definition siehe Abschnitt 3.2.5) über dem durch ein Byte darstellbaren Alphabet[3].

2.2 Malware

Malware ist ein Sammelbegriff für alle Arten von Programmen, deren Intention bösartig ist, oder deren Seiteneffekte bei der Ausführung bösartig sind. Dies spiegelt sich schon in der Namensgebung wieder: Malware ist ein Kofferwort aus *malicious* (engl. für *bösartig*) und *Software*. Unter bösartig ist hierbei zu verstehen, dass Malware in der Regel vom Benutzer unerwünscht ist und gegebenenfalls schädliche Funktionen ausführt. Die *Schadfunktionen* werden dabei häufig getarnt, oder die Malware selbst wird unbemerkt im Hintergrund ausgeführt. Das Spektrum von Malware umfasst eine Vielzahl spezifischer Bedrohungen, darunter Würmer, Viren, Trojanische Pferde und Spyware.
In der Literatur findet sich kein allgemeiner Konsens über eine formale Definition des Begriffs. Die oben gegebene breite Definition ist [2] entnommen und deckt sich mit der in [32]. Der Versuch einer einfachen Malware-Taxonomie, die Tarnung einschließt, findet sich in [52].
In der vorliegenden Arbeit werden die Begriffe Malware und *Schadprogramm* synonym verwendet.

Die von einer Malware ausgeführten Schadfunktionen können eine oder mehrere der folgenden Aktionen umfassen:

- Manipulieren oder Löschen von Daten
- Ausspähen und Weiterleiten von Daten (in diesem Fall spricht man auch von *Spyware*)

[3] Im Folgenden wird die Darstellung nach ISO/IEC 8859-15 verwendet.

- Kompromittierung oder Unbrauchbarmachung installierter Sicherheitsprogramme (wie zum Beispiel Firewalls oder Antivirenprogramme)
- Weiterverbreitung des Schadprogramms auf andere Rechensysteme
- Installation einer Hintertür (Definition im nächsten Abschnitt)
- Registrierung bei einem entfernten Rechensystem und Entgegennahme von beliebigen Befehlen (Malware, die diese Schadfunktion ausführt, wird auch als *Bot* oder *Zombie* bezeichnet, das entfernte Rechensystem wird *Command- and Control System* genannt)

Üblicherweise schlägt außerdem das Entfernen von Schadprogrammen mit den gängigen Methoden des Rechensystems fehl, so dass verbliebene Programmfragmente weiterhin Schaden anrichten können.

2.2.1 Arten von Malware

Wie bereits oben erläutert, umfasst Malware eine ganze Reihe von Bedrohungen. Exemplarisch seien hier einige davon aufgeführt [2]:

Virus Ein *Virus* ist ein Schadprogramm, das sich während seiner Ausführung kopiert und in andere ausführbare Dateien einbettet (siehe auch [12]). Nach einer erfolgten Einbettung ist die Datei mit dem Virus *infiziert*. Diese charakteristische Eigenschaft der *Selbstreproduktion* wird erstmals in [36] beschrieben und dient der Weiterverbreitung des Schadprogramms.

Wurm Ähnlich zu Viren sind *Würmer* sich selbst reproduzierende Schadprogramme. Jedoch sind Würmer zur Weiterverbreitung nicht auf andere ausführbare Dateien angewiesen. Die Verbreitung erfolgt bei der Ausführung im Allgemeinen über ein an das Rechensystem angeschlossenes Netzwerk.

Trojanisches Pferd Kombination eines zunächst scheinbar nützlichen (oder zumindest unbedenklichen) Wirtsprogramms mit versteckt arbeitender Schadfunktion, wie zum Beispiel die Installation einer Hintertür. *Trojanische Pferde* verbreiten sich nicht selbst, sondern sind zur Verbreitung auf Benutzerinteraktion angewiesen. Mit der nützlichen Funktionalität des Wirtsprogramms wird daher für die Installation durch den Benutzer geworben.

Backdoor Als *Hintertür*[4] wird ein Schadprogramm oder Teil eines Schadprogramms bezeichnet, welches eine Umgehung der normalen Zugriffskontrolle des Rechensystems ermöglicht. Häufig wird so auch die Steuerung des Rechensystems über ein Netzwerk ohne Kenntnis des Benutzers möglich.

2.2.2 Tarntechniken

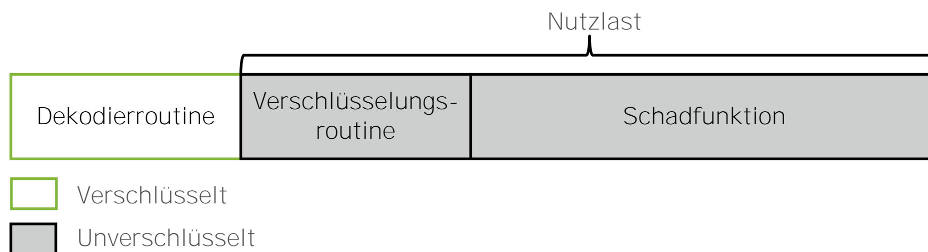

Abbildung 2.1: Ein einfaches Modell getarnter Malware

Um einer Entdeckung durch Antivirenprogramme (siehe Abschnitt 2.3) zu entgehen, implementieren einige Schadprogramme mehr oder weniger komplexe Tarntechniken (*Obfuscation*[5]). Mit Tarntechniken ist hier die Veränderung des syntakischen Erscheinungsbildes der Malware durch selbst-modifizierenden Objektcode (auch *Mutation* genannt) gemeint und nicht die unbemerkte Ausführung im Hintergrund.
Folgende Methoden werden dabei unterschieden [58]:

Ungetarnt Die Malware unternimmt keinerlei Versuche, einer Entdeckung zu entgehen. Offensichtlich ist solche Malware leicht zu entdecken.

Verschlüsselung, Oligomorphie Veschlüsselte Malware besteht aus zwei Teilen: einer *Dekodierroutine* und einer *Nutzlast* (engl. *payload*). Bei der Ausführung wird die Nutzlast durch die Dekodierroutine entschlüsselt und die enthaltene Schadfunktion zur Ausführung gebracht (Abb. 2.1). Die Entdeckung ist durch die Suche nach der unverschlüsselten Dekodierroutine möglich.
Oligomorphe Malware verschlüsselt die Nutzlast während der Weiterverbreitung mit Hilfe der Verschlüsselungsroutine erneut. Die Dekodierroutine enthält dabei entweder eine endliche Menge von Schlüsseln, oder die Verschlüsselungsroutine erzeugt eine neue Dekodierroutine mit einem neuen Schlüssel aus der gleichen

[4] Selbst in der deutschsprachigen Literatur wird meist der Begriff Backdoor verwendet.
[5] engl. etwa für Verschleierung oder Verwirrung

Menge. Die Entdeckung ist wiederum durch Suche nach der Dekodierroutine möglich oder nach der Menge der Schlüssel[6].

Polymorphie *Polymorphe* Malware ist oligomorphe Malware auf deren Nutzlast bei jeder Weiterverbreitung eine Mutationsfunktion angewendet wird, die neue Dekodierroutinen erzeugt (teilweise sogar durch Auswahl aus einer Vielzahl von algorithmisch unterschiedlichen Dekodierroutinen) und die Nutzlast selbst verändert. Dies kann beispielsweise durch Vertauschen von Registern, Vertauschen von Instruktionen, Ersetzen durch andere äquivalente Instruktionen oder andere Methoden [2] erfolgen. Daraus resultiert eine potenziell unendliche Zahl von Schlüsseln. Für polymorphe Malware lässt sich kein einfaches geschlossenes Suchmuster mehr angeben.

Metamorphie Der aus Sicht der Malware-Autoren nächste logische Schritt ist *metamorphe* Malware. Diese Schadprogramme sind selbst polymorph und besitzen keine getrennte Dekodier- oder Verschlüsselungsroutine, vielmehr wird bei jeder Verbreitung eine neue Version der Malware erzeugt. Die Mutationsfunktionen von polymorpher Malware können ebenfalls verwendet werden.

2.3 Antivirenprogramme und -Techniken

Antivirenprogramme[7] werden zur Entdeckung, Identifikation[8] und zur Entfernung von Schadprogrammen auf Rechensystemen verwendet. Nachfolgend werden die Begriffe Entdeckung und Identifikation zu Erkennung zusammengefasst.

Erkennungsmethoden, die für gegebenen ausführbaren Objektcode entscheiden, ob er ein Schadprogramm enthält, lassen sich in *statische* und *dynamische Erkennung* unterteilen. Statische Erkennung versucht allein anhand des syntaktischen Erscheinungsbildes des Objektcodes eine Entscheidung über das Vorhandensein von Malware zu treffen. Es wird also keinerlei zu untersuchender Objektcode ausgeführt. Dies ist die klassische Domäne von Antivirenprogrammen, praktisch alle auf dem Markt befindlichen Produkte bieten diese Erkennungsmethode an.

[6] Auch die Suche nach einer endlichen Schlüsselmenge kann unpraktikabel sein, wird für kleine Anzahlen aber tatsächlich durchgeführt.

[7] Eine exakte Bezeichnung für Antivirenprogramme wäre eher *Anti-Schadprogramm*, ersteres hat sich jedoch allgemein durchgesetzt. Trotz der etwas irreführenden Namensgebung sind Antivirenprogramme in der Lage, alle Arten von Malware zu erkennen.

[8] Die Identifikation eines entdeckten Schadprogrammes kann in einem separaten Schritt oder als Seiteneffekt der Überprüfung erfolgen.

Bei dynamischer Erkennung wird der zu untersuchende Objektcode in einer sicheren Umgebung ausgeführt, um durch Verhaltensanalyse Rückschlüsse auf eine Infektion durch Malware ziehen zu können (beispielsweise durch maschinelles Lernen [38]). Eine sichere Umgebung kann zum Beispiel durch Sandboxing ([64], [5]) oder Emulation bereitgestellt werden [7]. Moderne Antivirenprogramme implementieren immer häufiger Verfahren zur dynamischen Erkennung [48].

Die Ausrichtung dieser Arbeit liegt in der Generierung von Signaturen für die statische Erkennung (siehe Kapitel 3), daher wird die dynamische Erkennung nicht weiter betrachtet.

2.3.1 Scanner

Zentrale Komponente eines Antivirenprogramms ist der *Scanner*, der die Dateien des Rechensystems auf das Vorhandensein von Schadprogrammen überprüft. Grundsätzlich wird nach Art des Aufrufs zwischen zwei Arten von Scannern unterschieden:

Bei Zugriff Ein *Zugriffsscanner* (engl. *on-access scanner*) wird kontinuierlich im Hintergrund ausgeführt und überprüft jeden Zugriff auf eine Datei. Da dieser Ansatz einen negativen Einfluss auf die Antwortzeiten des Rechensystems haben kann, erlauben einige Scanner die Überprüfung auf bestimmte Dateitypen oder Zugriffsarten einzuschränken (zum Beispiel nur auf ausführbare Dateien oder nur auf lesenden Zugriff).

Auf Anforderung *Anforderungsscanner* (engl. *on-demand scanner*) werden vom Benutzer selbst aufgerufen. Da die meisten Scanner eine Datenbank bekannter Bedrohungen nutzen, ist ein manueller Aufruf der Überpüfung beispielsweise sinnvoll, wenn eine neuere Version dieser Datenbank installiert wurde. Ferner ist ein Scan auf Anforderung sinnvoll, um aus Netzwerken heruntergeladene Dateien oder als infiziert vermutete Rechensysteme zu überprüfen.[9]

Moderne Antivirenprogramme bieten beide Aufrufarten an; auf Arbeitsplatz- und Heimcomputern kommen meist Zugriffsscanner zum Einsatz, da für diese nur minimale Benutzerinteraktion erforderlich ist.
Wird ein Schadprogramm erkannt, so wird dies dem Benutzer in Form eines Alarms

[9] Die Überprüfung oder Bereinigung eines infizierten Rechensystems sollte stets von bekannt nicht-infizierten Startmedien oder Rechensystemen aus erfolgen, um die Ergebnisse nicht durch getarnte Malware zu verfälschen.

mitgeteilt und der weitere Zugriff auf die Datei unterbunden. Häufig bekommt der Benutzer danach die Möglichkeit, die Datei zu löschen, für spätere Analyse in einen sicheren Bereich zu verschieben (auch als *Quarantäne* bezeichnet) oder – sofern das Antivirenprogramm dies unterstützt – diese zu *bereinigen* (engl. *cleaning*).

2.3.2 Signaturen

Zur Erkennung von Malware durch den Scanner eines Antivirenprogramms werden die Daten des Rechensystems auf charakteristische Muster überprüft. *Signaturen* sind Byte-Sequenzen solcher Muster, die ein Schadprogramm im Idealfall eindeutig identifizieren. Besteht eine Signatur aus einer Byte-Sequenz fester Länge, wird sie als *Scan-String* bezeichnet. Praktisch alle modernen Antivirenprogramme unterstützen außerdem Signaturen variabler Länge mit „*Don't care*"-Symbolen (so genannten *Wildcards*), die Platzhalter für beliebige Zeichen oder eine Anzahl beliebiger Zeichen darstellen. Formal betrachtet handelt es sich bei solchen Signaturen um reguläre Ausdrücke über dem Alphabet der durch ein Byte darstellbaren Zeichen. Einige Antivirenprogramme benutzen jedoch auch ausdrucksmächtigere Signaturbeschreibungssprachen (siehe Abschnitt 2.4).

Signaturen werden in der Regel in *Signaturdatenbanken* organisiert, die somit die dem jeweiligen Antivirenprogramm bekannten Bedrohungen repräsentieren.

Scannen mit Signaturen

Kapitel 1 lässt bereits vermuten, dass die Anzahl der Signaturen in einer Signaturdatenbank sehr groß sein kann (je nach Hersteller wird mit 50 000 bis 100 000 erkannten Malware-Arten geworben). Daher stellt sich die Frage nach der Effizienz einer vollständigen Suche nach allen bekannten Schadprogrammen.
Da die Überprüfung auf einzelne Signaturen zu aufwändig ist (schließlich werden bei n Signaturen auch ebensoviele Scan-Durchläufe benötigt), werden in der Praxis häufig Varianten des Aho-Corasick Algorithmus ([1], [39]) verwendet. Dieser Algorithmus erzeugt in einem Vorverarbeitungsschritt zunächst einen endlichen Automaten, der alle vorgegebenen Signaturen akzeptiert. Anschließend ist nur noch ein Durchlauf erforderlich, um alle Signaturen überprüfen zu können.

Weitere verbreitete Suchalgorithmen werden in [2] beschrieben, einige davon erfordern jedoch Erweiterungen für den Einsatz in diesem Kontext.[10]

2.4 Signaturbeschreibungssprachen

Die Ausdrucksmächtigkeiten der verschiedenen in der Praxis im Einsatz befindlichen Signaturbeschreibungssprachen unterscheiden sich teilweise stark voneinander, daher werden in den folgenden Abschnitten einige dieser Sprachen kurz vorgestellt. Da Programme zur Einbruchs- und Missbrauchserkennung (*Intrusion-Detection Systeme*, IDS) ebenfalls Signaturen verwenden, wird in Abschnitt 2.4.5 kurz auf die Unterschiede zu Signaturen für die statische Erkennung von Malware eingegangen.

2.4.1 Semantische Aspekte

Zur Einordnung der Ausdrucksmächtigkeit von Signaturbeschreibungssprachen wird im Folgenden das von Meier [42] aus der Theorie der aktiven Datenbanken abgeleitete Modell für die Semantik von Signaturen benutzt (Tabelle 2.1). Eine Signatur im Sinne des Semantikmodells ist eine Folge von komplexen – also zusammengesetzten – Ereignismustern. Die von Meier definierte Ordnungsrelation $<_{\text{time}}$ erfasst zudem die Nebenläufigkeit einer Menge von Ereignissen. Analog dazu bettet $<_{\text{trail}}$ Ereignisse in den größeren Kontext der Ereignistrails ein.
Das Modell erlaubt die Untersuchung einer Signaturbeschreibungssprache anhand verschiedener Merkmalsdimensionen, deren Aspekte und Ausprägungen systematisch überprüft werden können. Dies kann zum Beispiel durch Leitfragen nach den *Ereignismustern*, der *Schrittinstanzselektion* und dem *Schrittinstanzkonsum* erfolgen.

Für eine ausführliche Einführung in das Semantikmodell nach Meier sei auf [43] verwiesen. Die folgenden Abschnitte verwenden die dort angegebenen Begriffsdefinitionen.

[10] Zu erwähnen ist hier der ursprünglich für die Suche nach festen Textmustern konzipierte Knuth-Morris-Pratt-Algorithmus [34], der in angepasster Form in einer frühen Version des Antivirenprogramms ClamAV zum Einsatz kam.

<table>
<tr><th>Dimensionen</th><th>Aspekte</th><th>Ausprägungen</th></tr>
<tr><td rowspan="14">Ereignismuster</td><td rowspan="5">Typ und Reihenfolge</td><td>Sequenz</td></tr>
<tr><td>Disjunktion</td></tr>
<tr><td>Konjunktion</td></tr>
<tr><td>Simultan</td></tr>
<tr><td>Negation</td></tr>
<tr><td rowspan="3">Häufigkeit</td><td>Genau</td></tr>
<tr><td>Mindestens</td></tr>
<tr><td>Höchstens</td></tr>
<tr><td rowspan="2">Kontinuität</td><td>Kontinuierlich</td></tr>
<tr><td>Nicht-kontinuierlich</td></tr>
<tr><td rowspan="2">Nebenläufigkeit</td><td>Überlappend</td></tr>
<tr><td>Nicht-überlappend</td></tr>
<tr><td rowspan="2">Kontextbedingungen</td><td>Intra-Ereignis-Bedingungen</td></tr>
<tr><td>Inter-Ereignis-Bedingungen</td></tr>
<tr><td colspan="2" rowspan="3">Schrittinstanzselektion</td><td>Erste</td></tr>
<tr><td>Letzte</td></tr>
<tr><td>Alle</td></tr>
<tr><td colspan="2" rowspan="2">Schrittinstanzkonsum</td><td>Konsumierend</td></tr>
<tr><td>Nicht-konsumierend</td></tr>
</table>

Tabelle 2.1: Dimensionen, -aspekte und -ausprägungen im Semantikmodell nach Meier

2.4.2 SHEDEL

Die ereignisbasierte Signaturbeschreibungssprache SHEDEL (*Simple Hierarchcal Event DEscription Language*) [43] wurde als Alternative zu den regelbasierten Signaturen im IDS-Umfeld entwickelt.

Anstelle einer imperativen Beschreibung von Handlungsabfolgen, die zu einem Erkennungsergebnis führen, werden in SHEDEL ausschließlich Ereignisse als Abstraktion verwendet.

Ereignisse können in SHEDEL verallgemeinert oder spezialisiert werden, um hierarchische Beziehungen auszudrücken. Ein typisches Beispiel hierfür ist die Überprüfung auf fehlgeschlagene Anmeldeversuche in ein Microsoft Windows-basiertes Rechensystem. Fehlgeschlagene Anmeldungen werden in diesen Systemen mit einer Ereigniskennung versehen und in ein Ereignislog eingetragen. Die Ereigniskennungen haben die Nummern 529 bis 537, sowie 539 – je nachdem, ob der Benutzer eine falsche Kombination aus

```
EVENT LoginFailure {
        // Jeder auftretende Einzelschritt löst eine
        // LoginFailure-Ereignis aus.
        STEP failure529 INITIAL EXIT TYPE EVENTID529 // Anmeldeereignis 529
        // ...
        STEP failure539 INITIAL EXIT TYPE EVENTID539 // Anmeldeereignis 539
        FEATURES
                hostname = (failure529.host | ... | failure539.host),
                username = (failure529.user | ... | failure539.user),
                // ...
}
```

Listing 2.1: SHEDEL-Signatur für fehlgeschlagene Anmeldeversuche

```
EVENT AdministratorFailedLogin {
        STEP failure INITIAL EXIT TYPE LoginFailure
        CONDITIONS
                // Nur fehlerhafte Anmeldungen des Administrators
                failure.username == "Administrator"
        FEATURES
                // ...
}
```

Listing 2.2: Spezialisierung einer SHEDEL-Signatur

Benutzernamen und Kennwort oder ein abgelaufenes Kennwort eingegeben hat, oder ob das Benutzerkonto gesperrt oder abgelaufen ist. Die direkte Überprüfung auf jede einzelne dieser Ereigniskennungen für die bedingte Generierung eines Alarms (etwa für die Bedingung „*drei fehlgeschlagene Anmeldeversuche in den letzten 30 Sekunden*") erfordert das Testen von neun Signaturen. SHEDEL erlaubt hier die Definition einer Verallgemeinerung – die oben genannten Ereignisse können wie in Listing 2.1 zu dem Ereignis `LoginFailure` zusammengefasst werden.

Für den Fall, dass nur fehlgeschlagene Anmeldeversuche für einen bestimmten Benutzer (etwa „Administrator") einen Alarm generieren sollen, ist eine Spezialisierung des `LoginFailure`-Ereignisses möglich (siehe Listing 2.2).

Eine ausführliche Beschreibung der Sprachelemente von SHEDEL ist in [44] zu finden.

Einordnung in Semantikmodell

Die Sprachmittel von SHEDEL unterstützen die Beschreibung von Ereignismustern in Form von *Sequenzen* und *Disjunktionen*. Die Häufigkeit eines Ereignismusters kann in der Ausprägung *mindestens* modelliert werden, *genau* und *höchstens* durch zusätzliche Abbruchschritte. Der Aspekt *Kontinuität* wird ebenfalls voll unterstützt. Die hierarchische Definition von Ereignissen erlaubt zudem die Angabe von *Intra-* und *Inter-Ereignis-Bedingungen*.
Die Dimension *Schrittinstanzselektion* wird in der Ausprägung *erste* direkt unterstützt, *letze* und *alle* lassen sich durch entsprechende Schleifenkonstrukte realisieren. Die Dimension *Schrittinstanzkonsum* wird ebenfalls voll unterstützt.

Zusammengefasst lassen sich in SHEDEL – bis auf den Aspekt *Nebenläufigkeit*[11] sowie den Ausprägungen *Konjunktion* und *Simultanität* von Ereignismustern – alle Elemente aus dem in Abschnitt 2.4.1 vorgestellten Semantikmodell realisieren.

Relation zu endlichen Automaten

Es fällt auf, dass die einzelnen Sprachelemente der Struktur von endlichen Automaten nachempfunden sind. Ein komplexes `EVENT`-Element kann als Endlicher Automat interpretiert werden, dessen Zustandsübergange den `STEP`-Elemten entsprechen. Vorzustände eines Zustandsübergangs werden durch die Angabe eines `REQUIRES`-Ausdrucks beschrieben. `CONDITIONS`-Ausdrücke geben die Bedingungen für einen Zustandübergang an.

Listing 2.3 realisiert beispielhaft einen endlichen Automaten `NestFound`, der Zeichenketten akzeptiert, die die Buchstaben n, e, s und t in genau der angegebenen Reihenfolge enthält. Ein Ereignis ist in dieser Signatur das Lesen eines Zeichens der Zeichenkette.[12] Formal akzeptiert der Automat die durch den regulären Ausdruck[13] $\Sigma^* n \Sigma^* e \Sigma^* s \Sigma^* t$ beschriebene Sprache für ein geeignet gewähltes Alphabet Σ. Mit Σ^* wird – der Notation aus [62] folgend – der Kleenesche Abschluss über Σ bezeichnet. Diese Konstruktion entspricht genau dem „*Don't care*"-Symbol aus Abschnitt 2.3.2.

---

[11] *Überlappend* ist in SHEDEL der Standardmodus für Nebenläufigkeit.

[12] So konstruierte Signaturen sind für den praktischen Einsatz eher ungeeignet, da sie schnell unübersichtlich werden können. Hier eignen sich *Intra-Ereignis-Bedingungen* besser zur Beschreibung von Mustervergleichen. Dessen ungeachtet demonstriert das angegebene Listing die Flexibilität von SHEDEL.

[13] Definition siehe Abschnitt 2.4.4.

```
EVENT Char { // ...
}
EVENT NestFound {
        STEP read_n INITIAL TYPE Char
        STEP read_e         TYPE Char REQUIRES read_n
        STEP read_s         TYPE Char REQUIRES read_e
        STEP read_t EXIT    TYPE Char REQUIRES read_s
        CONDITIONS
                read_n.character == "n", read_e.character == "e",
                read_s.character == "s", read_t.character == "t"
        FEATURES // ...
}

EVENT Char { // ...
}
```

Listing 2.3: SHEDEL-Signatur zur Erkennung einer nicht notwendigerweise zusammenhängenden Sequenz von Zeichen

2.4.3 EDL

Um auch den Aspekt von nebenläufigen Ereignismustern in einer Signaturbeschreibungssprache modellieren zu können, wurde von Meier die Sprache EDL (*Event Description Language*) als logischer Nachfolger von SHEDEL entwickelt [45]. An die Stelle von Automaten tritt hierbei das Konzept von Signaturnetzen [43]. Da, wie von Meier beschrieben, Signaturnetze die Modellierung aller Ausprägungen des Semantikmodells erlauben, ist dies bei EDL ebenso der Fall.[14]

Endliche Automaten lassen sich durch Signaturnetze simulieren – Listing 2.4 zeigt eine zu Listing 2.3 äquivalente Signatur, die den gleichen Automaten realisiert. Die Zustände des Automats entsprechen den `PLACES`-Angaben. Zustandsübergänge werden mit `TRANSITIONS` definiert, der Operator `(-)` steht dabei für den Schrittinstanzkonsum *konsumierend*.

[14]Der Modus nicht-kontinuierlich des Aspekts Kontinuität wird in EDL durch Negationen umgesetzt.

```
EVENT Char { // ...
}

EVENT NestFound2 {
	PLACES
		start  { TYPE INITIAL }
		read_n { }
		read_e { }
		read_s { }
		read_t { TYPE EXIT }
	TRANSITIONS
		start  (-) read_n { TYPE Char CONDITIONS character == "n" }
		read_n (-) read_e { TYPE Char CONDITIONS character == "e" }
		read_e (-) read_s { TYPE Char CONDITIONS character == "s" }
		read_t (-) read_t { TYPE Char CONDITIONS character == "t" }
}
```

Listing 2.4: EDL-Signatur zur Erkennung einer Sequenz von Zeichen

2.4.4 Reguläre Ausdrücke

Signaturen und die Sprachen, die sie beschreiben, gehören zu den Betriebsgeheimnissen der Hersteller von Antivirenprogrammen. Daher ist es schwierig, allgemeine Aussagen über die im Einsatz befindlichen Signaturbeschreibungssprachen zu treffen. [29] und [47] lassen jedoch vermuten, dass – wie bereits in Abschnitt 2.3.2 angedeutet – fast alle modernen Antivirenprogramme Signaturen verwenden, deren zugehörige Signaturbeschreibungssprachen eine Obermenge von regulären Ausdrücken darstellen.

Definition 2.4.1 (Regulärer Ausdruck)**.** Die Menge der *regulären Ausdrücke* über einem endlichen Alphabet Σ wird durch folgende Regeln rekursiv definiert:

1. Die leere Sprache, die Sprache des leeren Wortes und die Sprache des einbuchstabigen Wortes a mit $a \in \Sigma$ werden durch die regulären Ausrücke $\emptyset, \epsilon$ und a dargestellt.

2. Für zwei reguläre Ausdrücke R_1 und R_2 sind auch deren Konkatenation $(R_1)(R_2)$ (multiplikative Schreibweise), die disjunkte Vereinigung $(R_1) + (R_2)$ und der Kleenesche Abschluss $(R_1)^*$ (auch Potenzbildung genannt) reguläre Ausdrücke.

3. Die endliche Anwendung der Regeln 1 und 2 erzeugt einen regulären Ausdruck.

Zur Vermeidung überflüssiger Klammern werden Klammern um elementare reguläre Ausdrücke weggelassen und entsprechend der mathematischen Operator-Rangfolge Prioritäten auf den oben gegebenen Regeln definiert: Reguläre Ausdrücke werden von links nach rechts gelesen, dabei wird die Potenzbildung $*$ zuerst ausgewertet, danach die Konkatenation und zuletzt die Vereinigung $+$. Somit steht der Ausdruck $ab^* + a$ eindeutig für $(a(b^*)) + a$.
Die Notation Σ^* wird auch *Wildcard* genannt und ist eine Kurzschreibweise für die Konkatenation einer beliebigen Anzahl (inklusive Null) beliebiger Buchstaben aus Σ.
Ist ein Wort $w \in \Sigma^*$ in der durch den regulären Ausdruck bestimmten Sprache enthalten, *akzeptiert* der reguläre Ausdruck das Wort w.

Weitere Einzelheiten zu den formalen Eigenschaften regulärer Ausdrücke sind in [62] und [53] zu finden. Im Zusammenhang mit der Theorie endlicher Automaten sind die folgenden Sätze interessant:

Satz 2.4.2 (ohne Beweis)**.** *Die Klasse der von endlichen Automaten akzeptierten Sprachen und die Klasse der regulären Sprachen stimmen überein.* □

Satz 2.4.3 (ohne Beweis)**.** *Genau die regulären Sprachen lassen sich durch reguläre Ausdrücke beschreiben.* □

Syntax

In der Praxis hat sich bei der Verwendung regulärer Ausdrücke eine andere als die oben angegebene Notation durchgesetzt. Wenn von regulären Ausdrücken die Rede ist, sind häufig so genannte *Perl-Compatible Regular Expressions* (kurz PCRE, oder auch *Perl-Syntax/Perl-kompatibel*) gemeint, die eine Reihe von Erweiterungen und Kurzschreibweisen einführen[15], einige davon werden hier anhand von Beispielen kurz erläutert.
Ein zu Listing 2.3 äquivalenter, regulärer Ausdruck lautet in Perl-kompatibler Syntax[16]: ⌜`.*n.*e.*s.*t`⌟. Der Punkt ⌜`.`⌟ steht wiederum für ein beliebiges Zeichen und der Stern

[15]Hierbei ist anzumerken, dass es sich bei Ausdrücken mit *Look-around assertions* (engl. etwa umherschauende Annahme/Behauptung) oder bedingten Ausdrücken nicht mehr um reine reguläre Ausdrücke handelt, da damit auch Aspekte kontext-sensitiver Sprachen erfasst werden können.
[16]Die Markierungen ⌜ und ⌟ dienen lediglich der optischen Abgrenzung zum Text und sind nicht Bestandteil des Ausdrucks.

⌜*⌟ für Potenzbildung. ⌜.*⌟ entspricht also Σ^*. Perl-kompatible reguläre Ausdrücke verwenden also neben den Buchstaben des Alphabets eine Reihe von *Metazeichen*. Dazu gehören neben dem Punkt und dem Stern noch eine Reihe weiterer Zeichen wie +, ?, |, (,), [,], {, } und \. Sind die Metazeichen ebenfalls Bestandteil des Alphabets, wird ihnen ein umgekehrter Schrägstrich \ vorangestellt, falls sie selbst in Wörtern vorkommen sollen.
Für die disjunkte Vereinigung von zwei Ausdrücken wird anstelle von + ein senkrechter Strich ⌜|⌟ verwendet. Der Ausdruck ⌜eins|zwei⌟ steht also für eins + zwei. Das Pluszeichen ⌜+⌟ in PCRE ist ähnlich der Potenzbildung, nur dass es statt für „beliebig viele Zeichen" für „mindestens ein Zeichen" steht. Der Ausdruck ⌜ab+⌟ ist also äquivalent zu abb^*. Zuletzt drückt ⌜?⌟ Optionalität aus, also entweder null oder ein Vorkommen des vorangegangenen Zeichens oder Teilausdrucks: ⌜a?⌟ entspricht damit $\epsilon + a$.

Für eine umfassende praktische Einführung in die Verwendung von regulären Ausdrücken sei auf [19] verwiesen.

Einordnung in Semantikmodell

Für eine Einordnung von regulären Ausdrücken in das Semantikmodell ist zunächst die Festlegung des Ereignisbegriffs erforderlich. In den folgenden Ausführungen wird unter *Ereignis* das Lesen eines Zeichens aus einem endlichen Alphabet durch einen endlichen Automaten verstanden. Ereignismuster sind demzufolge aufeinanderfolgende Zeichen.

Die Ausprägungen *Sequenz* und *Disjunktion* des Aspekts *Typ und Reihenfolge* werden direkt unterstützt, Sequenz durch zeichenweises Verarbeiten von Zeichen, Disjunktionen durch Vereinigung von regulären Ausdrücken. *Negationen* von Zeichenketten können durch Ausdrücke der Form ⌜[^r_1]?...[^r_n]?⌟ realisiert werden[17], wobei $r_1 \ldots r_n$ die zu negierenden Zeichen darstellen. Die Negation beliebiger regulärer Ausdrücke wird nicht unterstützt und erfordert eine sorgfältige Konstruktion eines entsprechenden negierten Ausdrucks mit der gewünschten Semantik. Die Ausprägung *Simultan* wird ebenfalls nicht unterstützt, da das zeitliche Auftreten von Zeichen immer strikt nacheinander stattfindet. Der Aspekt *Häufigkeit* wird in allen Ausprägungen unterstützt, die Ausprägung *genau* durch eine entsprechende Anzahl ⌜.⌟ in der Signatur, *mindestens* durch ⌜.{n,}⌟[18] und *höchstens* durch ⌜.{0,m}⌟[19].

---

[17]Der Ausdruck ⌜[^a]⌟ enthält eine weitere PCRE-Erweiterung, die jedes Zeichen ungleich a akzeptiert.
[18]Ein Ausdruck, der mindestens n beliebige Zeichen akzeptiert.
[19]Ein Ausdruck, der maximal m beliebige Zeichen akzeptiert.

Die Aspekte *Kontinuität* und *Nebenläufigkeit* werden beide nicht unterstützt, die Standardmodi sind *kontinuierlich* und *nicht-überlappend*. Dies liegt wiederum an der strikten zeitlichen Reihenfolge von Ereignissen.
Für den Aspekt *Kontextbedingungen* gibt es ebenfalls keine Unterstützung in regulären Ausdrücken. *Intra-Ereignis-Bedingungen* haben beim Lesen von Zeichen keine Bedeutung (einzelne Ereignisse sind merkmalsfrei), und *Inter-Ereignis-Bedingungen* erfordern die Verwendung von nicht-regulären Erweiterungen.
Die Dimensionen *Schrittinstanzselektion* und *Schritinstanzkonsum* werden voll untersützt, teilweise jedoch durch spezielle Modifikatoren (bei der Schrittinstanzselektion zum Beispiel *Greedy* und *Non-Greedy*, siehe [19]) und andere Erweiterungen (siehe Fußnote auf Seite 18). Die Standardmodi sind *erste* und *konsumierend*.

Signaturen mit Perl-kompatiblen regulären Ausdrücken eignen sich also, um in Eingabedaten auf vielfältige Art nach Zeichen oder Textmustern zu suchen.

2.4.5 Online-/Offline-Signaturen

In den vorherigen Abschnitten wurden mit SHEDEL und EDL zwei Beispiele für Signaturbeschreibungssprachen im IDS-Umfeld vorgestellt. Hierbei fällt auf, dass diese Sprachen einige Aspekte und Ausprägungen des Semantikmodells unterstützen, die sich mit den in Antivirenprogrammen verwendeten Signaturen nicht modellieren lassen.[20]
Es gibt also einen Unterschied zwischen Signaturbeschreibungssprachen für die Suche nach Zeichen, Byte-Sequenzen oder ganz allgemein nach Textmustern und solchen für die Suche nach Ereignismustern in einer Kette von Ereignissen. Dieser Unterschied erlaubt die Einordnung von Signaturbeschreibungssprachen in die Kategorien Online- und Offline-Signaturen.

Ausgehend von der Einordnung regulärer Ausdrücke in das Semantikmodell im vorherigen Abschnitt sind *Offline-Signaturen* Signaturen, deren Signaturbeschreibungssprache folgende Ausprägungen unterstützt:

- *Sequenz* und *Disjunktion* des Aspekts Typ und Reihenfolge (optional *Negation*),
- mindestens eine der Ereignishäufigkeiten *genau*, *mindestens* oder *höchstens*,
- *nicht-überlappend* als Standardmodus für Nebenläufigkeit und *kontinuierlich* als Standardmodus für Kontinuität,

[20]Meier beschreibt in [43] weitere Sprachen aus dem IDS-Umfeld, für die diese Aussage ebenfalls gilt.

- *Inter-Ereignis-Bedingungen* (optional),
- mindestens einen der drei Schrittinstanzselektionsmodi, sowie einen der beiden Konsummodi.

Online-Signaturen sind Offline-Signaturen, deren Signaturbeschreibungssprache zusätzlich die Ausprägungen

- *Negation* und optional *Konjunktion* oder *Simultan* des Aspekts Typ und Reihenfolge,
- die beiden Kontinuitätsmodi *kontinuierlich* und *nicht-kontinuierlich*,
- *überlappend* als Standardmodus für Nebenläufigkeit,
- Kontextbedingungen mit *Intra-* und *Inter-Ereignis-Bedingungen*

des Semantikmodells unterstützt.

Nach dieser Definition eignen sich Online-Signaturen insbesondere für die dynamische Erkennung von Malware oder zur Verhaltensanalyse in der Einbruchs- und Missbrauchserkennung. Wichtige Merkmale sind hierbei die unterstützten Kontextbedingungen, mit denen sich Ereignisse untereinander verknüpfen lassen, sowie die Möglichkeit, die Nebenläufigkeit von Ereignissen modellieren zu können, damit Teilereignisse komplexer Ereignisse in beliebiger Reihenfolge erkannt werden können.

Das in Kapitel 3 vorgestellte Verfahren zur Signaturgenerierung verwendet Offline-Signaturen, Online-Signaturen werden daher nachfolgend nicht weiter betrachtet.

2.4.6 ClamAV-Signaturen

ClamAV [11] stellt eine Besonderheit unter den Antivirenprogrammen dar: Es ist das einzige[21] frei im Quelltext erhältliche, nicht-kommerzielle Antivirenprogramm auf dem Markt. Aus diesem Grund ist auch eine Spezifikation der von ClamAV verwendeten Signaturbeschreibungssprache verfügbar [35].

Signaturen für ClamAV sind in CVD-Dateien (*ClamAV Virus Database*, CVD) organisiert, die als Containerformat[22] für verschiedene Signaturdatenbanken fungieren. Im Folgenden werden nur die verschiedenen Arten von ClamAV-Signaturen vorgestellt, das Dateiformat von CVD-Dateien wird in [35] beschrieben.

[21] Stand: 2008

[22] Ein *Containerformat* ist ein Dateiformat, das andere Dateiformate enthalten kann.

```
1 44d88612fea8a8f36de82e1278abb02f:68:eicar.com
```

Listing 2.5: MD5-Signatur für die Standard EICAR Anti-Malware-Testdatei [17]

Signaturformate

Der Scanner von ClamAV kann eine ganze Reihe unterschiedlicher *Signaturformate* verarbeiten. Ein *Format* für eine Signatur ist in diesem Zusammenhang analog zu einem Dateiformat zu verstehen. Alle unterstützten Signaturformate bestehen jeweils aus einer Zeile[23], einzelne Komponenten werden durch Trennzeichen (meist ein Doppelpunkt) voneinander getrennt:

MD5-Signaturen Signaturen in diesem Format bestehen aus dem Ergebnis der kryptographischen Hashfunktion *MD5* [50], angewendet auf die ausführbare Datei des Schadprogramms, das erkannt werden soll, sowie der Dateigröße und einem Bezeichner für die Malware. Das Ergebnis der Hashfunktion (auch *MD5-Hash* genannt) wird in hexadezimaler Schreibweise angegeben. Listing 2.5 zeigt ein Beispiel für eine solche Signatur.
Mit MD5-Signaturen können ungetarnte Schadprogramme erkannt werden. Da sie außerdem inkrementell während des Lesens einer zu überprüfenden Datei berechnet werden können und alle Bytes einer Datei beim Scannen gelesen werden, sind MD5-Signaturen auch nicht wesentlich ineffizienter als Signaturen zur Suche nach Textmustern.

MD5-Signaturen für PE-Sections Malware, deren ausführbare Datei im *Portable Executable*-Format (PE-Format, [49]) vorliegt, besteht aus mehreren Abschnitten, so genannten *PE-Sections*. ClamAV erlaubt hier die Angabe von MD5-Hashes über einzelne Abschnitte. Dabei wird die Größe der PE-Section, das Ergebnis der Hashfunktion sowie ebenfalls ein Bezeichner für die Malware angegeben. So spezifierte Signaturen eignen sich zum Beispiel für die Erkennung verschlüsselter oder einfacher oligomorpher Malware.

Whitelist-Signaturen Mit *Whitelist-Signaturen* lassen sich falsche Positive bei der Erkennung von Malware korrigieren. Passt eine Whitelist-Signatur auf eine zu überprüfende Datei, wird sie als „bekannt Nicht-Malware" eingestuft und die Erkennung

[23] ClamAV verwendet UNIX-typische Zeilenenden, das heißt, das Ende einer Zeile wird mit dem ASCII-Kontrollzeichen <LF> (dezimal: 10) markiert.

```
1 Worm.Bagle-zippwd-23:1:*:57935:54963:*:8:5:*
```

Listing 2.6: Archiv-Metadaten-Signatur für eine Variante des Bagle-Wurms

```
1 Trojan.Crypted-3:1:EP+0:68??????00e8??????00*68000000008b74242c89e581ecc00000008
2 9e70375008a06{-8}0fb6c0
```

Listing 2.7: Erweiterte Signatur für eine Variante des Trojanischen Pferds Crypted

mit der nächsten Datei fortgesetzt. Whitelist-Signaturen haben denselben Aufbau wie MD5-Signaturen, werden jedoch in einer anderen Datenbank gespeichert.

Archiv-Metadaten-Signaturen Schadprogramme werden häufig in Archivdateien verbreitet. Damit der Scanner diese – meist komprimierten – Dateien nicht erst extrahieren muss, um eine Überpüfung des Inhalts vornehmen zu können, lassen sich Signaturen für die *Metadaten* von Archivdateien angeben. Allgemein enthält jede (unverschlüsselte) Archivdatei ein Inhaltsverzeichnis von enthaltenen Dateien, das zusätzliche Informationen wie Größe, CRC-Prüfsumme, Position innerhalb des Archivs oder andere Angaben beinhaltet. Archiv-Metadaten-Signaturen spezifizieren ein Suchmuster über diesen Daten, das – mit Doppelpunkten voneinander getrennt – an einen Bezeichner für die Malware angehängt wird. Listing 2.6 zeigt eine solche Signatur, * ignoriert das jeweilige Datum.
Konzeptionell erzeugt der Scanner bei Bedarf einen regulären Ausdruck, mit dem in der Archivdatei nach charakeristischen Mustern für die entsprechende Malware gesucht wird.

Basis-Signaturformat Das einfachste, vom ClamAV-Scanner unterstützte Signaturformat besteht aus einem Bezeichner für die Malware, gefolgt von einem Gleichheitszeichen und einer hexadezimalen Signatur (siehe Seite 24).
Dieses Signaturformat gilt inzwischen als veraltet und wird nur noch für ältere Malware verwendet.

Erweitertes Signaturformat Seit Version 0.8 verwendet ClamAV das *erweiterte Signaturformat*. Listing 2.7 zeigt ein Beispiel. Das Format enthält einen Bezeichner für die Malware, einen Zieltyp, einen Offset und eine hexadezimale Signatur. Optional kann außerdem die für die Signatur erforderliche Scanner-Version angegeben werden.
Der *Zieltyp* gibt an, auf welche Arten von Dateien die Signatur angewendet werden

soll (siehe Tabelle 2.2, das ELF-Format [14] (*Executable and Linking Format*, ELF) ist ein dem PE-Format vergleichbares Dateiformat für ausführbare Dateien).
Ein *Offset* gibt eine Position innerhalb der Datei an, ab der nach der Signatur gesucht werden soll. Die Positionsangabe kann dabei als absolute Dezimalzahl oder mit einem der Modifikatoren aus Tabelle 2.3 erfolgen. Ab Version 0.91 wird außerdem die Angabe eines Offset-Intervalls unterstützt, Intervallstart und -ende werden dann durch ein Komma voneinander getrennt.

Zieltyp	Dateien
0	alle Dateien
1	ausführbare Dateien im PE-Format
2	eingebettete Skriptdateien (zum Beispiel VBA)
3	normalisiertes HTML
4	gespeicherte Email-Dateien
5	Graphikdateien
6	ausführbare Dateien im ELF-Format
7	normalisierte Textdateien

Tabelle 2.2: Zieltypen im erweiterten Signaturformat

Zieltyp(en)	Modifikator(en)	Bedeutung
0	*	Kein Offset, jede Position wird untersucht
0	n	absolute Byte-Position in der Datei
1, 6	EOF $- n$	Position relativ zum Dateiende
1, 6	EP $+ n$ / EP $- n$	relative Position zum Programmeinsprungspunkt
1, 6	S$x + n$ / S$x - n$	Position relativ zum Start von Abschnitt x

Tabelle 2.3: Offset-Modifikatoren im erweiterten Signaturformat

Hexadezimale Signaturen

Hexadezimale Signaturen sind in ClamAV reguläre Ausdrücke über hexadezimal kodierten Byte-Sequenzen. Ein Byte besteht in dieser Darstellung aus zwei aufeinanderfolgenden alphanumerischen Zeichen aus der Menge $\{0, \ldots 9, a, \ldots, f\}$, oder formal betrachtet: Ein hexadezimal kodiertes Byte ist ein Zeichen aus dem Alphabet $\Sigma = \{\text{hex}_2(i) | \forall 0 \leq i < 2^8\}$. Mit $\text{hex}_k(x)$ sei eine Funktion bezeichnet, die eine Zahl x auf ihre Hexadezimaldarstel-

lung mit k-Stellen und führenden Nullen abbildet[24].

Syntax

Hexadezimale ClamAV-Signaturen verwenden die Metazeichen ?, $\star$, |, {, }, -, [und]. Wie oben erwähnt, sind alle anderen Zeichen hexadezimal kodierte Bytes. Wildcards werden wie folgt dargestellt: Ein einzelnes beliebiges Byte wird von dem Ausdruck ⌜??⌟ akzeptiert[25], eine beliebige Anzahl beliebiger Bytes durch ⌜$\star$⌟. ⌜{n}⌟ akzeptiert genau n, ⌜{-n}⌟ höchstens n und ⌜{n-}⌟ mindestens n beliebige Bytes. Die disjunkte Vereinigung zweier Ausdrücke R_1 und R_2 wird in ClamAV-Syntax ⌜(R_1|R_2)⌟ geschrieben. ⌜(aa|bb)⌟ akzeptiert also entweder das Byte aa oder das Byte bb.

Einordnung in Semantikmodell

Da es sich bei hexadezimalen ClamAV-Signaturen um reguläre Ausdrücke handelt, ist die Einordnung in das Semantikmodell nach den obigen Ausführungen zu einem großen Teil bereits behandelt. Die Einordnung der MD5-Signaturformate, der Whitelist-Signaturen und des Archiv-Metadaten-Signaturformats ist jedoch nicht offensichtlich. Die MD5-Signaturformate können als sehr spezifische hexadezimale Signaturen geschrieben werden, wenn man die Kollisionen der Hashfunktion MD5 nicht berücksichtigt.[26] Eine solche Signatur besteht dann einfach aus allen Bytes der zu erkennenden Malware. Whitelist-Signaturen sind die Negation von MD5-Signaturen, allgemein wird die Negation von ClamAV jedoch nicht unterstützt.
Das Archiv-Metadaten-Sigaturformat bereitet wiederum keine Probleme, da dieses vergleichsweise einfach in hexadezimale Signaturen transformiert werden können: Statt nur die Metadaten anzugeben, wird mit einer transformierten Signatur nach den entsprechenden Bytes der Metadaten in einer festgelegten Reihenfolge gesucht. Dies funktioniert, da Archivdateien ein festes Dateiformat haben, die Reihenfolge der Metadaten also bekannt ist.

---

[24] $\text{hex}_k(x)$ ist für $\log_{16}(x) > k$ hier nicht definiert.

[25] Seit Version 0.91rc1 wird außerdem das Akzeptieren der vier höchst- beziehungsweise niederwertigsten Bits eines Bytes unterstützt. Der Ausdruck ⌜?0⌟ akzeptiert ein Byte, dessen vier niederwertigen Bits beliebig und dessen höchstwertigen Bits auf Null gesetzt sind. ⌜0?⌟ akzeptiert den entgegengesetzten Fall.

[26] Um diese ebenfalls zu berücksichtigen, muss eine hexadezimale Signatur konstruiert werden, die alle Byte-Sequenzen akzeptiert, die auf den gleichen MD5-Wert abbilden. Da dies einem Brute-Force-Angriff auf die Hashfunktion entspricht, ist dieses Vorgehen allerdings nicht praktikabel.

Im Vergleich zu Perl-kompatiblen regulären Ausdrücken werden die Schrittinstanzselektionsmodi *letzte* und *alle*, sowie der Konsummodus *nicht-konsumierend* nicht unterstützt.

Tabelle 2.4 fasst die Ergebnisse der semantischen Betrachtungen zusammen.

2.5 Analyse von Malware

Vor der Erstellung einer Signatur zur Erkennung einer spezifischen Malware durch ein Antivirenprogramm ist zunächst eine umfassende Analyse des jeweiligen Schadprogramms erforderlich. Dabei wird nach charakteristischen Merkmalen der Malware gesucht, um daraus später eine Signatur zu erstellen. Dies geschieht üblicherweise manuell oder halb-automatisch.
Die folgenden Abschnitte richten den Blick auf einige Softwarewerkzeuge und Analysemethoden, bevor in Kapitel 3 ein automatisiertes Verfahren dieses Prozesses vorgestellt wird, welches auf den vorgestellten Methoden aufbaut.

2.5.1 IDA Pro

Schadprogramme liegen bei ihrer Erscheinung – also ihrer ersten Verbreitung – nicht im Quelltext, sondern in Form von Objektcode vor, der je nach Art der Malware unterschiedlich gut getarnt sein kann. Die Analyse der Binaries solcher Programme erfolgt mit einer symbolischen Repräsentation der vom Prozessor[27] des Rechensystems ausgeführten Instruktionen, auch *Assemblersprache* (engl. *assembly language*) oder *Assembler* genannt. Ein *Disassembler* ist ein Programm, das Objektcode in diese symbolische Darstellung übersetzt, wobei der Objektcode meist zunächst aus einer ausführbaren Datei extrahiert wird (der Disassembler simuliert hier gewissermaßen das Ladeprogramm des Betriebssystems).
Da Assemblersprache schwierig zu lesen und zu verstehen ist[28], implementieren Disassembler eine Reihe von Techniken, die die Assemblerdarstellung eines Programms lesbarer gestalten.

[27] Da ein bestimmter Ausführungspfad eines Programms logisch immer nur auf einer Recheneinheit ausgeführt wird, genügt es, hier von *einem* Prozessor zu sprechen, selbst wenn das Programm Nebenläufigkeit und mehrere Prozessoren benutzt.

[28] Dies hängt natürlich stark von der Expertise des Benutzers ab, ist aber in etwa vergleichbar mit einer Wort-für-Wort-Übersetzung eines Textes.

Dimension/Aspekt		Ausprägung	Online	Offline	SHEDEL	EDL	PCRE	ClamAV
Ereignismuster	Typ und Reihenfolge	Sequenz	✓	✓	✓	✓	✓	✓
		Disjunktion	✓	✓	✓	✓	✓	✓
		Konjunktion	□	∅	∅	✓	∅	∅
		Simultan	□	∅	∅	✓	∅	∅
		Negation	✓	□	✓	✓	✓[a]	∅[b]
	Häufigkeit	Genau	✓[c]	✓[c]	✓	✓	✓	✓
		Mindestens			✓	✓	✓	✓
		Höchstens			✓	✓	✓	✓
	Kontinuität	Kontinuierlich	✓	∅	✓	✓	∅[d]	∅[d]
		Nicht-kont.	✓	∅	✓	✓	∅	∅
	Nebenläufigkeit	Überlappend	✓	∅	✓	✓	∅	∅
		Nicht-überl.	□	∅	∅	✓	∅[d]	∅[d]
	Kontext-bedingungen	Intra-EB	✓	∅	✓	✓	∅	∅
		Inter-EB	✓	□	✓	✓	✓[e]	∅
Schrittinstanzselektion		Erste	✓[c]	✓[c]	✓	✓	✓	✓
		Letzte			✓	✓	✓	∅
		Alle			✓	✓	✓	∅
Schrittinstanzkonsum		Konsumierend	✓	✓	✓	✓	✓	✓
		Nicht-kons.			✓	✓	✓	∅

✓ Unterstützt ∅ Nicht unterstützt □ Optional

[a] Negation muss konstruiert werden

[b] Whitelist-Signaturen sind Negation von MD5-Signaturen

[c] Mindestens eine Ausprägung erforderlich

[d] Standardmodus

[e] Nur mit Look-Around-Ausdrücken

Tabelle 2.4: Überblick über die vorgestellten Signaturbeschreibungssprachen und deren Kategorien

Der Industriestandard im Bereich des Reverse-Engineering und der Analyse von Malware ist IDA Pro [25] – ein erweiterbarer Disassembler, der zur Veranschaulichung von Programmen deren Kontrollfluss graphisch darstellen kann. Die Abkürzung IDA steht für *Interactive Disassembler* (engl. für interaktiver Disassembler), der Name soll verdeutlichen, dass der Benutzer während seiner Analyse eine Vielzahl von Darstellungsparametern interaktiv ändern kann. Ein Beispiel hierfür ist die manuelle Markierung von Datenbereichen im ausführbaren Code – eine Aufgabe, die allgemein mit Rechnerhilfe nicht entscheidbar ist[29] und daher mit Heuristiken gelöst wird.
IDA Pro unterstützt zahlreiche Prozessor- und Befehlsarchitekturen (die gebräuchlicheren sind IA-32[30], AMD64, ARM, PowerPC, MIPS und SPARC); die Analyse von Programmen für unterschiedliche Plattformen erfolgt somit in einer einheitlichen Umgebung. Ein nicht-technischer Überblick über die in IDA Pro enthaltenen Funktionen ist in [24] zu finden, alle Angaben beziehen sich auf die aktuelle Version[31] 5.2.
Abbildung 2.2 zeigt die graphische Benutzeroberfläche (*Graphical User Interface*, GUI) unter dem Microsoft-Windows-Betriebssytem. Neben der Windows-Version sind außerdem Versionen für GNU/Linux und Mac OS X verfügbar.

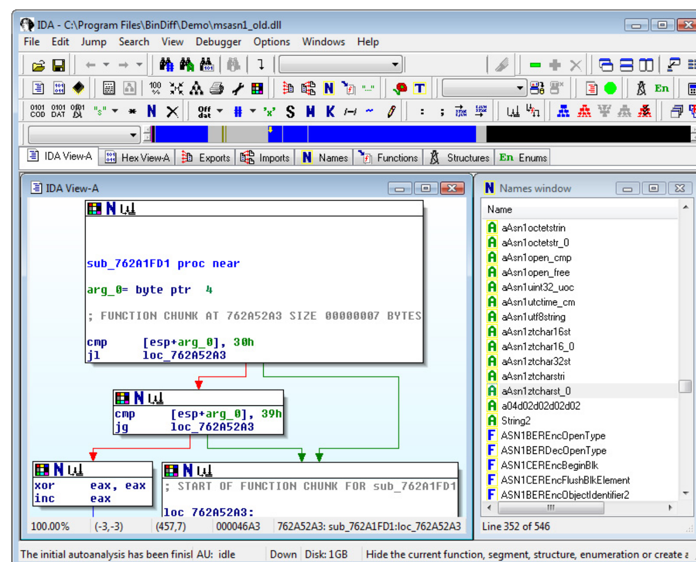

Abbildung 2.2: IDA Hauptfenster mit geöffnetem Aufrufgraph

[29] Trennung von Programm und Daten ist äquivalent zum Halteproblem. Eine Ausnahme stellen die Harvard-Architekturen dar, bei denen Programm- und Datenspeicher immer klar getrennt sind.
[30] Meist als *x86* oder *x86-32* bezeichnet.
[31] Stand: Mai 2008

```
1  #include <idc.idc>
2
3  static main() {
4    auto ea, x;
5    for (ea = NextFunction(0); ea != BADADDR; ea = NextFunction(ea)) {
6      Message("Function at %08lX: %s", ea, GetFunctionName(ea));
7      x = GetFunctionFlags(ea);
8      if (x & FUNC_NORET) Message(" Noret");
9      if (x & FUNC_FAR)   Message(" Far");
10     Message("\n");
11   }
12 }
```

Listing 2.8: IDC-Script für die Ausgabe einer Funktionsliste

IDA Pro ist modular aufgebaut und kann mit *Plugins* (engl. etwa *Erweiterungen*) um zusätzliche Funktionalität erweitert werden, wie zum Beispiel die Unterstützung für neue Prozessortypen oder Dateiformate für ausführbare Programme. Technisch handelt es sich dabei um spät gebundene Programmbibliotheken (*shared libraries*), die bei Bedarf aufgerufen werden. Die in den Abschnitten 2.5.2 und 4.3.1 vorgestellten Werkzeuge sind Beispiele für IDA-Pro-Plugins.
Um die Plugin-Entwicklung zu unterstützen, bietet IDA Pro eine Programmierschnittstelle (*Application Programming Interface*, API), die von Plugins zur Bereitstellung ihrer Funktionalität aufgerufen werden kann. Fast alle über die graphische Benutzeroberfläche verfügbaren Befehle lassen sich so aufrufen.[32]

Für einfache Aufgaben bietet IDA Pro neben der Programmierschnittstelle eine C-ähnliche Skriptsprache an (so genannte *IDC-Scripts*). Listing 2.8 zeigt ein solches Skript zur Ausgabe einer Liste von Funktionen in der aktuell geladenen ausführbaren Datei. Auch hier sind viele der über die Benutzeroberfläche erreichbaren Funktionen verfügbar.[32] Für die Stapelverarbeitung von Dateien können IDC-Scripts ebenfalls verwendet werden.

Eine wesentliche Erleichterung bei der Analyse von Programmen ist außerdem die automatische Analyse beim Laden von ausführbaren Dateien – IDA Pro erzeugt für jede geladene Datei eine Datenbank (*IDA Database*, IDB) mit Metainformationen wie zum

[32]Eine bemerkenswerte Ausnahme ist die Funktion zum Öffnen von Dateien. Diese lässt sich ausschließlich über die GUI aufrufen.

Beispiel der Dateistruktur (für ausführbare Dateien im PE- oder ELF-Format sind dies beispielsweise Informationen über die einzelnen Abschnitte), Einsprungadressen, gefundenen Programmfunktionen und Einstellungen des Benutzers[33]. Die automatische Analyse versucht außerdem deskriptive Namen für verwendete Bibliotheksfunktionen zu finden und fügt Kommentare zu bekannten Programmfragmenten hinzu. Für die Erkennung von Bibliotheksfunktionen werden vom Hersteller so genannte FLIRT-Signaturen (*Fast Library Identification and Recognition Technology*, FLIRT) bereitgestellt und gepflegt. Die so enstandene Sicht auf das Binary kann dem ursprünglichen Quelltext recht nahe kommen[34] und bildet einen guten Ausgangspunkt für das weitere Reverse-Engineering des jeweiligen Programms oder Objektcodes.

2.5.2 BinDiff

Das Einspielen von Programmaktualisierungen (engl. *Updates*), die bekannt gewordene Sicherheitslücken in bestehenden Produkten beheben, sind unbestreitbar wichtig für die Sicherheit von Rechensystemen. Im produktiven Einsatz in Unternehmen spielt jedoch häufig noch das betriebliche Kontinuitätsmanagement (engl. *Business Continuity Management*, BCM, siehe auch [63]) eine wichtige Rolle. Zur Sicherung der Kontinuität von Geschäftsprozessen zählen in diesem Zusammenhang auch die Einhaltung von Wartungsintervallen und die Minimierung von Ausfallzeiten der verwendeten Rechensysteme. Updates können deshalb in der Regel nicht sofort auf kritischen Systemen eingespielt werden, was potenziellen Angreifern einen zeitlichen Vorteil verschafft. Für eine Risikoabschätzung im Unternehmen kommt erschwerend hinzu, dass Software-Hersteller häufig keine oder nur unzureichende Details über geschlossene Sicherheitslücken veröffentlichen.

Mit BinDiff der Firma Zynamics [66] steht ein Softwarewerkzeug zur Verfügung, das es erlaubt, Änderungen zwischen zwei ausführbaren Dateien sichtbar zu machen. Diese Kernfunktionalität unterstützt ein manuelles Reverse-Engineering, um Gemeinsamkeiten und Unterschiede von Varianten einer Malware herauszuarbeiten[35] oder die in Programmaktualisierungen enthaltenen Änderungen zu rekonstruieren.
Im Fall von Programmaktualisierungen ermöglicht es BinDiff zu entscheiden, ob ein

[33] Insbesondere vom Benutzer eingetragene Kommentare zu einzelnen Teilen der Datenbank.
[34] Dies trifft so nur auf Programme zu, die keine Maßnahmen zur Erschwerung von Reverse-Engineering ergreifen.
[35] Diese Aufgabe kann mit dem in Abschnitt 2.6.1 vorgestellten Softwarewerkzeug bis zum einem gewissen Grad automatisiert werden.

Update unmittelbar auf den jeweiligen Produktivsystemen eingespielt werden muss, da es eine oder mehrere kritische Sicherheitslücken schließt, oder ob es sich bei dem Update um die reguläre Behebung von Fehlern im zu aktualisierenden Programm handelt, so dass es in den geplanten Wartungsintervallen eingespielt werden kann.
Gefundene Gemeinsamkeiten von Varianten einer Malware können hingegen – wie in Kapitel 3 gezeigt wird – für die Generierung von Signaturen für Antivirenprogramme herangezogen werden. Außerdem können Analyseergebnisse von einer ausführbaren Datei auf eine andere übertragen werden, um wiederholte Arbeitsschritte zu reduzieren.[36]
Die Resistenz des verwendeten Algorithmus auch gegen umfangreichere Änderungen zwischen zwei ausführbaren Codes auf Instruktionsebene macht den Einsatz von BinDiff auch interessant, um Beweise für den Diebstahl geistigen Eigentums oder Patentrechtsverletzungen zu sammeln.
Die aktuelle Version[37] von BinDiff ist 2.0, alle Angaben beziehen sich auf diese Version.

Plugin für IDA Pro

BinDiff ist als Plugin für den Disassembler IDA Pro[38] realisiert. Unterschiede zwischen zwei Binaries können mit der graphischen Benutzeroberfläche von IDA in Listendarstellung angezeigt werden; BinDiff bietet allerdings eine eigene Benutzeroberfläche mit einer erweiterten Darstellung. Hierbei sind sowohl die visuelle Darstellung von Unterschieden und Gemeinsamkeiten im Kontrollflussgraph, als auch die Darstellung in Assemblersprache möglich (siehe Abbildung 2.3). Die Art der Darstellung ist vergleichbar mit der üblichen Darstellung von Unterschieden in Programmquelltexten.

2.5.3 Graph-basiertes Vergleichen von Objektcode

Der in BinDiff verwendete Algorithmus zum Vergleichen von ausführbarem Code zweier Programme wird in [18] und [16] beschrieben und im Folgenden kurz skizziert.

Wird ein Programmquelltext von zwei unterschiedlichen Compilern oder zweimal mit jeweils unterschiedlichen Einstellungen kompiliert, ändert sich die Darstellung des resultierenden Objektcodes in Assemblersprache teilweise erheblich.[39] Zu beobachten ist

[36] Dies stellt besonders bei umfrangreich kommentierten IDA-Pro-Datenbanken eine Arbeitserleichterung dar.
[37] Stand: Mai 2008
[38] Vorraussetzung ist mindestens Version 4.9.
[39] Dies gilt erst recht, wenn die in Abschnitt 2.2.2 genannten Tarntechniken zum Einsatz kommen.

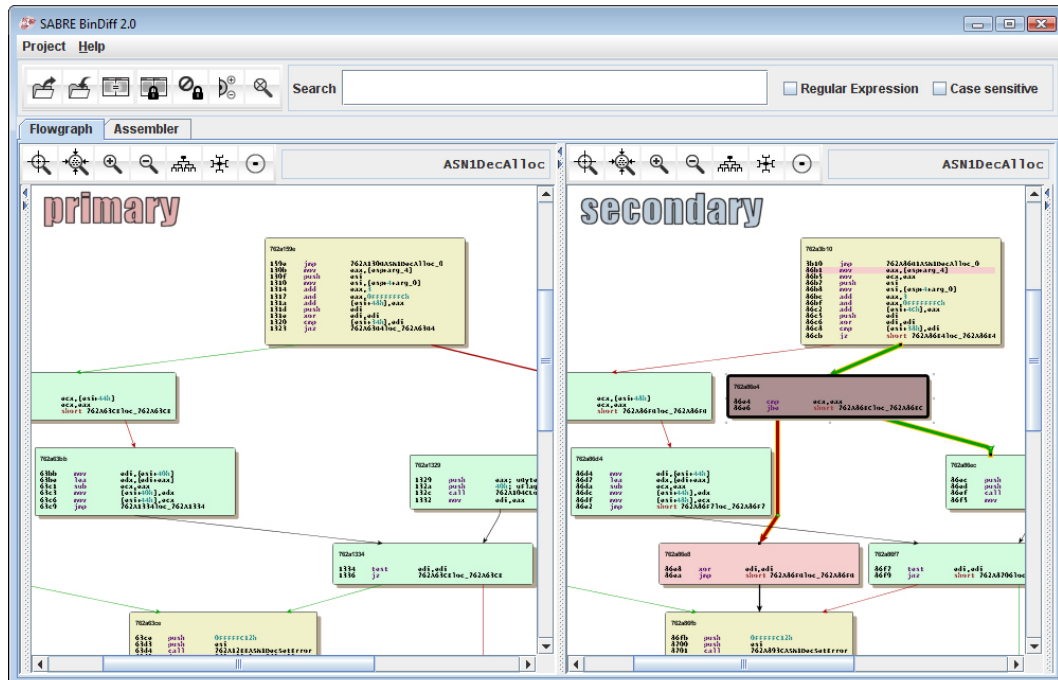

Abbildung 2.3: Visuelle Darstellung von Unterschieden zweier Funktionen in BinDiff

dabei, dass die Semantik des betreffenden Programms weitestgehend erhalten bleibt, sich das syntaktische Erscheinungsbild wie eben erwähnt aber deutlich verändert.
Jedes Verfahren, das den Vergleich zweier Varianten eines Programms (oder zweier Programme mit einem signifikanten Anteil von gemeinsamen Objektcode) erlaubt, muss mit einem unterschiedlichen syntaktischen Erscheinungsbild der Programme umgehen können. Unter anderem können die folgenden Veränderungen zwischen zwei Varianten von Objektcode auftreten:

Unterschiedliche Zuordnung von Registern Abhängig von Compilereinstellungen oder Quelltextänderungen werden identischen Instruktionen unterschiedliche Register zugeordnet.

Vertauschung der Instruktionsreihenfolge Je nach vom Compiler verwendeten internen Modell der Befehls-Pipeline des Zielprozessors können Instruktionen in unterschiedlicher Reihenfolge angeordnet werden.

Vertauschung von Verzweigungen Häufig kann die Reihenfolge von bedingten Sprungzielen vom Compiler vertauscht worden sein; die Bedingung des Sprungs wird

dabei negiert.

Außer den oben genannten sind noch eine ganze Reihe weiterer Veränderungen denkbar, die einen Vergleich auf Instruktionsebene erschweren. Die Implementierung von BinDiff verfolgt daher einen anderen Ansatz: Ausführbare Dateien werden strukturell auf Basis ihrer Aufruf- und Kontrollflussgraphen verglichen, ein Binary wird dabei als „Graph von Graphen“ aufgefasst.

Funktionen und Basic-Blocks

Formal besteht eine ausführbare Datei A aus einer Menge $F_{(A)} = \{f_{(A),1}, \ldots, f_{(A),n}\}$ von Funktionen. Jede Funktion entspricht dabei ihrer symbolischen Assembler-Darstellung der im ursprünglichen Quelltext definierten Funktion. Der *Aufrufgraph* (engl. *call graph*, auch CG) einer ausführbaren Datei ist ein gerichteter Multigraph $G_{(A)} = (F_{(A)}, E_{(A)})$ mit der Menge der Funktionen $F_{(A)}$ als Knotenmenge und der Kantenmenge $E_{(A)} \subset F_{(A)} \times F_{(A)}$. Eine Kante zwischen zwei Funktionen $f_{(A),i}$ und $f^{(A),j}$ impliziert, dass $f_{(A),i}$ einen Unterprogrammaufruf von $f_{(A),j}$ enthält. Die Funktionen $f_{(A),i} \in F_A$ werden wiederum als gerichtete Graphen aufgefasst und *Kontrollflussgraphen* (engl. *control flow graph*, auch CFG) genannt. Der Kontrollflussgraph einer Funktion $f_{(A),i}$ ist gegeben durch $G_{(A),i} = (B_{(A),i}, E_{(A),i})$ mit Knotenmenge $B_{(A),i} = \{b_{(A,i),1}, \ldots, b_{(A,i),m}\}$ und Kantenmenge $E_{(A),i} \subset B_{(A),i} \times B_{(A),i}$. Die Elemente $b_{(A,i),j}$ der Knotenmenge werden als *Basic Blocks*[40] bezeichnet. Eine Kante zwischen zwei Basic-Blocks $b_{(A,i),j}$ und $b_{(A,i),k}$ impliziert eine lokale Verzweigung oder einen bedingten Sprung im Programm. Schließlich besteht ein Basic-Block $b_{(A,i),j} = \{i_{(A,i,j),1}, \ldots, i_{(A,i,j),l}\}$ aus einer Folge von Instruktionen, an deren Ende ein bedingter Sprung oder ein Rücksprung aus einem Unterprogramm steht. Da Adressen innerhalb einer ausführbaren Datei eindeutig sind, werden Funktionen, Basic-Blocks und Instruktionen jeweils mit ihren Startadressen identifiziert.
Eine ausführbare Datei kann also als hierarchische Graph-Struktur aufgefasst werden, deren Elemente Aufrufgraphen, Kontrollflussgraphen und Instruktionsfolgen[41] sind.

[40] engl. für *Basisblöcke*, nachfolgend wird nur noch der englische Begriff verwendet

[41] Eine Instruktionsfolge kann als primitiver, gerichteter, azyklischer Graph aufgefasst werden. Die Knoten der einzelnen Instruktionen werden ihrer Reihenfolge entsprechend mit Kanten verbunden.

Strukturelles Vergleichen

Definition 2.5.1 (Graph-Isomorphismus). Zwei Graphen $G = (V, E)$ und $H = (V', E')$ heißen *isomorph* zueinander, falls eine bijektive Abbildung $\varphi : V \to V'$ existiert, so dass für alle $(e, f) \in E$ mit $e \neq f$ gilt: $(\varphi(e), \varphi(f)) \in E' :\Leftrightarrow (e, f) \in E$.

Für den strukturellen Vergleich zweier gegebener ausführbarer Dateien A und B mit $|F_{(B)}| \leq |F_{(A)}|$ wird zunächst nach einer Einbettung des Aufrufgraphen $G_{(B)}$ in $G_{(A)}$ gesucht. Da das Finden solcher Einbettungen algorithmisch aufwändig ist, wird in BinDiff nach Graph-Isomorphismen gesucht.
Zunächst gilt, sind die beiden Dateien strukturell identisch, dass ein Isomorphismus $\varphi : F_{(A)} \to F_{(B)}$ existiert, der die Knoten und Kanten der beiden Aufrufgraphen einander zuordnet, sowie weitere Isomorphismen $\varphi_i : B_{(A),i} \to B_{(B),i}$ mit $0 < i \leq |F_{(B)}|$, die dieselbe Funktion für die Kontrollflussgraphen erfüllen. In der Praxis kommt dieser Fall allerdings höchst selten vor, da selbst sehr ähnliche ausführbare Dateien meist eine unterschiedliche Anzahl von Funktionen aufweisen. Unterstützt der verwendete Compiler *Inlining* von Funktionen[42], kann es vorkommen, dass selbst für zwei Kompilate desselben Quelltextes keine Isomorphismen existieren.

Die direkte Verwendung von Graph-Isomorphismen eignet sich also nur bedingt für den Vergleich von zwei ausführbaren Dateien. In [16] werden daher noch *Selektoren* und *Eigenschaften* auf den jeweiligen Graphen definiert, die eine iterative Konstruktion von Graph-Isomorphismen unter heuristisch definierten Eigenschaften erlaubt.

Definition 2.5.2 (Selektor). Ein *Selektor* auf zwei Graphen $G = (V, E)$ und $H = (V', E')$ ist definiert als Abbildung $s : V \times \mathcal{P}(V') \to V' \cup \emptyset$, die entweder ein Element aus der Knotenmenge des zweiten Graphen oder die leere Menge auswählt. $\mathcal{P}(V)$ bezeichnet hierbei die Potenzmenge von V.

In BinDiff werden Selektoren verwendet, um den zu einem Knoten V ähnlichsten Knoten auszuwählen. Falls es mehrere Knoten mit gleicher Ähnlichkeit gibt, wird die leere Menge ausgewählt. Als Ähnlichkeitsmaß wird dabei eine vereinfachte Form des Small-Primes-Product (SPP) – sowohl auf Funktions-, als auch auf Basic-Block-Ebene – verwendet [16]. Intuitiv ist klar, dass die Wahrscheinlichkeit, die leere Menge auszuwählen, mit der Anzahl der Elemente in der Knotenmenge steigt. Um diese Wahrscheinlichkeit zu senken,

[42] *Inlining* bezeichnet das Einfügen der Instruktionen einer Funktion an jeder Stelle, an der sie aufgerufen wird.

werden Eigenschaften verwendet, die die Anzahl der Elemente der Knotenmengen reduzieren.

Definition 2.5.3 (Eigenschaft). Eine *Eigenschaft* π zweier gegebener Graphen $G = (V, E)$ und $H = (U, E')$ ist eine Abbildung $\pi : V \times U \to V' \times U'$ mit $V' \subset V$ und $U' \subset U$.

Eine Eigenschaft wählt also Teilmengen der Knotenmengen der gegebenen Graphen aus. Für Aufrufgraphen und Kontrollflussgraphen lassen sich verschiedene Eigenschaften definieren. Beispiele sind die Auswahl von Knoten

- die gleiche Namen oder Verweise auf die gleichen Datenbereiche enhalten,
- mit jeweils gleichem Small-Primes-Product,
- die denselben Unterprogrammaufruf enthalten oder
- Knoten mit gleichen Eingangs- oder Ausgangsgraden.

Die Konstruktion eines initialen Isomorphismus kann nun durch die Anwendung der Selektoren und Eigenschaften erfolgen. Dieser Isomorphismus lässt sich dann iterativ verfeinern, bis keine Änderung mehr auftritt. Dieses Verfahren wird in [16] im Detail beschrieben.

Zusammenfassung

Ein nach dem oben beschriebenen Verfahren konstruierter Isomorphismus kann anschließend wie in Abbildung 2.3 graphisch dargestellt werden, oder zur Berechnung eines Ähnlichkeitsmaßes zwischen zwei ausführbaren Dateien verwendet werden.
Die Verwendung des Small-Primes-Product erlaubt die Abstraktion von der Assembler-Darstellung: Selbst ausführbare Dateien, die für verschiedene Prozessor- oder Befehls-architekturen kompiliert wurden, können so miteinander strukturell verglichen werden [65]. Ein weiterer Vorteil des SPP liegt in der Resistenz gegen übliche Veränderungen des ausführbaren Codes auf Instruktionsebene. Eine Vertauschung von Instruktionen oder Registern ändert das von BinDiff verwendete Ähnlichkeitsmaß nicht oder nur sehr gering.

2.6 Automatische Klassifizierung von Malware

Das Erstellen von Signaturen für Stämme von Malware erfordert zunächst die Einordnung der ausführbaren Dateien einer gegebenen Menge von Schadprogrammen in die jeweiligen Familien. Allgemein wird die Einordnung von ausführbaren Dateien in eine Familie als *Klassifikation* bezeichnet.
Die Klassifikation von Malware wird üblicherweise auf Grundlage einer ausführlichen manuellen Analyse vorgenommen, ein Vorgang der äußerst zeitaufwändig und fehleranfällig ist. Automatische Klassifikationsmethoden versprechen hier eine signifikante Verkürzung des Analyseprozesses. Analog zu den in Abschnitt 2.3 genannten Erkennungsmethoden der Antivirenprogramme lassen sich zwei Arten von Klassifikationsmethoden unterscheiden: *statische* und *dynamische Klassifikation*. Die dynamischen Klassifikation verfolgt hier den Ansatz, aus dem Verhalten der zu untersuchenden Schadprogramme Rückschlüsse auf eine Familienzugehörigkeit zu ziehen. Beispiele für derartige Verfahren sind [38] und [4]. Die folgenden Ausführungen betrachten die statische Klassifikation.

Statische Klassifikation von Malware arbeitet ebenso wie statische Erkennung auf den ausführbaren Dateien oder dem Objektcode von Malware. Das Verfahren in [9] ist ein Beispiel für die statische Klassifikation unter Verwendung des *normalisierten Kompressionsabstands* (engl. *normalized compression distance*, NCD), und in [20] wird ein Verfahren vorgeschlagen, das dem im folgenden Abschnitt vorgestellten Softwarewerkzeug ähnelt. Das in [31] vorgeschlagene Verfahren verwendet so genannte n-Permutationen zur Bestimmung eines Ähnlichkeitsmaßes.
Alle genannten Klassifikationsmethoden geben eine Matrix mit den paarweisen Ähnlichkeiten der zu untersuchenden Programme aus und verwenden Clustering-Verfahren für die eigentliche Klassifikation in Familien.

2.6.1 VxClass

VxClass [65] ist ein Softwarewerkzeug zur automatisierten statischen Klassifikation von Malware in Familien und ist – wie schon das in Abschnitt 2.5.2 vorgestellte BinDiff – ein Produkt der Firma Zynamics [66]. Die Benutzerschnittstelle ist Web-basiert und erlaubt authentifizierten Benutzern das Hochladen von ausführbaren Dateien zur Klassifikation. Hochgeladene Dateien werden in Stapelverarbeitung in der Reihenfolge, in der sie hinzugefügt wurden, bearbeitet. Da die automatische Analyse einige Zeit in Anspruch nehmen kann, erlaubt die Benutzeroberfläche außerdem die komfortable Verwaltung von aktiven

Dateien.
Die Ergebnisse der Klassifikation können wie in Abbildung 2.4 graphisch dargestellt und ausgewertet werden. Dafür stehen verschiedene Baumdarstellungen und Graphen-Layouts zur Verfügung. Die Anzeige der berechneten Ähnlichkeitswerte ist ebenso möglich, wie eine aus der Cluseranalyse bekannte Dendrogramm-ähnliche Darstellung. Eine Listenansicht erlaubt zudem die Anzeige und Filterung der bearbeiteten Schadprogramme nach verschiedenen Kriterien wie zum Beispiel Dateinamen, MD5-Hash, Analyse-Dauer oder Dateikommentar.
Für den Fall, dass Malware an anderer Stelle – zum Beispiel durch den Einsatz von Honeypots[43] – gesammelt oder gespeichert wird, bietet VxClass eine RPC-Schnittstelle (*Remote Procedure Call*, RPC) über das XML-RPC-Protokoll an [59], mit der Dateien automatisiert hochgeladen und der Klassifikationsprozess gestartet werden kann.

Da die einzelnen Komponenten einen hohen Integrationsgrad aufweisen, wird VxClass in der Regel als so genannte *Appliance* vertrieben, also im Paket aus Software und der dazugehörigen Hardware. Auf Anfrage ist jedoch auch der alleinige Bezug des Softwarewerkzeuges möglich.

Architektur

Die Software-Architektur von VxClass ist modular aufgebaut und besteht aus vier logischen Schichten (siehe Abbildung 2.5). Auf der obersten Ebene befinden sich die Web-basierte Benutzerschnittstelle und die RPC-Schnittstelle. Diese beiden Komponenten können Schadprogramme entgegennehmen (entweder manuell über den Webbrowser des Benutzers, oder automatisiert über das XML-RPC-Protokoll) und speichern diese zusammen mit einigen Metainformationen (unter anderem sind dies ein Zeitstempel und der Dateiname, weitere Felder werden von anderen Komponenten geschrieben) in einer Datenbank. Als Datenbanksystem kommt die relationale Datenbank MySQL [46] zum Einsatz, der MySQL-Dienstprozess stellt logisch die zweite Schicht dar.
Wird eine Malware in die Datenbank eingefügt, wird mittels Triggers[44] die in der Programmiersprache Python geschriebene Steuerkomponente von VxClass aktiviert (dritte logische Schicht). Diese Komponente verwaltet die Stapelverarbeitung und steuert alle anderen Komponenten.

[43] Ein *Honeypot* [55] ist ein Programm, das Netzwerkdienste eines Rechensystems simuliert, mit dem Ziel Schadprogramme oder Informationen über Angriffe zu sammeln.
[44] Ein *Trigger* ist ein benanntes Datenbankobjekt, das mit einer Tabelle verbunden ist und aktiviert wird, wenn für diese Tabelle ein bestimmtes Ereignis eintritt.

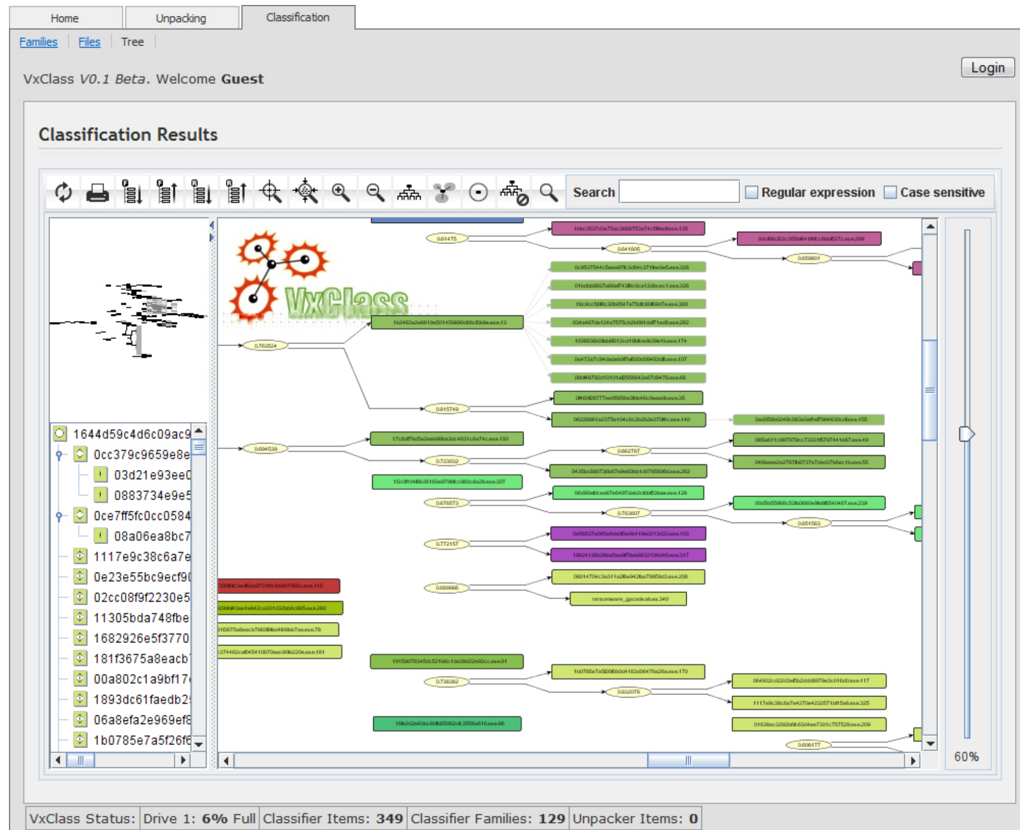

Abbildung 2.4: Von VxClass generierte Baumdarstellung der Ähnlichkeiten von Malware

In der untersten logischen Schicht findet die eigentliche Klassifikation statt, sie besteht aus folgenden Komponenten, die jeweils in einer eigenen virtuellen Maschine auf Basis der Virtualisierungsoftware VMware [61] ausgeführt werden:

Entpacker Bringt eine möglicherweise getarnte Malware in eine Form, die von einem Disassembler verarbeitet werden kann.

Disassembler/Analyse Übersetzt den Objektcode der Malware in Assemblersprache und beginnt die eigentliche Analyse.

Klassifizierer Berechnet – basierend auf den Ergebnissen der Analyse – eine Ähnlichkeitsmatrix aller aktuell in VxClass vorhandenen Schadprogramme und führt eine Clusteranalyse durch.

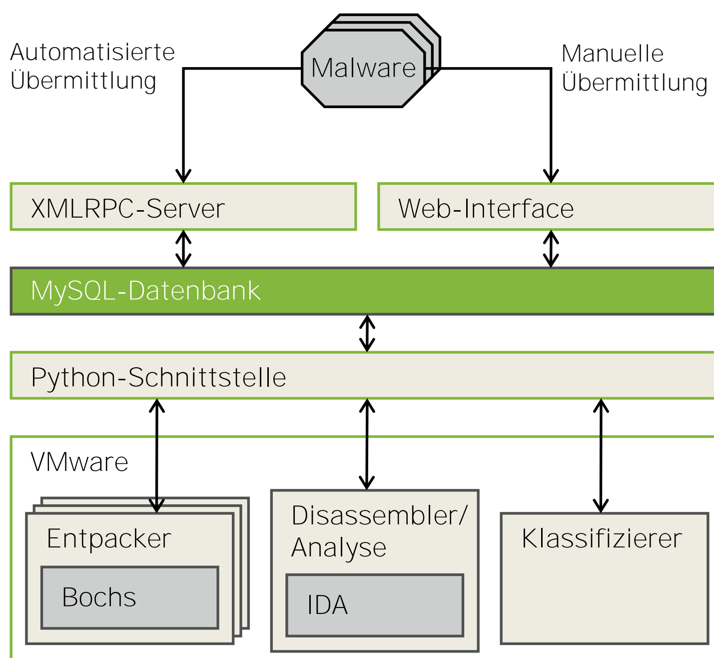

Abbildung 2.5: Architektur von VxClass

Ablauf der Klassifikation

Ein Schadprogramm durchläuft nach der Speicherung der ausführbaren Datei in der Datenbank folgende Stationen [15]:

1. (Entpacken) Die Steuerkomponente kopiert die ausführbare Datei der Malware aus der Datenbank in die virtuelle Maschine des Entpackers. Dort wird sie mithilfe des PC-Emulators Bochs [37] in einer definierten Umgebung[45] zur Ausführung gebracht (Sandboxing, siehe Seite 2.3). Durch die Ausführung wird dafür gesorgt, dass die Dekodierroutine einer möglicherweise getarnten Malware die Nutzlast dekodiert. Nach einer festgelegten Anzahl von Instruktionen (der Standardwert dieses einstellbaren Parameters liegt bei etwa 50 Mio.), wird die Ausführung gestoppt und ein Speicherabbild des Emulators in der Datenbank gespeichert.
   Der Vorgang des Entpackens bedient sich der gleichen Methode, die auch bei der dynamischen Klassifizierung angewendet wird. Der Unterschied besteht jedoch darin, dass bei der Ausführung der Malware keinerlei Verhaltensanalyse stattfindet.

[45]Innerhalb des PC-Emulators wird das Microsoft-Windows-Betriebssytem ausgeführt.

Lediglich modifizierte Speicherseiten werden registriert und in ein – dann statisch untersuchtes – Speicherabbild übernommen.

2. (Disassembler) Nach dem Entpacken wird das in der Datenbank abgelegte Speicherabbild der Malware in die virtuelle Maschine mit der Disassembler- und Analyse-Komponente kopiert. In dieser Komponente wird der Disassembler IDA Pro verwendet, um eine interne Repräsentation der Malware zu erstellen, die als Eingabe für die Analyse-Komponente verwendet wird.

3. (Analyse) Die Analyse-Komponente führt auf Basis der exportierten Malware eine stark verkürzte Version des in BinDiff verwendeten Algorithmus zum strukturellen Vergleichen von Objektcode aus. Die Malware wird auf diese Weise mit allen momentan in VxClass vorhandenen Schadprogrammen verglichen. Das Ergebnis dieses Vergleichs ist ein Ähnlichkeitsvektor, der den Grad der Ähnlichkeit der untersuchten Malware zu allen anderen verglichenen Schadprogrammen angibt und in der Datenbank gespeichert wird.

4. (Klassifizierer) Die Menge der Ähnlichkeitsvektoren werden von der Klassifizierer-Komponente in eine Ähnlichkeitsmatrix überführt, und es wird eine Clusteranalyse mit einem proprietären Verfahren durchgeführt. Das Ergebnis bestimmt schließlich die Familienzuhörigkeit der Malware.

Die oben beschriebene Klassifikation mit VxClass zerlegt eine gegebene Menge von Schadprogrammen disjunkt in Familien. Ein Schwellwert erlaubt zudem eine Filterung der graphischen Darstellung. So kann zum Beispiel eine Auswahl von Schadprogrammen angezeigt werden, deren berechnete und auf 100 Prozent normierte Ähnlichkeitswerte einen bestimmten Prozentsatz aufweisen.

3
Automatisierte Signaturgenerierung

Schaut man sich die vom ClamAV-Projekt bereitgestellten Signaturdatenbanken genauer an, fällt auf, dass sie viele Signaturen für Schadprogramme aus derselben Familie enthalten. So enthält die aktuelle Hauptdatenbank[1] etwa 55 000 Einträge[2], jedoch entfallen allein auf Varianten des trojanischen Pferdes Trojan.MyBot über 4 000 Einträge. Da die Hersteller kommerzieller Antivirenprogramme mit einer vergleichbaren Zahl erkannter Bedrohungen werben, ist davon auszugehen, dass sich die Situation in den dort verwendeten Signaturdatenbanken ähnlich darstellt.

Aufbauend auf den Erfahrungen bei der Analyse von Malware mit den im vorangegangenen Kapitel vorgestellten Softwarewerkzeugen, wird daher in diesem Kapitel ein automatisiertes Verfahren vorgestellt, das aus einer gegebenen Menge von Schadprogrammen einer Familie eine Signatur für das Antivirenprogramm ClamAV erzeugt, die die Erkennung des gesammten Malware-Stamms ermöglicht.

Es folgt zunächst ein Überblick über das Signaturgenerierungsverfahren, dessen prototypische Implementierung in Kapitel 4 beschrieben wird. Im darauffolgenden Abschnitt werden einige Definitionen eingeführt, welche die anschließende ausführliche formale Beschreibung des skizzierten Verfahrens ermöglichen. Das beschriebene Verfahren bedient sich eines Algorithmus für das k-LCS-Problem (siehe Abschnitt 3.2.1). Dies motiviert die nähere Beschäftigung mit längsten gemeinsamen Teilsequenzen in Abschnitt 3.3. Es wird das Verfahren der dynamischen Programmierung beschrieben und ein Überblick über weitere Verfahren gegeben.

In Abschnitt 3.4 wird ein Algorithmus für das k-LCS-Problem auf Permutationen vorgestellt, dessen Korrektheit bewiesen wird. Es folgt eine grobe Abschätzung der Laufzeiteigenschaften. Aufbauend auf dem vorgstellten Algorithmus wird in Abschnitt 3.5 eine Heuristik für den allgemeinen Fall des k-LCS-Problems vorgestellt.

Abschnitt 3.6 beschreibt verschiedene einfache Strategien zur Kürzung von gefundenen

[1] Version 46, Stand: April 2008. Diese Datenbank wird mit jeder neuen Version von ClamAV mitgeliefert. Ergänzt werden die in ihr enthaltenen Signaturen durch Tagesdatenbanken.

[2] Diese Zahl erfasst nur Signaturen im erweiterten Signaturformat.

gemeinsamen Teilsequenzen.
Zum Schluss dieses Kapitels wird ein Algorithmus vorgestellt, der aus einer gemeinsamen Teilsequenz einer gegebenen Menge von Sequenzen einen regulären Ausdruck konstruiert, der alle Sequenzen der Menge akzeptiert. Das Verfahren in Abschnitt 3.2.2 benutzt diesen Algorithmus, um die eigentliche Signatur für den jeweiligen Malware-Stamm zu erzeugen.

3.1 Überblick

Eine mögliche Verbesserung (in Bezug auf die Reduzierung der Anzahl der Signaturen bei gleicher Erkennungsleistung) der Signaturdatenbank von ClamAV wird in [51] vorgestellt: Zunächst werden die Signaturen nach dem Namen der jeweiligen Malware geordnet. Da Varianten eines Schadprogramms mit eigener Signatur in ClamAV einfach durchnummeriert werden, erhält man auf diese Weise eine (grobe) Einordnung in Familien – Varianten stehen in aufsteigender Reihenfolge ihrer Nummer hintereinander. Nachdem eine Auswahl für eine Familie, für die eine Signatur erstellt werden soll, getroffen wurde, und Exemplare von Malware aus dieser Familie gesammelt worden sind, werden die ausführbaren Dateien mittels IDA Pro disassembliert, und es wird mittels BinDiff nach Gemeinsamkeiten gesucht. Gefundene gemeinsame Stellen werden anschließend herangezogen, um manuell eine Teilsignatur für den entsprechenden Objektcode zu erstellen. Teilsignaturen werden anschließend – mit Wildcards (siehe Seite 2.4.1) voneinander getrennt – zur fertigen ClamAV-Signatur zusammengefügt.
Diese Vorgehensweise hat jedoch den Nachteil, dass weiterhin eine zum großen Teil manuelle Analyse von Malware erforderlich ist. Ein strukturiertes Vorgehen erfordert zudem viel Erfahrung und Intuition, um gute Ergebnisse zu liefern.

3.1.1 Ziele

Nach den vorangegangenen Ausführungen sollte ein automatisiertes Verfahren zur Generierung von Signaturen also folgende Ziele verfolgen:

- Schadprogramme werden vorab automatisiert in Familien klassifiziert.
- Das Disassemblieren und Finden von ähnlichen Fragmenten von Objektcode geschieht ohne jeden Benutzereingriff.

- Die Erzeugung der Signaturen erfolgt auf Basis der Assemblerdarstellung der identifizierten ähnlichen Stellen.
- Es wird eine gültige und möglichst kurze Signatur erzeugt. Wildcards und andere verfügbare Sprachmittel der Ziel-Signatursprache werden automatisch hinzugefügt.

Außerdem ist es wünschenswert, dass keine oder nur wenige falsche Positive und falsche Negative bei der Verwendung der Signatur auftreten und die Laufzeit des gesamten Verfahrens möglichst kurz ist.

3.1.2 Idee für ein Signaturgenerierungsverfahren

Eine Idee für ein Verfahren, das die oben genannten Ziele erfüllt, ist folgende:

Vorverarbeitung: Automatische Klassifizierung einer Menge von Schadprogrammen in Familien und Auswahl einer Familie.
Eingabe: Eine Familie von Schadprogrammen.
Ausgabe: Eine ClamAV-Signatur für die ausgewählte Malware-Familie.

1. Vorab wird zur Klassifizierung einer Menge von Schadprogrammen das Softwarewerkzeug VxClass benutzt, denkbar wäre jedoch auch der Einsatz eines der anderen in Abschnitt 2.6 genannten Verfahren. Dies liefert eine Zerlegung der Ausgangsmenge in Familien, aus der eine Familie für die Generierung einer Signatur ausgewählt wird. Die Auswahl kann dabei manuell oder automatisch nach bestimmten festgelegten Kriterien erfolgen.
2. Die einzelnen Schadprogramme werden in einer beliebigen aber festen Reihenfolge angeordnet und das Programm BinDiff sequentiell auf je zwei Paaren ausgeführt, um strukturell identische Funktionen und Basic-Blocks zu finden.
3. In den gefundenen Funktionen wird nach *Kandidaten-Funktionen* gesucht. Kandidaten-Funktionen sind Funktionen, die in allen Schadprogrammen vorkommen. Aus den Basic-Blocks der Kandidaten-Funktionen werden analog *Kandidaten-Basic-Blocks* gebildet.
4. Für jede ausführbare Datei einer Malware wird die Reihenfolge der Adressen, in der die Kandidaten-Basic-Blocks vorkommen, festgehalten.

5. Aus den einzelnen Adressfolgen werden Adressen bestimmt, die überall in derselben aufsteigenden Reihenfolge vorkommen. Diese werden mit Trennsymbolen hintereinander in eine Liste geschrieben.
6. Für jeden Eintrag der Liste, der nicht das Trennsymbol ist, werden aus den ausführbaren Dateien die Instruktionen der Kandidaten-Basic-Blocks extrahiert und wiederum mit Trennzeichen in Listen für das jeweilige Schadprogramm geschrieben.
7. Für jeden Kandidaten-Basic-Block werden die gemeinsamen Bytes der Instruktionen über alle betrachteten Malware-Exemplare bestimmt und hintereinander geschrieben in einer Zeichenfolge festgehalten. Nicht zusammenhängende Bytes werden mit einem ClamAV-Wildcard-Symbol ⌜*⌟ voneinander getrennt, zwischen je zwei Kandidaten-Basic-Blocks wird ebenfalls ein Wildcard eingefügt.
8. Aus der entstandenen Zeichenfolge werden doppelte Wildcard-Symbole entfernt, und es wird ein Name für die Signatur der Malware-Familie erzeugt. Der Name – zusammen mit der in hexadezimale Darstellung umgewandelten Zeichenfolge – ergibt die ClamAV-Signatur.

3.2 Signaturgenerierung mittels BinDiff und k-LCS

Für eine formale Beschreibung des oben skizzierten Verfahrens sind zunächst noch einige Definitionen erforderlich.

3.2.1 Definitionen

Definition 3.2.1 (BinDiff-Funktion, CGISO, CFGISO)**.** Eine Funktion, die zwei gegebenen Graphen-Strukturen[3] $G_{(A)} = (F_{(A)}, E_{(A)})$ und $G_{(B)} = (F_{(B)}, E_{(B)})$ zweier ausführbarer Dateien A und B mittels Selektoren und Eigenschaften einen Isomorphismus $\varphi : F_{(A)} \to F_{(B)}$ zuordnet, wird *BinDiff-Funktion* $\mathrm{bindiff} : F_{(A)} \times F_{(B)} \to \mathrm{Abb}(F_{(A)} \cup F_{(B)})$ genannt, wobei $\mathrm{Abb}(F_{(A)} \cup F_{(B)})$ die Menge aller möglichen Abbildungen zwischen $F_{(A)}$ und $F_{(B)}$ bezeichnet.
Ein durch die BinDiff-Funktion zugeordneter Graph-Isomorphismus $\varphi : F_{(A)} \to F_{(B)}$ lässt sich durch zwei Mengen $CGISO_{A,B} := \{(f,e) | f \in F_{(A)}, e \in F_{(B)}\}$ und $CFGISO_{A,B} :=$

[3]siehe Seite 33

$\{(b, c) | \forall 0 < i \leq |F_{(A)}| : b \in B_{(A),i}, c \in B_{(B),i}\}$ von Adresspaaren beschreiben. Diese Mengen sind somit eine alternative Darstellung für die mithilfe der BinDiff-Funktion konstruierten Graph-Isomorphismen zwischen den Aufruf- und Kontrollflussgraphen der ausführbaren Dateien A und B.

Korollar 3.2.2 (ohne Beweis). *Das Softwarewerkzeug BinDiff realisiert genau die in Definition 3.2.1 angegebene Funktion.* □

Satz 3.2.3 (ohne Beweis). *Die BinDiff-Funktion ist reflexiv, symmetrisch und transitiv und realisiert damit eine Äquivalenzrelation auf einer Menge von Schadprogrammen.* □

Definition 3.2.4 (Kandidatenfunktionen und -Basic-Blocks). Sei $M = \{M_1, \ldots, M_n\}$ eine gegebene Menge von Schadprogrammen einer Familie, f eine Funktion aus $F_{(M_i)}$ (mit $0 < i < n$) und bezeichne $\varphi_{M_i,M_{i+1}}(f)$ eine Funktion, die f ihrer Adresse in M_{i+1} zuordnet, falls ein Paar $(f, g) \in CGISO_{M_i,M_{i+1}}$ existiert. $\varphi_{M_i,M_{i+1}}$ bildet den Isomorphismus zwischen den Aufrufgraphen von M_i und M_{i+1} nach, falls ein solcher existiert. Die Funktion f ist genau dann eine *Kandidatenfunktion*, wenn für alle $0 < i < n$ ein Paar $(f, \varphi_{M_i,M_{i+1}}(f)) \in CGISO_{M_i,M_{i+1}}$ existiert.
Ein Basic-Block $b \in B_{(A),i}$ ist genau dann ein *Kandidaten-Basic-Block*, wenn für alle $0 < i < n$ ein Paar $(b, \chi_{M_i,M_{i+1}}(b)) \in CFGISO_{M_i,M_{i+1}}$ existiert. Die Funktion $\chi_{M_i,M_{i+1}}(b)$ ordnet dabei dem Basic-Block b seine Adresse in M_{i+1} zu, falls ein Paar $(b, c) \in CFGISO_{M_i,M_{i+1}}$ existiert.
Anschaulich betrachtet, sind Kandidatenfunktionen Funktionen, die strukturell in allen Elementen einer gegebenen Malware-Familie vorkommen. Kandidaten-Basic-Blocks sind analog dazu Basic-Blocks, die strukturell in allen gegebenen Familienelementen vorkommen.

Definition 3.2.5 (String, Sequenz, Teilsequenz). Eine *Sequenz* x, auch *String* oder *Verkettung* genannt, über einem endlichen Alphabet Σ ist eine geordnete Menge von Symbolen aus Σ, wobei Wiederholungen erlaubt sind, das heißt $x := \langle x_i | \forall 1 \leq i \leq n : x_i \in \Sigma \rangle$. Statt $\langle x_1, \ldots, x_n \rangle$ wird kürzer auch $x_1 \ldots x_n$ geschrieben und x mit einem Wort aus Σ^* identifiziert. Die *Länge* einer Sequenz wird mit $|x|$ bezeichnet.
Eine Sequenz $y = y_1 \ldots y_m$ wird genau dann *Teil-* oder *Subsequenz* einer Sequenz x genannt, wenn eine Einbettung von Indizes $I = (i_1, \ldots, i_m)$ mit $1 \leq i_1 < \ldots < i_m \leq |x|$ aus y in x existiert, so dass für alle Indizes k mit $1 \leq k \leq m$ gilt: $x_{i_k} = y_k$. Man erhält die Teilsequenz y aus x durch Löschen von $n - m$ Symbolen aus x.
Die Menge aller Teilsequenzen von x wird mit $\text{seq}(x) := \{y | y \text{ ist Teilsequenz von } x\}$ bezeichnet.

Für eine Teilsequenz kann es mehrere mögliche Einbettungen geben. Beispielsweise ist aar eine Teilsequenz von aardvark, mit den vier möglichen Einbettungen $(1,2,3)$, $(1,2,7)$, $(1,6,7)$ und $(2,6,7)$ (Abbildung 3.1).

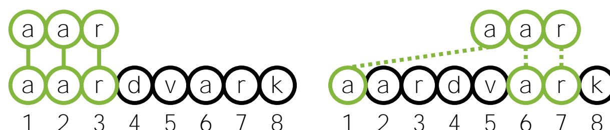

Abbildung 3.1: Zwei mögliche Einbettungen einer Teilsequenz

Definition 3.2.6 (k-Longest Common Subsequence Problem)**.** Eine Sequenz x heißt *gemeinsame Teilsequenz* einer gegebenen Menge von Sequenzen $\mathcal{S} = \{S_{(1)}, \dots, S_{(k)}\}$, wenn sie eine Teilsequenz von allen Sequenzen aus $\mathcal{S}$ ist, also $\forall S_{(i)} \in \mathcal{S} : x \in \text{seq}(S_{(i)})$ gilt. Eine *längste gemeinsame Teilsequenz* $\text{LCS}(\mathcal{S})$ von $\mathcal{S}$ ist eine Teilsequenz maximaler Länge: $\text{LCS}(\mathcal{S}) := \max_{S_{(i)} \in \mathcal{S}} \{|x| | x \in \text{seq}(S_{(i)})\}$.
Das Finden einer längsten gemeinsamen Teilsequenz einer Menge von k Sequenzen wird als *k-Longest Common Subsequence Problem* (kurz k-LCS) bezeichnet. Für den Fall $k = 2$ wird auch von dem *Longest Common Subsequence Problem* (LCS) gesprochen.

3.2.2 Formale Beschreibung

Es sind nun alle nötigen Definitionen für eine formalisierte Beschreibung der Idee aus Abschnitt 3.1.2 vorhanden.

Vorverarbeitung: Automatische Klassifizierung einer Menge von Schadprogrammen in Familien und Auswahl einer Familie.
Eingabe: Eine Familie $M = \{M_1, \dots M_n\}$ von Schadprogrammen.
Ausgabe: Eine ClamAV-Signatur S für die Schadprogramme aus M.

1. Vorab wird zur Klassifizierung einer Menge von Schadprogrammen das Softwarewerkzeug VxClass benutzt, denkbar wäre jedoch auch der Einsatz eines der anderen in Abschnitt 2.6 genannten Verfahren. Das Ergebnis der Vorverarbeitung ist eine Zerlegung der Ausgangsmenge in Familien, aus der eine Familie M für die Generierung einer Signatur ausgewählt wird.

2. Die einzelnen Schadprogramme aus M werden in einer beliebigen, aber festen Reihenfolge angeordnet und die BinDiff-Funktion auf alle Paare (M_i, M_{i+1}) mit $0 < i < n$ angewendet. Dies liefert jeweils $d = n - 1$ Mengen $CGISO_{M_i, M_{i+1}}$ und

$CFGISO_{M_i,M_{i+1}}$ von Adresspaaren der zwischen M_i und M_{i+1} gefundenen strukturell identischen Funktionen und Basic-Blocks.

3. Es wird eine Menge von Kandidatenfunktionen berechnet (siehe Abbildung 3.2). Dabei wird eine Matrix F_{cand} mit d Spalten angelegt. Der i-te Spaltenvektor enthält die Adressen, die den einzelnen Kandidatenfunktion in M_i zugeordnet wurden.

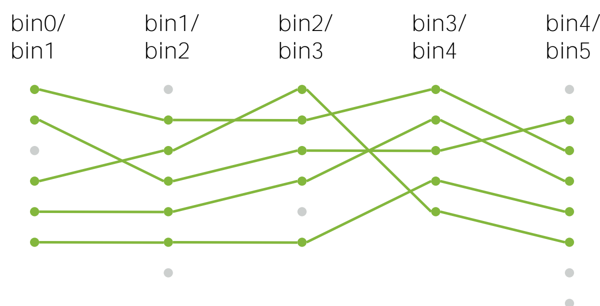

Abbildung 3.2: Symbolische Darstellung der Kandidatenfunktionen von sechs verglichenen Schadprogrammen vor dem Einfügen in F_{cand}.

4. Aus den Basic-Blocks der Kandidatenfunktionen werden nun – analog zu 3. – *Kandidaten-Basic-Blocks* berechnet, die in allen Elementen der Malware-Familie vorkommen. Auch hier wird eine Matrix $B_{cand} = (b_{i,j})$, die die Adressen der einzelnen Kandidaten-Basic-Blocks festhält, angelegt. Bezeichnet c die Anzahl der gefundenen Kandidaten-Basic-Blocks, ergibt sich also eine $c \times d$-Matrix.

5. Für jedes Schadprogramm M_i wird nun das „Wort" über dem Alphabet der Kandidaten-Basic-Blocks generiert. Anders ausgedrückt: Für jedes M_i wird eine Permutation π_i bestimmt, die die Reihenfolge der Adressen der Kandidaten-Basic-Blocks in M_i angibt. Dazu werden den Adressen der einzelnen Spaltenvektoren ihre Zeilennummern zugeordnet, und die so entstandenen Paare werden in einer $c \times d$-Matrix $W = (w_{i,j})$ gespeichert. Ein Eintrag $w_{i,j}$ besteht aus dem Paar $(j, b_{i,j})$. Die Paare jeder Spalte werden anschließend nach aufsteigenden Adressen sortiert. Die Nummern aus den Paaren der i-ten Spalte bilden dann – hintereinander geschrieben – die Elemente der Permutation π_i. π_i ist ein Element der symmetrischen Gruppe der c-stelligen Permutationen. Nach der Sortierung lässt sich ein Eintrag $w_{i,j}$ durch das Paar $(\pi_i(j), b_{i,j})$ beschreiben.

6. Da ClamAV-Signaturen ausschließlich die Suche nach Byte-Sequenzen in einer

festen Reihenfolge ermöglichen[4], wird nach einer Teilsequenz von Kandidaten-Basic-Blocks gesucht, die in aufsteigender Reihenfolge ihrer Startadressen in allen Schadprogrammen vorkommen.
Fasst man die Spaltenvektoren von W als Sequenzen auf, entsteht eine solche Teilsequenz durch Anwendung eines Algorithmus für das k-LCS-Problem (siehe Definition 3.2.6) auf den Nummern der einzelnen Paare (diese stellen wie in Schritt 5 die Permutation π_i dar). Ein Polynomialzeitalgorithmus für das k-LCS-Problem auf Permutationen und eine Heuristik für das allgemeine k-LCS-Problem werden in den Abschnitten 3.4 und 3.5 vorgestellt.
Das Ergebnis der Ausführung dieses Schritts ist eine Sequenz $K = k_1 \ldots k_\ell$, die eine längste gemeinsame Teilsequenz aller π_i ist.

7. Da die Länge der im vorherigen Schritt berechneten gemeinsamen Teilsequenz K in der Praxis sehr groß sein kann[5], wird nun die Länge von K durch einen Kürzungsalgorithmus (siehe Abschnitt 3.6) nach oben begrenzt. Die gekürzte Sequenz wird mit K' bezeichnet, die resultierende Länge mit ℓ'.

8. Es werden aus allen Spalten der Matrix W die Paare gestrichen, deren zugeordnete Nummer nicht in der Sequenz K' vorkommt. Das heißt, aus Spalte i werden alle Paare $w_{i,j} = (\pi_i(j), b_{i,j})$ gestrichen, für die $\pi_i(j) \notin K'$ ist. Die resultierende $\ell' \times d$-Matrix wird mit $W' = (w'_{i,m})$ bezeichnet, die einzelnen Elemente von W' bestehen aus den Paaren $w'_{i,m} = (\pi_i(\text{prev}_j(m)), b_{i,\text{prev}_j(m)})$, wobei $\text{prev}_j(m)$ den Zeilenindex des Paares in Spalte j und Zeile m vor der Anwendung des Kürzungsalgorithmus bezeichnet.

9. Die Matrix W' wird spalten- und zeilenweise durchlaufen und eine $\ell' \times d$-Matrix $I_W = (i_{W_{i,m}})$ mit den Byte-Sequenzen der Instruktionen der Basic-Blocks aus den Elementen von W' angelegt.

10. Die einzelnen Zeilen von I_W stellen eine Menge von Byte-Sequenzen dar, auf welche die in Abschnitt 3.5 beschriebene Heuristik für das k-LCS-Problem angewendet wird. Die gemeinsamen Teilsequenzen, die sich aus den einzelnen Zeilen ergeben, werden in einem Zeilenvektor T der Länge ℓ' gespeichert. $T = (t_m)$ ergibt sich also aus der Faltung der Matrix I_W durch Anwendung der Heuristik.

[4] Das Zulassen von n Byte-Sequenzen in beliebiger Reihenfolge erfordert die Konstruktion einer Signatur mit $n!$ disjunkten Vereinigungen $\ulcorner|\lrcorner$ und ist somit nicht praktikabel.

[5] Schadprogramme, die mehrere Tausend strukturell identische Basic-Blocks gemeinsam haben, sind keine Seltenheit.

11. Der Vektor T und die Matrix I_W werden gleichzeitig zeilenweise durchlaufen, dabei wird für jede gemeinsame Teilsequenz t_m ein regulärer Ausdruck r_m erzeugt, der die Byte-Sequenzen der Zeilen von I_W akzeptiert. Ein Verfahren, das aus einer gegebenen Menge von Sequenzen und einer gemeinsamen Teilsequenz dieser Menge einen solchen regulären Ausdruck berechnet, wird in Abschnitt 3.7 vorgestellt.

12. Die einzelnen regulären Ausdrücke r_m werden hintereinander geschrieben und mit einem Wildcard-Symbol $\ulcorner\ast\lrcorner$ voneinander getrennt. Dies ergibt einen neuen regulären Ausdruck $R = r_1\ulcorner\ast\lrcorner\ldots\ulcorner\ast\lrcorner r_m$.

13. Nach der Erzeugung eines geeigneten Namens ($\ulcorner\texttt{<name>}\lrcorner$) für die Signatur der Malware-Familie ergibt sich eine ClamAV-Signatur

$$S = \ulcorner\texttt{<name>:0:*:}\lrcorner\,\mathrm{hex}_R(r_1)\ulcorner\ast\lrcorner\ldots\mathrm{hex}_R(r_m).$$

$\mathrm{hex}_R(r)$ bezeichnet dabei die Darstellung des regulären Ausdrucks r nach der Umwandlung der enthaltenen Byte-Sequenzen in hexadezimale Darstellung. Der Teilausdruck $\ulcorner\texttt{:0:*:}\lrcorner$ gibt an, dass es sich bei der generierten ClamAV-Signatur um eine Signatur im erweiterten Signaturformat handelt, die auf alle Dateien und alle Dateipositionen anzuwenden ist.

Die oben gegebene Beschreibung weicht etwas von der in Abschnitt 3.1.2 skizzierten Idee ab, beispielsweise werden nach Konstruktion keine doppelten Wildcard-Symbole in der finalen Signatur erzeugt.
Die Länge der Beschreibung lässt außerdem vermuten, dass es sich bei dem vorgestellten Verfahren um eine sehr aufwändige Operation handelt. Tatsächlich aber handelt es sich bei den meisten Schritten nur um einfache Durchläufe der verwendeten Matrizen. Lediglich die Verwendung eines Algorithmus für das k-LCS-Problem ist algorithmisch aufwändiger – wie in Abschnitt 3.4.2 gezeigt wird, ist dies allerdings eher bezogen auf den verwendeten Speicherplatz, als auf die benötigte Rechenzeit der Fall.

3.3 Längste gemeinsame Teilsequenzen

Das in Abschnitt 3.2.1 definierte k-Longest-Common-Subsequence-Problem ist ein klassisches und viel untersuchtes Problem, das in vielen Teilbereichen der Informatik Anwendung findet, von der Datenkompression bis hin zu Multiple-Sequence-Aligment-Problemen in der Molekularbiologie. Bereits 1978 konnte Maier [40] nachweisen, dass

das allgemeine k-LCS-Problem NP-hart ist. Spätere Ergebnisse in [30] zeigen außerdem, dass auch die Approximation des k-LCS-Problems algorithmisch aufwändig ist.
Das eingeschränkte Problem des Findens einer gemeinsamen Teilsequenz von zwei Teilsequenzen (2-LCS) ist ebenfalls gut untersucht und ist in Polynomialzeit berechenbar.

3.3.1 Dynamische Programmierung

Der klassische Ansatz zur Lösung des LCS-Problems mit der Methode der dynamischen Programmierung wird in [60] beschrieben und benötigt Rechenzeit $O(mn)$ und ebensoviel Speicherplatz, wobei mit m und n die Länge der beiden Sequenzen bezeichnet wird. Abbildung 3.3 zeigt den Pseudocode dieses Algorithmus.

```
DYN-LCS(s = s_1 ... s_m, t = t_1 ... t_n)
 1  opt ← dim(m + 1, n + 1)                                  ▷ Tabelle anlegen
    ▷ LCS-Länge aller Teilprobleme mit dynamischer Programmierung berechnen
 2  for i ← m downto 1
 3       do for j ← n downto 1
 4              do if s_i = t_j
 5                    then opt[i][j] ← opt[i + 1][j + 1] + 1
 6                    else  opt[i][j] ← max(opt[i + 1][j], opt[i][j + 1])
    ▷ LCS selbst aus Tabelle rekonstruieren
 7  i ← 1, j ← 1, result = NIL
 8  while i ≤ m and j ≤ n
 9       do if s_i = t_j
10            then append(result, s[i])
11                 i ← i + 1
12                 j ← j + 1
13          elseif opt[i + 1][j] ≥ opt[i][j + 1]
14            then i ← i + 1
15          else j ← j + 1
16  return result
```

Abbildung 3.3: Berechnung des LCS mit dynamischer Programmierung

Da das k-LCS-Problem NP-hart ist, führt eine direkte Verallgemeinerung dieser Methode

auf k Sequenzen der Länge n erwartungsgemäß zu einer exponentiellen Laufzeit $O(n^k)$ und Platzbedarf $O(n^{k-1})$. Für große Eingabeinstanzen ist der Ansatz der dynamischen Programmierung daher nicht geeignet.

3.3.2 Weitere Verfahren

Im allgemeinen Fall mit k Sequenzen kommen in der Praxis häufig heuristische Verfahren zum Einsatz, die eine optimale Lösung nicht garantieren. Ist jedoch die Berechnung einer optimalen Lösung erforderlich, muss meist eine exponentielle Worst-Case-Laufzeit in Kauf genommen werden. Dennoch können Algorithmen zur Berechnung einer optimalen Lösung für k-LCS im Average-Case eine gute Laufzeit aufweisen. In [27] wird beispielsweise ein Branch-and-Bound-Algorithmus mit exponentieller Worst-Case-Laufzeit und einer guten Average-Case-Laufzeit von $O(nk|\Sigma|)$ beschrieben, der eine optimale Lösung berechnet, wobei $|\Sigma|$ die Größe des Eingabealphabets bezeichnet.
Ein massiv-paralleler Ansatz wird in [8] verfolgt, benötigt jedoch bei k Sequenzen ebensoviele Prozessoren, um die Laufzeit von $O(|\mathrm{LCS}(s_1, \dots, s_k)|)$ zu erreichen. Hierbei bezeichnen die $s_1, \dots, s_k$ die Eingabesequenzen.

Im Falle von 2-LCS kann der quadratische Platzbedarf der *opt*-Tabelle aus Abbildung 3.3 reduziert werden: Hirschberg [26] beschreibt einen Algorithmus, der mit linearem Platzbedarf $O(m+n)$ bei gleicher Zeitkomplexität auskommt, dafür jedoch deutlich komplizierter ist. Der Algorithmus mit der gegenwärtig[6] besten bekannten Laufzeitschranke wird von Masek und Paterson in [41] beschrieben und benutzt die so genannte „Vier Russen"-Methode, um die Worst-Case-Laufzeit auf $O(mn/\log n)$ zu reduzieren.
Hunt und Szymanski [28] beschreiben einen weiteren Algorithmus für das LCS-Problem mit einer Worst-Case-Laufzeit von $O(n^2 \log n)$ (es wird davon ausgegangen, dass die Sequenzen die gleiche Länge n haben). Diese Worst-Case-Laufzeit ist zwar schlechter, als die Methode der dynamischen Programmierung, jedoch wird für bestimmte Eingabeinstanzen eine signifikant bessere Laufzeit erwartet. Bezeichne $\mathcal{R}$ die Anzahl der geordneten Paare von Positionen, an denen die Sequenzen übereinstimmen. Die Laufzeit des beschriebenen Algorithmus ist dann $O((\mathcal{R}+n)\log n)$. Ein Ergebnis aus [28] ist, dass für Eingaben, bei denen $\mathcal{R}$ nahe an n liegt, die erwartete Laufzeit $O(n \log n)$ beträgt.

[6] Stand: 2008

3.4 Längste gemeinsame Teilsequenzen von Permutationen

Schritt 6 der Beschreibung aus Abschnitt 3.2.2 erfordert die Berechnung einer längsten gemeinsamen Teilsequenz (LCS) von k Sequenzen der Länge n. Wie im vorangegangenen Abschnitt 3.3 gesehen, ist im Allgemeinen eine exponentielle Laufzeit für die Berechnung einer exakten Lösung zu erwarten. Da es sich jedoch bei der Eingabe um Sequenzen aus Elementen von n-stelligen Permutationen handelt, besteht die berechtigte Hoffnung, durch geschickte Ausnutzung der Eigenschaften der Eingabe die Laufzeit wesentlich zu verbessern. Dazu wird die Definition eines Distanzmaßes zwischen zwei Sequenzen benötigt.

Definition 3.4.1 (Hamming-Abstand)**.** Seien $x = x_1 \dots x_n$ und $y = y_1 \dots y_n$ Sequenzen über einem endlichen Alphabet Σ. Der *Hamming-Abstand* [23] zwischen x und y ist definiert als $\Delta_h(x, y) := \sum_{x_i \neq y_i} 1$.
Anschaulich betrachtet, ist der Hamming-Abstand zweier gleichlanger Sequenzen also gleich der Anzahl ihrer unterschiedlichen Stellen.

Zunächst lässt sich folgendes beobachten: Die Methode der dynamischen Programmierung für k-LCS hat eine exponentielle Laufzeit, da sich die Größe der *opt*-Tabelle nicht verringert und alle Einträge berechnet werden müssen, sofern man nicht nur die Länge der LCS in Erfahrung bringen will. Die Idee zur Verringerung der Laufzeit ist daher folgende: Die Problemgröße wird verringert, indem die zwei – bezüglich eines geeignet definierten Distanzmaßes – ähnlichsten Sequenzen durch ihre LCS ersetzt werden. Der oben definierte Hamming-Abstand ist ein geeignetes Distanzmaß. Aus den restlichen Sequenzen werden alle Elemente gestrichen, die nicht in der berechneten LCS vorkommen, da sie nicht Teil der LCS aller Sequenzen sein können. Mit dem reduzierten Problem wird so lange rekursiv fortgefahren, bis nur noch zwei Sequenzen übrig sind. Die Berechnung einer LCS von zwei Sequenzen ist zum Beispiel mit dynamischer Programmierung in Polynomialzeit möglich.

Ein Algorithmus HammingLCS, der die obige Idee umsetzt, wird in Abbildung 3.4 im Pseudocode dargestellt. Zur Bestimmung der zwei ähnlichsten Sequenzen wird paarweise der Hamming-Abstand berechnet. Neben der Ersetzung zweier Sequenzen durch ihre LCS wird als zusätzliche Optimierung noch eine Liste von Indizes geführt, deren zugehörige Sequenzen gelöscht werden können, da sie mehrfach in der Eingabe vorkommen. Für die Streichung von Elementen aus Sequenzen wird die Funktion RetainAlphabet verwendet.

```
RetainAlphabet(S = s_1 ... s_m, Σ)
1  result = NIL
2  for i ← 1 to m
3        do if s_i ∈ Σ
4              then append(result, s_i)
5  return result
```

```
HammingLCS(S_1, ... S_k)
 1  if k = 2                  ▷ Löse Problem der Größe 2 mit dynamischer Programmierung
 2     then return Dyn-LCS(S_1, S_2)
 3  kill ← NIL                              ▷ Liste mit Indizes zu löschender Sequenzen
 4  hmin ← +∞, hcur ← 0                     ▷ Minimaler und aktueller Hamming-Abstand
 5  shd1 ← 0, shd2 ← 0
    ▷ Indizes der Sequenzen mit dem kleinsten Hamming-Abstand suchen
 6  for i ← 1 to k
 7       do for j ← 1 to i
 8              do if i = j
 9                    then continue
10                 hcur ← Δ_h(S_i, S_j)                  ▷ Hamming-Abstand berechnen
11                 if hcur = 0
12                    then append(kill, i)               ▷ Identische Sequenz löschen
13                 elseif hcur < hmin
14                    then hmin ← hcur
15                         shd1 ← i, shd2 ← j
16  if |kill| = k − 1                       ▷ Alle Sequenzen bis auf eine identisch?
17     then return S_1                                      ▷ Diese ist dann LCS
    ▷ Berechne LCS der „ähnlichsten Sequenzen“ mit dynamischer Programmierung
18  minlcs ← Dyn-LCS(S_shd1, S_shd2)
19  sublist ← NIL
20  for i ← 1 to k
21       do if i ∈ kill or i = shd2
22             then continue
23          elseif i = shd1
24             then append(sublist, minlcs)
25          else append(sublist, RetainAlphabet(S_i, minlcs))
26  if |sublist| = 1                          ▷ Nur noch eine Sequenz übrig?
27     then return first(sublist)
28     else  return HammingLCS(sublist)                          ▷ Rekursion
```

Abbildung 3.4: Ein Algorithmus zur Berechnung von gemeinsamen Teilseqenzen

3.4.1 Korrektheit

Satz 3.4.2. *Der Algorithmus* HAMMINGLCS *berechnet bei Eingabe von k Sequenzen S_i mit $0 < i \leq k$, die aus Elementen von n-stelligen Permutationen bestehen, die längste gemeinsame Teilsequenz aller S_i.*

Beweis. Die Eingabesequenzen haben die Eigenschaft, dass sie alle das gleiche endliche Alphabet verwenden. Da die Sequenzen außerdem aus Elementen von n-stelligen Permutationen bestehen, kommt jedes Element in einer Sequenz genau einmal vor. Der Hamming-Abstand zweier Sequenzen S_i und S_j gibt daher die Anzahl der Fixpunkte der Permutation $\pi = \pi_{S_i} \circ \pi_{S_j}$ an, die sich aus der Hintereinanderausführung der Permutationen der beiden Sequenzen ergibt.
Die LCS von S_i und S_j bestimmt ein neues Alphabet Σ_π, das nun auf alle anderen Sequenzen durch Streichung von Elementen, die nicht in Σ_π enthalten sind, aufgeprägt wird. Die Streichung ist korrekt, da die LCS von S_i und S_j in jedem Fall eine Teilsequenz der LCS aller Sequenzen der Eingabe ist.
Der Algorithmus terminiert, da jeder rekursive Aufruf die Problemgröße um mindestens Eins verringert und ein Problem der Größe Zwei mit der Methode der dynamischen Programmierung gelöst wird. Nach $k-1$ Rekursionen bleibt also nur noch die LCS von allen Sequenzen S_i übrig. □

3.4.2 Laufzeiteigenschaften

Es folgt eine grobe Abschätzung der von HAMMINGLCS benötigten Laufzeit. Da mit Ausnahme der Liste *kill* kein zusätzlicher Platz benötigt wird, ist der Speicherplatz durch $O(n^2)$ nach oben beschränkt.
Die Abschätzung der Laufzeit ist durch die Angabe einer rekursiven Funktion $T(k,n)$ möglich, die die Laufzeiten der einzelnen Schritte zusammenfasst:

- Jeder rekursive Aufruf reduziert die Problemgröße um mindestens Eins, nach $k-1$ Aufrufen sind also höchstens noch zwei Sequenzen übrig. Diese Sequenzen haben höchstens die Länge $n-k+1$. Identische Sequenzen werden gestrichen, und die LCS von zwei Sequenzen hat damit höchstens die Länge $n-1$. RETAINALPHABET kürzt auf Grund der Permutationseigenschaften der Eingabe alle anderen Sequenzen auf diese Länge.

- Die paarweise Berechnung des Hamming-Abstands und Bestimmung der Indizes h_1 und h_2 der ähnlichsten Sequenzen benötigt $O(\binom{k}{2}n)$ Operationen.
- Die Berechnung der LCS der beiden ähnlichsten Sequenzen benötigt $O(n^2)$ Operationen.
- Die Länge $\ell = |\mathrm{LCS}(S_{h_1}, S_{h_2})|$ der LCS der beiden ähnlichsten Sequenzen ist durch $n-1$ nach oben beschränkt. Der Aufruf von RetainAlphabet auf allen $k-1$ Sequenzen benötigt dann $O((k-1)n(n-1))$ Operationen.
- Der rekursive Aufruf des Algorithmus auf $k-1$ Sequenzen der Länge $n-k+1$ benötigt $T(k-1, n-k+1)$ Operationen.
- Ein Problem der Größe Zwei benötigt – unter Verwendung der dynamischen Programmierung – $O(n^2)$ Operationen.

Nach Anwendung einiger Abschätzungen und Indextransformationen ergibt sich für die Laufzeitfunktion:

$$
\begin{aligned}
T(2, n-k+1) &= n^2 \\
T(k,n) &= \binom{k}{2}n + n^2 + (k-1)n(n-1) + T(k-1, n-k+1) \\
&\leq k^2 n + n^2 + (k-1)n^2 + T(k-1, n-k+1) \\
&\leq k^2 n + kn^2 + \sum_{i=1}^{k-2}\left[(k-i)^2(n-i) + (k-i)(n-i)^2\right] + T(2, n-k+1) \\
&\leq k^2 n + kn^2 + \sum_{i=1}^{k-2} n(k-i)^2 + \sum_{i=1}^{k-2}(k-i)n^2 + n^2 \\
&= k^2 n + kn^2 + n\sum_{i=2}^{k-1} i^2 + n^2\sum_{i=2}^{k-1} i + n^2 \\
&= k^2 n + kn^2 + \frac{1}{2}n(k-1)^2((k-1)^2+1) + \frac{1}{2}n^2(k-1)k \\
&= O(k^4 n + k^2 n^2)
\end{aligned}
$$

Diese Laufzeit ist – wie bereits vermutet – signifikant besser als die exponentielle Laufzeit für die Berechnung einer LCS für k Sequenzen im allgemeinen Fall.

3.5 Eine Heuristik für das k-LCS-Problem

Wie gezeigt, berechnet der Algorithmus aus Abbildung 3.4 bei Eingabe von k Sequenzen aus Elementen von n-stelligen Permutationen in Zeit $O(n^4 n + n^2)$ eine optimale Lösung für das k-LCS-Problem auf Permutationen. Es liegt nahe, diesen Algorithmus auch in Schritt 10 aus Abschnitt 3.2.2 zur Berechnung von gemeinsamen Teilsequenzen zu verwenden. Die Byte-Sequenzen der Eingabe sind dort jedoch nicht aus Elementen von Permutationen entstanden und haben demnach auch keine besondere Struktur. Jede Sequenz verwendet außerdem eine potentiell andere Teilmenge des Alphabets der durch ein Byte darstellbaren Zeichen. Erschwerend kommt hinzu, dass die Sequenzen der Eingabe eine unterschiedliche Länge aufweisen können, was die Definition eines anderen Distanzmaßes erfordert.
Eine optimale Lösung für das allgemeine k-LCS-Problem ist daher nicht zu erwarten. Der Algorithmus HAMMINGLCS kann jedoch dazu verwendet werden, um bei ansonsten gleicher Zeitkomplexität eine gemeinsame – nicht notwendigerweise längste – Teilsequenz von k Sequenzen zu berechnen. Dazu wird die Definition 3.4.1 verallgemeinert:

Definition 3.5.1 (erweiterter Hamming-Abstand)**.** Seien $x = x_1 \dots x_n$ und $y = y_1 \dots y_m$ Sequenzen über einem endlichen Alphabet Σ. Ohne Beschränkung der Allgemeinheit gelte $n \leq m$. Der *erweiterte Hamming-Abstand* zwischen x und y ist definiert als

$$\Delta_{h_x}(x, y) := \Delta_h(x, y_1 \dots y_n) + m - n.$$

Um eine Heuristik für k-LCS zu erhalten, wird anstelle des Hamming-Abstands in Abbildung 3.4 der erweiterte Hamming-Abstand verwendet. Nach Konstruktion des Algorithmus ist sichergestellt, dass nur gemeinsame Teilsequenzen berechnet werden. Es kann jedoch vorkommen, dass keine gemeinsame Teilsequenz gefunden wird. In diesem Fall wird die leere Sequenz $\langle\rangle$ zurückgegeben.
Nach den obigen Ausführungen kann also als Ergebnis festgehalten werden:

Satz 3.5.2. *Der Algorithmus* HAMMINGLCS *berechnet bei Eingabe von k Sequenzen S_i mit $0 < i \leq k$ und maximaler Länge $n = \max_{0<i\leq k} |S_i|$ in Zeit $O(k^4 n + k^2 n^2)$ eine gemeinsame Teilsequenz aller S_i, unter Verwendung des erweiterten Hamming-Abstands als Distanzmaß.*

□

3.6 Kürzungsstrategien

Für das Kürzen einer (längsten) gemeinsamen Teilsequenz S auf die maximale Länge *maxlen* sind viele Strategien denkbar. In Schritt 7 aus Abschnitt 3.2.2 wird einer der folgenden Kürzungsalgorithmen benutzt, wobei davon ausgegangen wird, dass $|S| >$ *maxlen* gilt:

- TRIMFIRST – Entfernt die ersten $|S|$ – *maxlen* Elemente aus S.
- TRIMLAST – Entfernt $|S|$ – *maxlen* Elemente am Ende von S.
- TRIMRANDOM – Entfernt $|S|$ – *maxlen* Elemente an zufälligen Positionen von S.
- TRIMSKIP – Entfernt $|S|$ – *maxlen* Elemente aus S, dabei wird S als Ringpuffer aufgefasst und das jeweils $|S|/5$-te Element (gerundet) entfernt.

TRIMSKIP ist eine willkürlich gewählte Kürzungsstrategie. Sie kann jedoch verwendet werden, wenn Elemente von beliebigen Positionen entfernt werden sollen, die Ergebnisse der Kürzung aber über mehrere Ausführungen des in Abschnitt 3.2.2 vorgestellten Verfahrens vergleichbar sein sollen.
Bei Verwendung von veketteten Listen können Elemente aus S in Zeit $O(1)$ entfernt werden.

3.7 Konstruktion eines regulären Ausdrucks aus einer gemeinsamen Teilsequenz

Der in Abschnitt 3.4 vorgestellte und in Abschnitt 3.5 verallgemeinerte Algorithmus hat den Nachteil, dass nach der Berechnung einer gemeinsamen Teilsequenz *cs* keine Information darüber vorliegt, ob zusammenhängende Teilstrings von *cs* auch in allen Sequenzen der Eingabe zusammenhängend vorkommen. Soll – wie in Schritt 11 aus Abschnitt 3.2.2 beschrieben – aus einer gemeinsamen Teilsequenz einer gegebenen Menge S von Bytes-Sequenzen ein regulärer Ausdruck konstruiert werden, der alle Sequenzen aus S akzeptiert, führt das Fehlen dieser Information zunächst zu regulären Ausdrücken mit unnötig vielen Wildcard-Symbolen.
Beispielsweise haben die Sequenzen `abbacd`, `acbadb` und `abbadc` die gemeinsame Teilsequenz `abad`. Da aus der Betrachtung der gemeinsamen Teilsequenz nicht klar wird, dass `ba` in allen Sequenzen zusammenhängend vorkommt, führt die naive Konstruktion

eines regulären Ausdrucks zu einem Ausdruck der Form[7] ⌜a*b*a*d⌟. Ein kürzerer Ausdruck wäre jedoch ⌜a*ba*d⌟[8]. Hier wird sofort ersichtlich, dass ab in allen Sequenzen zusammenhängend vorkommt. Zusammenhängende Teilstrings sind von Scannern wie ClamAV außerdem effizienter zu erkennen[9].

Abbildung 3.7 zeigt den Pseudocode eines rekursiven Algorithmus, der aus einer gemeinsamen Teilsequenz *cs* einer Menge von Sequenzen $\mathcal{S}$ einen regulären Ausdruck in ClamAV-Syntax konstruiert und dabei $l-1$ Wildcard-Symbole ⌜*⌟ einfügt, falls *cs* aus l zusammenhängenden Teilstrings besteht. Der Aufrufparameter *cur* hat dabei initial den Wert 1, und *regex* ist zu Beginn leer.
Es gilt zunächst: Das erste Element cs_{cur} der gemeinsamen Teilsequenz *cs* muss Teil eines konstruierten regulären Ausdrucks *regex* sein und wird diesem hinzugefügt. Anschließend wird die Einbettung des Elements cs_{cur} in den Sequenzen aus $\mathcal{S}$ gesucht. Eine solche Einbettung gibt die Positionen von cs_{cur} innerhalb der einzelnen Sequenzen an. Die gefundenen Indizes werden in dem Feld *idx* gespeichert. Ausgehend von den Indizes aus *idx* wird nach Elementen gesucht, die in allen Sequenzen direkt auf cs_{cur} folgen. Dazu werden die Sequenzen $S_i \in \mathcal{S}$ ab der Position $idx[i]$ untereinander geschrieben und spaltenweise auf übereinstimmende Elemente überprüft. Für jede Spalte, deren Elemente übereinstimmen wird der Index *cur* erhöht und *regex* das nächste Element cs_{cur} hinzugefügt. *cur* gibt die Anzahl der bearbeiteten Elemente von *cs* an. Stimmen die Elemente einer Spalte nicht überein, wird *regex* ein Wildcard-Symbol ⌜*⌟ hinzugefügt und der Durchlauf abgebrochen. Das Verfahren wird anschließend beginnend mit dem *cur*-ten Element von *cs* rekursiv aufgerufen. Sind alle Elemente bearbeitet, wird *regex* zurückgegeben.

Satz 3.7.1. *Der Algorithmus* BUILDREGEX *aus Abbildung 3.7 berechnet in Zeit* $O((k+|cs|)n)$ *aus einer gemeinsamen Teilsequenz cs von k gegebenen Sequenzen* $\mathcal{S}=\{S_1,\dots,S_k\}$ *einen regulären Ausdruck regex in ClamAV-Syntax, der die Sequenzen aus* $\mathcal{S}$ *akzeptiert und dabei* $l-1$ *Wildcard-Symbole in regex einfügt, wobei l die Anzahl der zusammenhängenden Teilsequenzen aus cs und n die Länge der längsten Sequenz aus* $\mathcal{S}$ *bezeichnet.*

Beweis. Der Algorithmus durchläuft die Elemente der Sequenzen $\mathcal{S}$ zum Finden der Einbettungen der Elemente aus *cs*. Stehen diese Zeichen am Ende der Sequenz, werden für die Suche nach allen Einbettungen insgesamt höchstens $O(kn)$ Operationen

[7] Es wird die ClamAV-Syntax verwendet
[8] Ein noch spezifischerer regulärer Ausdrück wäre ⌜a??ba{-1}d⌟, hier gibt es noch Raum für Erweiterungen.
[9] Zusammenhängende Teilstrings stehen zum Beispiel in dem von ClamAV zum Scannen konstruierten Suchbaum in einem Knoten.

BUILDREGEX($\mathcal{S} = \{S_1, \ldots S_k\}, cs, cur, regex$)

```
 1  if cur = |cs|                          ▷ Abbruchbedingung, cs komplett durchlaufen
 2     then return regex
    ▷ Das erste Element von cs muss Teil des regulären Ausdrucks sein
 3  regex ← regex · cs_cur
    ▷ Nach der Einbettung des aktuellen Elements in allen Sequenzen suchen
 4  idx ← dim(k)                           ▷ Indizes des Elements in den Sequenzen
 5  c ← 0
 6  for each S ∈ 𝒮
 7       do for i ← cur to |S|
 8              do if cs_cur = S_i
 9                    then idx[c] ← i
10                         c ← c + 1
11                         break           ▷ Mit nächster Sequenz fortfahren
    ▷ Nach zusammenhängenden Elementen suchen
12  j ← 0
13  while cur < |cs|
14       do for i ← 1 to k
15              do S ← 𝒮_i
16                 if cs_cur ≠ S_{idx[i]+j}
17                    then regex ← regex · *          ▷ Wildcard hinzufügen
18                         break 2                    ▷ Beide Schleifen verlassen
19          cur ← cur + 1, j ← j + 1
20          regex ← regex · cs_cur
21  return BUILDREGEX(𝒮, cs, cur, regex)
```

Abbildung 3.5: Konstruktion eines regulären Ausdrucks aus einer gemeinsamen Teilsequenz einer Menge von Sequenzen

benötigt. Ist eine Einbettung gefunden, wird nach zusammenhängenden Elementen in den Sequenzen gesucht. Besteht die gemeinsame Teilsequenz *cs* aus einem einzigen zusammenhängenden String, werden dafür maximal $O(|cs|n)$ Operationen benötigt, zusammen also $O((k + |cs|)n)$.
Nach Konstruktion werden genau dann Wilcard-Symbole zu *regex* hinzugefügt, wenn ein zusammenhängender Teilstring in *cs* unterbochen wird. Dies kann jedoch höchstens $l - 1$ mal vorkommen.
Der Algorithmus terminiert, da in jedem rekusiven Aufruf der Index *cur* in *cs* um mindestens Eins erhöht wird, und es somit höchstens $|cs| - 1 \leq n$ solcher Aufrufe gibt. □

4 Implementierung

Dieses Kapitel beschreibt die im Laufe dieser Arbeit entstandene prototypische Implementierung des im vorangegangenen Kapitel 3 vorgestellten Verfahrens. Zunächst wird ein Überblick über das im Laufe dieser Arbeit entwickelte Programm gegeben und die verwendeten Programmiersprachen und Softwarewerkzeuge vorgestellt.
Sozusagen als „Nebenprodukt“ der Implementierung ist mit IDAJava ein Softwarewerkzeug für den Disassembler IDA Pro entstanden, das eine vollständige Steuerung aller verfügbaren Funktionen des Disassemblers ermöglicht und sich für die automatisierte Analyse von Programmen als sehr hilfreich erwiesen hat. Da in dem Hauptprogramm zur Signaturgenerierung von diesen Automatisierungsmöglichkeiten ausgiebig Gebrauch gemacht wird, erläutert Abschnitt 4.3 zunächst die Software-Architektur von IDAJava. Es folgt eine Beschreibung des entwickelten Hauptprogramms und seiner Komponenten, wobei der Schwerpunkt auf den Besonderheiten bei der Implementierung der verwendeten Algorithmen liegt.
Der letzte Teil dieses Kapitels widmet sich einigen Ausführungen zu den aufgetretenen Schwierigkeiten und Problemen bei der Implementierung des Verfahrens.

4.1 Überblick

Im Verlauf dieser Arbeit ist das kommandozeilenorientierte Programm SigGen entstanden, das eine ClamAV-Signatur für eine Familie von Schadprogrammen generiert. Die Klassifizierung einer Menge von Malware in Familien erfolgt dabei vorab mit dem bereits in Kapitel 2.6.1 vorgestellten Softwarewerkzeug VxClass. Ein typischer Arbeitsablauf mit dem Programm SigGen ist der folgende:

1. Eine Menge von – beispielsweise mit dem Honeypot Nepenthes [3] – gesammelter Malware wird über die Web-basierte Benutzerschnittstelle von VxClass hochgeladen, und der Klassifikationsprozess wird gestartet[1].

2. Nach erfolgter Klassifikation wird eine Malware-Familie, für die eine Signatur generiert werden soll, ausgewählt. Das Disassembler-Teilsystem von VxClass hat für jede ausführbare Malware-Datei der Familie eine IDA-Pro-Datenbank angelegt. Diese werden (ebenfalls über die Web-basierte Benutzerschnittstelle) heruntergeladen und dienen SigGen als Eingabe.

3. Vor dem ersten Start einer Signaturgenerierung sind zunächst noch einige Konfigurationsoptionen einzustellen. Bei den Konfigurationsoptionen handelt es sich im Wesentlichen um die Angabe des Speicherorts der IDA-Pro-Datenbanken für die einzelnen Schadprogramme und um Angaben für den Speicherort der erzeugten Ausgabedateien[2], sowie um Dateipfade für die ausführbaren Dateien von IDA Pro und BinDiff. Nicht spezifizierte Angaben werden mit Standardeinstellungen initialisiert.

4. Es folgt der eigentliche Programmaufruf über die Kommandozeile. Je nach Größe und Komplexität der Malware kann die Generierung einer Signatur einige Minuten dauern. Während der Ausführung von SigGen werden die IDA-Pro-Datenbanken geöffnet und in ein XML-Format exportiert (hierfür wird ein separates IDA-Pro-Plugin verwendet). Dieses Format dient als Eingabe für das Programm BinDiff, das so genannte *Difference Reports* ebenfalls in einem XML-Format speichert. Die einzelnen Difference Reports werden eingelesen, und es wird das in Kapitel 3 vorgestellte Verfahren angewendet. Die für die Signaturgenerierung benötigten Byte-Sequenzen werden mit IDA Pro und dem eingangs erwähnten Softwarewerkzeug IDAJava aus den IDA-Pro-Datenbanken extrahiert.
   Die so erzeugte ClamAV-Signatur im erweiterten Signaturformat wird in einer Signaturdatenbank gespeichert und ist damit für den Scanner von ClamAV sofort verfügbar.

5. Die neue Signatur wird zum Beispiel in Bezug auf falsche Positive getestet, oder es werden andere Experimente durchgeführt.

[1] Dieser Vorgang kann – abhängig vom Umfang und der Anzahl der hochgeladenen Malware – einige Minuten bis Stunden dauern.
[2] Diese Angaben können alternativ auch über die Kommandozeile übergeben werden

Die verschiedenen Parameter und Aufrufarten von SigGen werden in Listing 4.4 dargestellt.

4.2 Verwendete Programmiersprachen

In den folgenden Abschnitten werden die verwendeten Programmiersprachen und Klassenbibliotheken beschrieben.

4.2.1 Java

Das Programm SigGen wurde in der Programmiersprache Java [21] geschrieben und benötigt die aktuelle[3] Version 1.6.0 des *Java Development Kit* (JDK).
Im Gegensatz zu vielen anderen Programmiersprachen werden in Java geschriebene Programme in der Regel nicht direkt in ausführbare Programme kompiliert, sondern in eine Zwischensprache für eine virtuelle Befehlsarchitektur, die *Byte-Code* genannt wird. Die Ausführung auf dem jeweiligen Rechensystem erfolgt über die Laufzeitumgebung (*Java Runtime Environment, JRE*) von Java, welche den Byte-Code in die Befehle für die Befehlsarchitektur des Rechensystems übersetzt. Dies geschieht dabei für Programme wie SigGen völlig transparent. Ein Vorteil dieses Ansatzes ist die Plattformunabhängkeit, das heißt, in Java geschrieben Programme können ohne erneutes Kompilieren auf verschiedenen Prozessor- und Befehlsarchitekturen sowie Betriebssystemen ausgeführt werden. Zum Aufruf von plattformabhängigen ausführbaren Code steht das *Java Native Interface* (JNI) zur Verfügung. Mit JNI können in Java geschriebene Programme Funktionen aus Programmbibliotheken aufrufen, die in anderen Programmiersprachen programmiert worden sind (siehe Abschnitt 4.3.2).

Für die Programmiersprache Java sprechen ebenfalls einige Vorteile, allen voran die langjährige Erfahrung des Autors dieser Arbeit mit Java. Die weite Verbreitung und die Verfügbarkeit von umfangreichen Klassenbibliotheken – sowohl die mit Java mitgelieferten, als auch solche von Drittherstellern – erleichtern die Anwendungsentwicklung. Das Programm SigGen macht von den angebotenen, fertig einsetzbaren und effizienten Datenstrukturen intensiven Gebrauch.
Die Möglichkeit des entfernten Methodenaufrufs (*Remote Procedure Call*, RPC) mit der

[3] Stand: Mai 2008

Java-Technologie RMI (*Remote Method Invocation*), die – im Gegensatz zu anderen Techniken – kaum Änderungen an bestehender Programmlogik für die Kommunikation über ein Netzwerk[4] erfordert, ist ein weiterer Vorteil von Java.

Für die Programmierung und das Testen der einzelnen Java-Klassen der in diesem Kapitel vorgestellten Programme SigGen und IDAJava wurde die integrierte Entwicklungsumgebung Eclipse verwendet.

4.2.2 C++

Das in Abschnitt 4.3 vorgestellte Plugin für IDA-Pro ist in der Programmiersprache C++ [56] geschrieben. Diese Programmiersprache ist ebenfalls weit verbreitet und wird als Industriestandard in vielen Bereichen angesehen. C++-Programme werden üblicherweise in nativ auf dem jeweiligen Zielsystem ausführbaren Code kompiliert. Da C++ eine objektorientierte Erweiterung der Programmiersprache C [33] ist, werden – unter anderem – dieselben Aufrufkonventionen wie in C unterstützt. Dies macht C++ zur ersten Wahl bei der Programmierung von Plugins für IDA Pro. Die Programmierschnittstelle von IDA Pro ist außerdem am einfachsten mit C oder C++ nutzbar.[5] Zur Vermeidung von häufigen Fehlern und Standard-Problemen mit C++[6] wird die Klassenbibliothek STL (*Standard Template Library*) verwendet. Diese bietet ebenfalls fertige Datenstrukturen zur direkten Verwendung an.

Die Programmierung und das anschließende Testen des IDA-Pro-Plugins von IDAJava wurden mit der Entwicklungsumgebung Microsoft Visual Studio Team System 2008 durchgeführt.

4.3 Schnittstelle zu IDA Pro und BinDiff

Wie Abschnitt 4.1 zu entnehmen ist, werden während der Ausführung von SigGen Funktionen der Programme IDA Pro und BinDiff aufgerufen. Während sich der Aufruf von BinDiff auf den Aufruf der ausführbaren Programmdatei und anschließendem

[4] Das zugrundeliegende Netzwerkprotokoll trägt ebenfalls den Namen RMI.

[5] Mit dem Plugin IDAJava versucht der Autor dieser Arbeit hier eine Brücke zu schlagen.

[6] Besonders hervorzuheben ist das Fehlen eines eigenen Datentyps für Zeichenketten. Diese werden daher meist als Felder von Zeichen verwaltet, deren Ende durch ein Null-Byte markiert wird. Viele der von Malware ausgenutzten Sicherheitslücken sind letztlich auf den fehlerhaften Umgang mit Zeichenketten zurückzuführen.

Einlesen der erzeugten Ausgabedateien beschränkt, ist das Zusammenspiel mit IDA Pro deutlich komplexer und leistungsfähiger.

4.3.1 IDAJava

IDAJava ist ein Softwarewerkzeug zur Automatisierung des Disassemblers IDA Pro. Abbildung 4.1 zeigt einen Überblick über die Softwarearchitektur der einzelnen Komponenten von IDAJava. Das Programm SigGen wird dabei als aufrufendes Programm dargestellt.

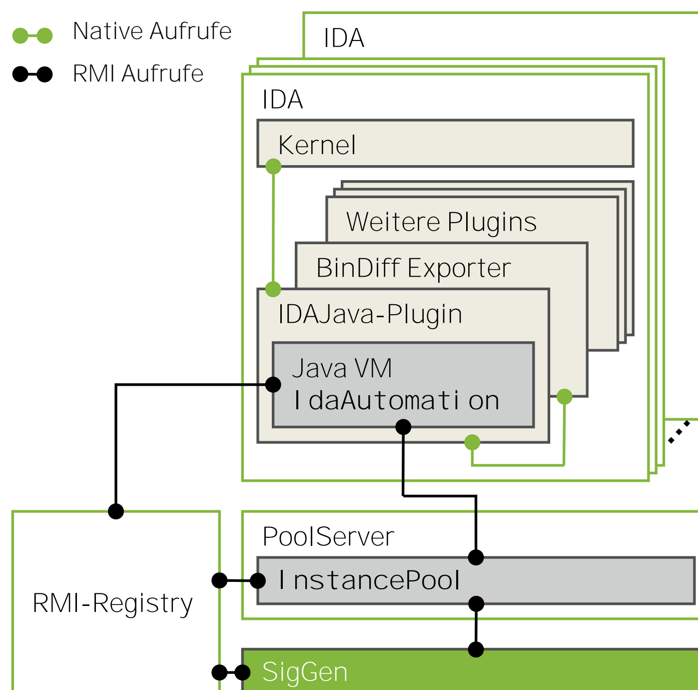

Abbildung 4.1: Blockdiagramm der von IDAJava bereitgestellten Infrastruktur

IDAJava besteht im Wesentlichen aus drei Komponenten: einem in der Programmiersprache C++ geschriebenen Plugin für IDA Pro, einer in der Programmiersprache Java geschriebenen Automatisierungskomponente und einer – ebenfalls in Java programmierten – Server-Komponente.
Soll ein Java-Programm wie SigGen die Funktionen des Disassembler benutzen, geschieht dies unter Verwendung des RMI-Protokolls in mehreren Schritten. Zunächst wird vom aufrufenden Programm in dem – von der Java Laufzeitumgebung bereitgestellten –

Namensverzeichnis (*RMI-Registry* genannt) nach einer Server-Komponente[7] gesucht und eine neue Instanz des Disassemblers mit der Angabe einer zu öffnenden Datei[8] angefordert. Die Server-Komponente startet einen neuen IDA-Pro-Prozess mit der angegebenen Datei. Beim Öffnen der Datei wird von IDA Pro das IDAJava-Plugin ausgeführt. Das Plugin erzeugt eine neue Java-VM und startet die Automatisierungskomponente[9]. Die Automatisierungskomponente registriert sich wiederum bei der Server-Komponente als verfügbare Disassembler-Instanz und wird von der Server-Komponente über RMI an das aufrufende Programm zurückgegeben. Die weitere Kommunikation läuft anschließend direkt zwischen dem aufrufenden Programm und der Automatisierungskomponente ab, letztere ruft die Funktionen des Disassemblers und anderer Plugins über das Java Native Interface auf.

Abbildung 4.2 zeigt das Klassendiagramm des C++-Plugins in UML-Notation. Die Klasse `IDAJavaPlugin` ist von der im IDA Pro SDK (*Software Development Kit*, SDK) definierten Klasse `plugin_t` abgeleitet und benutzt zur Erzeugung einer Java VM die Hilfsklasse `JavaVMCreator`. Objekte der Klassen `JavaVM` und `JNIEnv` sind Bestandteil der JNI-Schnittstelle.

In Abbildung 4.3 werden die Klassen der in Java geschriebenen Server- und Automatisierungskomponenten dargestellt. Die Klasse `PoolServer` realisiert die Server-Komponente und wird beim Aufruf über das RMI-Protokoll über die Schnittstelle `IDAJavaInstancePool` angesprochen. Ähnlich verhält es sich bei den Klassen der Automatisierungs-Komponente: Die Klasse `IDARemoteAutomation` ist von `IDAAutomationPlugin` abgeleitet und wird über RMI über die Schnittstelle `IDARemoteAutomation` aufgerufen. Die Schnittstelle `IDAJavaConstants` und die Klasse `IDAJava` realisieren die Anbindung an die Funktionen des Disassemblers mittels SWIG (siehe Abschnitt 4.3.2).

Beispiel

Obwohl das Zusammenspiel von IDAJava, IDA Pro und dem aufrufenden Programm zunächst kompliziert anmutet, gestaltet sich die Verwendung der angebotenen Dienste recht komfortabel. Listing 4.1 zeigt den Java-Quelltext eines Beispielprogramms, das einen Text im Nachrichtenfenster von IDA Pro ausgibt. Der Disassembler wird am Ende

[7] Im Quelltext wird diese durch die Klasse `PoolServer` realisiert.

[8] Die Angabe eines Dateinames beim Starten von IDA Pro ist der einzige dokumentierte Weg, um in IDA Pro automatisiert eine Datei zu öffnen. Hier besteht von Seiten des Herstellers Nachholbedarf.

[9] Diese ist im Quelltext durch die Klasse `IDARemoteAutomationPlugin` realisiert.

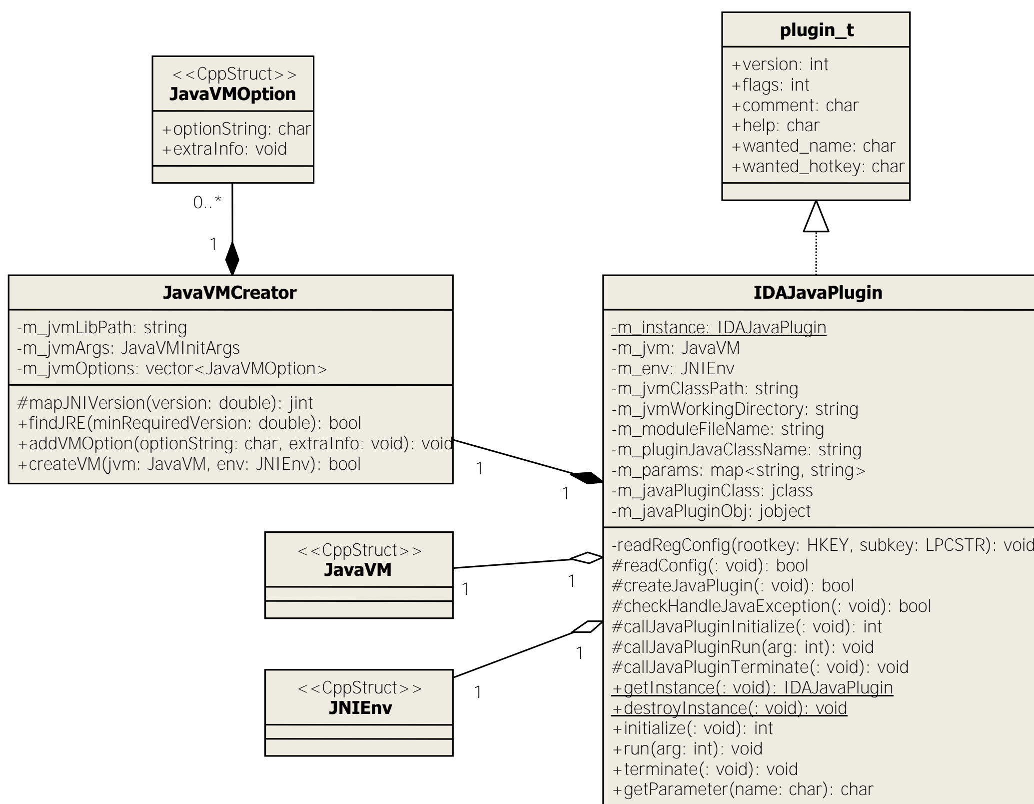

Abbildung 4.2: UML-Klassendiagramm des IDA-Pro-Plugins von IDAJava

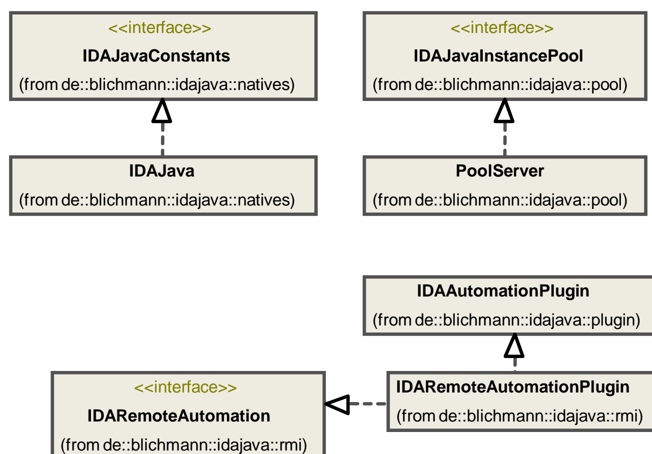

Abbildung 4.3: UML-Klassendiagramm der Server- und Automatisierungskomponente von IDAJava

des Programms bewusst nicht beendet, damit der ausgegebene Text gelesen werden kann.

4.3.2 SWIG

Die Anbindung der Automatisierungs-Komponente von IDAJava an IDA Pro ist mittels SWIG (*Simplified Wrapper and Interface Generator*) realisiert [6]. SWIG ist ein Programm zur Erzeugung von Schnittstellen von und zur Programiersprache C++ und anderen Programmiersprachen. SWIG wurde ursprünglich entwickelt, um von einem in der Programmiersprache Tcl/Tk geschriebenen Programm aus in C++ geschriebenen ausführbaren Code aufzurufen. Seitdem sind eine ganze Reihe von sogenannten *Bindings* (engl. etwa *Anbindungen*) für andere Programmiersprachen entwickelt worden, wobei eine jeweils Programmiersprachen-spezifische Methode für den Aufruf von in C++ geschriebenen ausführbarem Code verwendet wird. Für Java wird hierfür JNI verwendet. Das Programm SWIG lässt sich bezüglich der Funktionalität mit einem C++-Compiler[10] vergleichen, der anstelle von ausführbarem Code Quelltext für den Aufruf aus anderen Programmiersprachen erzeugt. Intern werden dabei die Aufrufkonventionen von C verwendet. Häufig genügt es, als Eingabe die Header-Dateien des aufzurufenden C++-Quelltexts zu verwenden. Im Fall von IDAJava sind dies die Header-Dateien des

[10] Tatsächlich implementiert SWIG alle Sprachelemente von C++ und einen kompletten C99-Präprozessor.

```
1  import java.io.*; import java.rmi.*;
2  import de.blichmann.idajava.rmi.*;
3  import de.blichmann.idajava.util.*;
4  import de.blichmann.idajava.pool.*;
5
6  public class IDAJavaHello {
7          public static void main(String[] args) throws Exception {
8                  // Server-Komponente im Namensverzeichnis suchen
9                  IDAJavaInstancePool ip = (IDAJavaInstancePool) Naming.lookup(
10                                 "///" + IDAJavaInstancePool.DEFAULT_NAME);
11
12                 // Datei aus Kommandozeile mit IDA Pro öffnen
13                 IDARemoteAutomation ida = ip.createInstance(args[1], 0);
14
15                 PrintStream ps = new PrintStream(
16                         new IDAConsoleOutputStream(ida), true);
17
18                 // Einen Text im IDA-Pro-Nachrichtenfenster ausgeben
19                 ps.println("Hello,␣World␣of␣IDA!"); } }
```

Listing 4.1: Ein Beispiel für die Verwendung von IDAJava

IDA Pro SDK, für die jedoch einige kleinere Anpassungen erforderlich sind, um sie als Eingabe für SWIG zu verwenden.

4.4 Programmbeschreibung

SigGen ist ein stark algorithmisch orientiertes Programm, dessen Kernkomponente die in dem UML-Klassendiagramm aus Abbildung 4.4 dargestellte Klasse `ClamAVSignatureGenerator` ist. Die – mit Ausnahme von Methoden zum Setzen von Parametern der Signaturgenerierung – einzige öffentliche Methode dieser Klasse ist die Methode `generate()`, die prozedural programmiert ist und sich streng an die Beschreibung aus Kapitel 3.2.2 hält.
Daneben gibt es eine ganze Reihe von Java-Klassen, die keinen direkten Bezug zu der eigentlichen Signaturgenerierung haben und die Funktion eines Frameworks erfüllen:

- SigGen – Diese Klasse enthält das über die Kommandozeile aufzurufende Hauptprogramm.
- CommandLineParser – Wie der Name schon andeutet, wird diese Klasse zur Auswertung der übergebenen Kommandozeilenparameter verwendet.
- Config – Kapselt die Funktionalität der verwendeten Konfigurationsdatei. Alle Einstellungen und Parameter des Programms werden in dieser Klasse zentral verwaltet.
- CoreGraphIsoHandler, ControlFlowGraphIsoHandler – Hilfsklassen für den zum Parsen der von BinDiff erzeugten XML-Dateien verwendeten SAX-Parser[11].

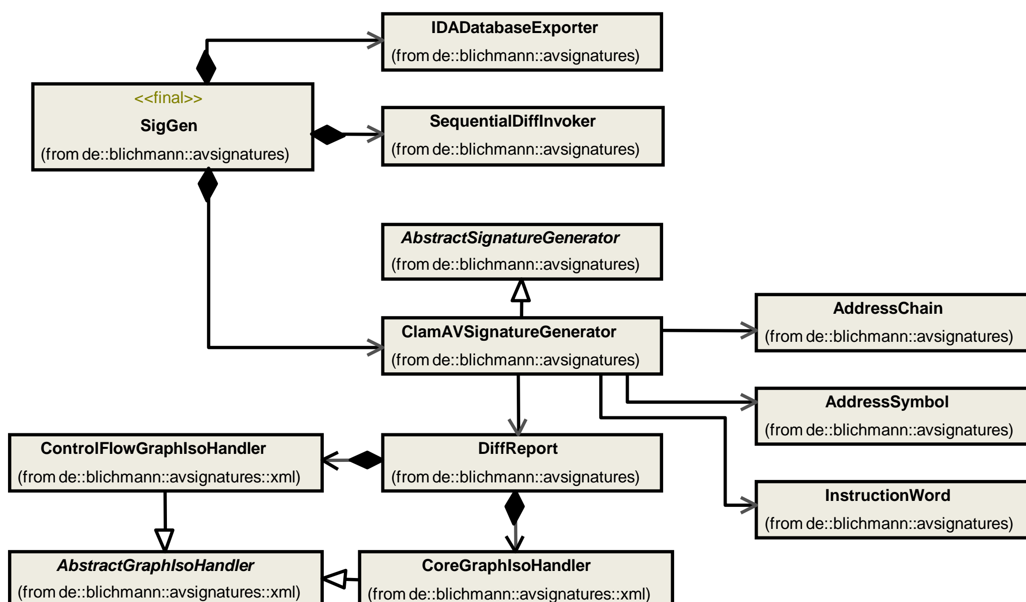

Abbildung 4.4: UML-Klassendiagramm einer Auswahl von Klassen in SigGen

4.4.1 Implementierung des Rahmenprogramms

Nachdem die Kommandozeile und die Konfigurationsdatei ausgewertet wurden, arbeitet das in der Klasse SigGen realisierte Rahmenprogramm die in Kapitel 3.2.2 beschriebenen

[11] *Simple API for XML*, SAX

Schritte der Reihe nach ab. Listing 4.2 zeigt den betreffenden Ausschnitt aus dem Quelltext.[12]

```
1   // ...
2   // IDA Datenbanken nach XML exportieren
3   final IDADatabaseExporter ide = new IDADatabaseExporter();
4   ide.set...(... // Parameter setzen
5   ide.export(idbFilenames, ...
6
7   // BinDiff auf den exportierten XML Dateien aufrufen
8   final SequentialDiffInvoker sdi = new SequentialDiffInvoker();
9   sdi.set...(... // Parameter setzen
10  if (!sdi.invokeDiff(...)) { ...
11          return; }
12
13  // Signatur erzeugen
14  final ClamAVSignatureGenerator csg = new ClamAVSignatureGenerator();
15  csg.set...(... // Parameter setzen
16  if (!csg.generate(idbFilenames, ...)) { ...
17          return; }
```

Listing 4.2: Ausschnitt aus dem Rahmenprogramm

Die `generate()`-Methode der Klasse `ClamAVSignatureGenerator` setzt dies fort (siehe Listing 4.3).

4.4.2 Datenstrukturen und Optimierungen

Die in Kapitel 3.2.2 verwendeten Matrizen werden – wenn mögich – durch Klassen der Java-Klassenbibliothek nachgebildet. Zeile 10 aus Listing 4.3 ist dafür ein gutes Beispiel: Der DatenTyp `ArrayList<ArrayList<AddressSymbol>>` bildet die Matrix W aus Schritt 5 nach, der Datentyp `AddressSymbol` einen Eintrag $w_{i,j} = (\pi_i(j), b_{i,j})$. Tatsächlich enthält die Deklaration von `AddressSymbol` die beiden Felder `id` und `addr`, die die beiden Elemente des Eintrags speichern.
Listen werden in SigGen allgemein durch die Standardklasse `ArrayList<T>` realisiert,

[12] Der angebene Java-Quelltext ist stark verkürzt wiedergegeben und enthält keinerlei Fehlerüberprüfungen.

```
1          ...
2          // BinDiff Dateien parsen
3          logger.info("Parsing␣diff␣reports...");
4          for (DiffReport dr : diffReports) dr.parse();
5
6          // Kandidatenfunktionen und -Basic-Blocks bestimmen
7          findCandidates();
8
9          // Das "Wort" der Basic-Blocks über alle Schadprogramme erzeugen
10         final ArrayList<ArrayList<AddressSymbol>> bbWords =
11                 basicBlockPermutations();
12
13         // Berechnung einer stabilen Reihenfolge mit dem k-LCS-Algorithmus auf
14         // Permutationen. Die resultierende Sequenz bei Bedarf kürzen
15         logger.info("Solving␣constraint␣LCS-problem...");
16         final ArrayList<AddressSymbol> seq = trimSequence(LCS.hammingLCS(
17                         bbWords), null);
18
19         // Aus jedem "Wort" Elemente löschen, die nicht in der LCS vorkommen
20         pruneBasicBlockSequences(bbWords, seq);
21         ...
```

Listing 4.3: Ausschnitt aus der Methode `generate()`

wobei T den Datentyp der Liste bezeichnet. Wie oben beschrieben, kommen für Matrizen und (große) zweidimensionale Felder auch verschachtelte Listen zum Einsatz. Der Grund dafür liegt in der effizienten Implementierung und der geschickten Speicherverwaltung[13] dieser Standardklasse.

Weitere Optimierungen werden dadurch erreicht, dass Elemente in Datenstrukturen an geeigneten Stellen sortiert werden, etwa um Abbruchbedingungen von Schleifen schneller erfüllen zu können. Diese Optimierung wird beispielsweise in der Funktion `findCandidates()` zum Suchen von Kandidatenfunktionen und -Basic-Blocks verwendet.

Für das schnelle Auffinden von Funktionen und Basic-Blocks anhand ihrer Adressen

[13] Die Speicherverwaltung erfolgt in Java automatisch, dennoch kann sich eine günstige Allozierung von Speicher positiv auf die Performanz der Programms auswirken.

wird außerdem die Bibliotheksklasse `TreeMap` verwendet. Diese Klasse ist mit einem Rot-Schwarz-Baum implementiert und bietet daher gute Laufzeiten für die einzelnen Operationen.

4.4.3 Einschränkungen von ClamAV-Signaturen

Die direkte Implementierung des Signaturgenerierungsverfahrens führt zur Ausgabe von Signaturen, die von dem Scanner von ClamAV nicht zum Erkennen von Schadprogammen verwendetet werden können. Dies ist darin begründet, dass in ClamAV-Signaturen zwischen zwei Wildcard-Symbolen ⌜*⌟ immer mindestens zwei hexadezimale Bytes stehen müssen. Das gleiche gilt für den Anfang und das Ende der Signatur: Ein Wildcard-Symbol darf, gerechnet vom Beginn oder vom Ende der Signatur, nicht an der zweiten Byte-Position einer hexadezimalen ClamAV-Signatur stehen. Diese Einschränkung wird in [35] beschrieben und bei der Implementierung der Funktion `buildRegEx()` berücksichtigt.

4.4.4 Unterstützte Plattformen

Da SigGen in der Programmiersprache Java geschrieben ist, ist das Programm prinzipiell auf jeder Plattform[14] lauffähig, für die eine Implementierung einer Java VM existiert. Das Programm BinDiff ist jedoch momentan[15] nur für das Microsoft-Windows-Betriebssystem und die IA-32 Befehlsarchitektur verfügbar. Das IDAJava IDA-Pro-Plugin wurde ebenfalls primär für diese Plattform erstellt, die enthaltenen Funktionen sollten sich jedoch vergleichsweise einfach auf andere Plattformen portieren lassen.

4.4.5 Benutzerschnittstelle

Da SigGen ein kommandozeilenorientiertes Programm ist, beschränkt sich die Benutzerschnittstelle auf die Ausgabe von Meldungen während der Signaturgenerierung und die Anzeige der in Listing 4.4 gezeigten Hilfeseite beim Aufruf mit dem Parameter `-h`.

[14]Mit *Plattform* wird in diesem Zusammenhang eine Kombination aus Prozessor- und Befehlsarchitektur sowie einem Betriebssystem bezeichnet.
[15]Stand: Mai 2008

```
SigGen  version 0.1  Copyright (c)2007,2008 Christian Blichmann

Usage: java de.blichmann.avsignatures.SigGen [OPTION]... [SPEC]...
Generate a signature for the malware family specified by SPEC or one of the
dir-options. Currently, only ClamAV signatures can be generated.

Options:
  -h, --help                display this help and exit
      --check-scanner       check for the presence of an on-access scanner,
                             exit if one is found
  -i, --idb-dir IDBDIR      use IDA databases in IDBDIR
  -x, --xml-dir XMLDIR      store XML exports in XMLDIR
  -r, --reports-dir RPTDIR  store BinDiff reports in RPTDIR
  -b, --max-basic-blocks    maximum number of basis
  -t, --trim-strategy NUM   set the trim strategy to shorten long basic block
                             sequences when needed: 0 - trim first, 1 - skip
                             some blocks, 2 - remove random bytes, 3 - trim
                             last bytes
  -n, --clamav-ndb FILE     save generated signature to FILE
  -p, --name-prefix PREFIX  set a naming prefix in signature
  -m, --malware-type TYPE   set malware type in signature to TYPE
  -f, --force               overwrite existing files
  -S, --save-temps          do not delete temporary files
      --no-dotlock          disable lock-file
      --version             output version information and exit

Options override configuration files. Every database is exported to XML,
existing exports and BinDiff reports are preserved, unless ‘-f’ is specified.
Use ‘--’ to separate options from SPECs. For unspecified options default values
from siggen_conf.xml are used.

Report bugs to <siggen-bug@blichmann.de>.
```

Listing 4.4: SigGen Kommandozeilen-Optionen

4.5 Probleme

Zu Beginn der Implementierungphase war noch völlig unklar, wie genau eine Automatisierung des Disassemblers zu realisieren sei. Eine frühe experimentelle Version des Teils des Verfahrens, das IDA Pro auf einer Disassembler-Datenbank aufruft und diese in ein XML-Format exportiert, wurde daher zunächst mit Shell- und IDC-Skripten implementiert. Schließlich wurde mit IDAJava ein Softwarewerkzeug zur Automatisierung des Disassemblers entwickelt.

Bei der Beschäftigung mit der Programmierschnittstelle von IDA Pro wurde schnell klar, dass die bestehende Dokumentation zu definierten Funktionen und Datenstrukturen häufig ungenau und stellenweise sogar falsch ist. Hinzu kommt, dass die Benennung von API Funktionen völlig uneinheitlich ist.

Ein weiteres Problem trat beim Testen des IDAJava-Plugins auf: Die in den IDA-Pro-Prozess eingebettete Java VM beendete bei Java-Fehlern oder bei einem – laut Dokumentation des Herstellers am Ende eines Programms empfohlenen – Aufruf von `DestroyJavaVM()` den Prozess abrupt, so dass geöffnete Dateien nicht ordnungsgemäß geschlossen wurden. Da sich herausstellte, dass dieses Problem der Java VM bereits seit 1997 bekannt ist, wurde eine Methode implementiert, die den Disassembler-Prozess kontrolliert beendet.

Das Testen der Implementierung mit einem Debugger erwies sich als äußerst zeitaufwändig, da das gesamte Verfahren neu gestartet werden musste, sobald ein Fehler entdeckt wurde. Wie am Anfang dieses Kapitels in Abschnitt 4.1 beschrieben, kann ein Durchlauf von SigGen jedoch durchaus einige Minuten dauern. Bei zwei aufeinanderfolgenden Aufrufen von SigGen mit den gleichen Parametern werden daher bereits existierende Ausgabedateien wiederverwendet, um schnellere Testzyklen zu ermöglichen.

5
Bewertung

Die Ausführungen dieses Kapitels bewerten das in Kapitel 3 vorgestellte Verfahren, dessen protoypische Implementierung im vorangegangenen Kapitel näher erläutert wurde. Zunächst wird untersucht, in welchem Maße die in Abschnitt 3.1.1 definierten Ziele erreicht wurden.
Die von dem Verfahren erzeugten Signaturen werden bezüglich falscher Positive getestet und die dafür verwendete Testumgebung in Abschnitt 5.2.1 genauer beschrieben. Ein Ergebnis im Zusammenhang mit falschen Negativen wird in Abschnitt 5.3 diskutiert. Es folgen einige Kennzahlen zur Bewertung der Performanz des Prototyps im praktischen Einsatz.

5.1 Erreichte Ziele

Abschnitt 3.1.1 stellt einige Anforderungen an ein Verfahren zur automatisierten Signaturgenerierung. Für das Progamm SigGen, das ein derartiges Verfahren realisiert, gilt im Einzelnen:

1. *Schadprogramme werden vorab automatisiert in Familien klassifiziert.*
   Zur Klassifizierung von Schadprogrammen in Familien wird das in Abschnitt 2.6.1 vorgestellte Softwarewerkzeug VxClass verwendet. Die Klassifizierung erfolgt automatisiert und erfüllt damit die gestellte Anforderung.

2. *Das Disassemblieren und Finden von ähnlichen Fragmenten von Objektcode geschieht ohne jeden Benutzereingriff.*
   Diese Anforderung wird durch den automatisierten Aufruf des Disassemblers IDA Pro und des Softwarewerkzeugs BinDiff erfüllt.

3. *Die Erzeugung der Signaturen erfolgt auf Basis der Assemblerdarstellung der identifizierten ähnlichen Stellen.*
   Die in Schritt 9 aus Abschnitt 3.2.2 verwendeten Byte-Sequenzen werden aus der Disassembler-Datenbank der ausführbaren Datei der jeweiligen Malware extrahiert.

4. *Es wird eine gültige und möglichst kurze Signatur erzeugt. Wildcards und andere verfügbare Sprachmittel der Ziel-Signatursprache werden automatisch hinzugefügt.*
   Es wird der in Abschnitt 3.7 vorgestellte Algorithmus BUILDREGEX verwendet und erzeugt immer einen gültigen regulären Ausdruck[1]. Dieser Algorithmus fügt dem regulären Ausdruck für die Signatur genau ein Wildcard-Symbol weniger hinzu, als zusammenhängende Teilstrings in den gemeinsamen Teilsequenzen der jeweiligen Byte-Sequenzen der Schadprogramme auftreten. Dies sind bei Verwendung des Wildcard-Symbols ⌜*⌟ nicht mehr, als nötig. Der Algorithmus

5. *Es treten keine oder nur wenige falsche Positive und falsche Negative bei der Verwendung der Signatur auf.*
   Die Ausführungen zu falschen Positiven und falschen Negativen finden sich in den Abschnitten 5.2 und 5.3. Die an das Verfahren gestellten Anforderungen werden erfüllt.

6. *Die Laufzeit des gesamten Verfahrens ist möglichst kurz.*
   Die Bewertung der praktischen Performanz des Verfahrens wird in Abschnitt 5.5 vorgenommen, auch hier werden die gestellten Anforderungen erreicht.

5.2 Falsche Positive

In diesem Abschnitt werden die von SigGen erzeugten Signaturen in Bezug auf falsche Positive bei der Erkennung von Malware durch den Scanner ClamAV getestet. Dabei wurden für zehn Familien von Malware mit jeweils 5 bis 30 Mitgliedern Signaturen erzeugt, die anschließend in einer ClamAV-Signaturdatenbank gepeichert wurden.
Falsche Positive bezeichnen in diesem Zusammenhang, dass bei Verwendung der beschriebenen Signaturdatenbank zur Erkennung von Malware das Vorhandensein eines Schadprogramms gemeldet wird, obwohl es sich bei der betreffenden Datei nicht um Malware handelt.

[1] Dies kann auch der leere Ausdruck sein.

5.2.1 Testverfahren

Um die Reproduzierbarkeit eines Tests zu gewährleisten, wurde das Betriebssystem Microsoft Windows XP Professional[2] in eine virtuelle Maschine auf Basis von VirtualBox [57] installiert. Anschließend wurden alle verfügbaren Updates für das Windows-Betriebssystem eingespielt[3] und einige, für einen typischen *„Büroarbeitsplatz“* verwendete Softwarepakete installiert. Bei den in der virtuellen Maschine installierten Programmen handelte es sich um folgende: 7-Zip 4.57, Adobe Reader 8.1.2, Microsoft Office 2003 Professional, Mozilla Firefox 2.0.0.14, RealVNC 4.1, Skype 3.6.0, sowie der Java Laufzeitumgebung in Version 1.6.0_06 und das Microsoft .NET Framework 2.0.
Alle Dateien der virtuellen Maschine wurden anschließend mit dem Scanner ClamAV[4] und der erzeugten Signaturdatenbank gescannt.

Der Test mit der oben beschriebenen virtuellen Maschine wird durch die System-Scans von drei weiteren Rechensystemen ergänzt, die sich zur Zeit[3] im produktiven Einsatz befinden:

- *„Büro-Server“* – Betriebssystem Microsoft Windows 2003 Server for Small Business Server mit installiertem Microsoft Exchange Server 2003 und allen verfügbaren Updates[3], sowie Java 1.5 und .NET Framework 2.0. Dieses System wird produktiv eingesetzt als Datei- und Mailserver, und verwaltet einige 10 000 Emails und einige Gibibyte[5] Daten.
- *„Web-Host“* – Betriebssystem SuSE Linux 9.1 (32-bit) und Kernel-Version 2.6.18-Xen, sowie den installierten Programmen Apache, MySQL, PHP, Postfix, Spamassassin, VsFTP und Dovecot (jeweils in den von SuSE Linux 9.1 mitgelieferten Versionen). Dieses System wird als Web-, FTP-, Mail- und Datenbankserver eingesetzt und verwaltet ebenfalls einige 10 000 Emails und einige Gibibyte Daten.
- *„Entwickler“* – Betriebssystem Microsoft Windows Vista Business SP1 (32-bit). Unter anderem sind Java 1.6.0_06, Eclipse 3.3 und Microsoft Visual Studio 2008 Teamssuite Edition installiert. Dieses System wurde zur Entwicklung von SigGen verwendet.

[2] Der zur Installation verwendete Original-Datenträger enthielt bereits das Update *Service-Pack 2*.
[3] Stand: 15.05.2008
[4] Version 0.93
[5] Die Verwendung des Begriffs *Gibibyte* für eine Anzahl von 1 073 741 824 Bytes, sowie der Binärpräfixe MiB und GiB erfolgen nach IEC 60027-2.

Die Ergebnisse der einzelnen System-Scans wurden jeweils mit einem zweiten System-Scan unter Verwendung der ClamAV-eigenen Signaturdatenbank verglichen. Die gesammelten Daten werden in Tabelle 5.1 festgehalten.

	Büroarbeitsplatz	**Büro-Server**	**Web-Host**	**Entwickler**
Verzeichnissse	2 137	12 250	25 382	34 057
Dateien	21 315	164 845	329 594	279 053
Gescannte Daten	4,91 GiB	101,58 GiB	58,59 GiB	34,19 GiB
Zeit	7 863,69 s	41 232,70 s	6 056,48 s	8 512,35 s
Als infiziert erkannt	0 (8[a])	0	4[b]	0
Als infiz. erkannt (2. Lauf)	0 (8[a])	0	4	0

[a]Für den Scanner von ClamAV zu große Archivdateien.

[b]Tatsächliches Vorkommen eines Schadprogramms aus einer der Familien, für die eine Signatur generiert wurde.

Tabelle 5.1: Ergebnisse von kompletten ClamAV-Scandurchläufen

Bei Betrachtung der Tabelle fällt auf, dass vier von den Dateien des Rechensystems *„Web-Host“* als Schadprogramm erkannt wurden. Der Vergleich mit dem zweiten System-Scan unter Verwendung der mitgelieferten ClamAV-Signaturdatenbank und eine anschließende manuelle Analyse bestätigten das Vorhandensein von vier identischen Exemplaren einer Variante des Trojanischen Pferds „Fenomen Downloader“ in einem Anhang einer gespeicherten Email. Die Listings 5.1 und 5.2 zeigen beispielhaft die in der mitgelieferten ClamAV-Signaturdatenbank enthaltene und die von SigGen aus 22 Mitgliedern der Malware-Familie erzeugte Signatur.

```
1 Trojan.OnLineGames-65:1:*:47616d657320446f776e6c6f616465720000000000446f776e6c6f61
2 64696e672e2e2e000050617573656 42e2e2e000000256925252 06f6620252e31666d6220646f776e
3 6c6f616465642028252e31666d622900000 25693a253032692052656d61
```

Listing 5.1: Signatur aus der mitgelieferten ClamAV-Signaturdatenbank für eine Variante des Trojanischen Pferds „Fenomen Downloader“

Die acht als infiziert erkannten Dateien aus den Dateien der virtuellen Maschine „Büroarbeitsplatz“ sind Dateiarchive, die Dateien enthalten, deren Kompressionsfaktor[6]

[6]Der *Kompressionsfaktor* einer komprimierten Datei bezeichnet den Quotienten aus der Dateigröße vor der Komprimierung geteilt durch die Dateigröße nach der Komprimierung.

```
Worm.SigGen-20080528144600-1631:0:*:5c24*c64424*0000*048d*6860*4100*c64424*5803e
8*0883c40c8d5424*4100*3bc3*0f86*0000*83ec0c8d*8bcc89*c68424b0020000*83ec0c8d*8bc
c89*52c68424bc020000*83ec0c8d*8bcc89*50c68424c8020000*518b8c24*0000*52c68424cc02
0000*8d4c24*535153683f000f00*53535368e4*41006801000080*ffd785c075*4100*a80175*a8
0175*8ad0b9*4100*80ca018815*4100*0000*83c4*807c14*568bf18b4e08*85c974*c746140000
00*8b4c2408*8b4610*03c1894610*4100*53b30184c375*5f32c0*5e83c410c208*518bc18a4c24
03*884810*33c98948148948188948 1c8948248948048848 20*8b442404*568bf18a4c240c*85c0c
7460c00000000*894608*880e75
```

Listing 5.2: Von SigGen erzeugten Signatur für 22 Mitglieder der Malware-Familie „Fenomen Downloader“

ungewöhnlich groß ist. Dies ist eine dokumentierte Funktion des ClamAV-Scanners [10], die sich konfigurieren lässt.

Zusammengefasst finden sich in den Einträgen der Tabelle keine Anhaltspunkte für falsche Positive.

5.3 Falsche Negative

Passend zur Interpretation in Abschnitt 5.2 wird im Kontext dieses Kapitels unter falschen Negativen das Nicht-Erkennen von vorhandener Malware verstanden, wobei die für die Familie des Schadprogramms erzeugte Signatur verwendet wird.
Ein Test auf falsche Negative testet also gewissermaßen die Korrektheit der erzeugten Signaturen. Nach Konstruktion der für die ClamAV-Signaturen verwendeten regulären Ausdrücke in den Abschnitten 3.7 und Schritt 12 aus Abschnitt 3.2.2 ist eine erzeugte Signatur entweder korrekt, oder es wird keine Signatur erzeugt.

Dieses Ergebnis wird dadurch untermauert, dass alle ausführbaren Dateien der Schadprogramme von sieben der zehn Familien, für die Signaturen erzeugt wurden, vom Scanner-ClamAV erkannt wurden. Die Nicht-Erkennung von Malware-Exemplaren der restlichen Familien beruht auf der Tatsache, dass der Scanner von ClamAV nicht in allen Fällen in der Lage ist, ausführbare Dateien vor der Erkennung zu entpacken. Die Entpacker-Komponente des verwendeten Softwarewerkzeug VxClass (siehe 2.6.1) kann hingegen eine viel größere Zahl von ausführbaren Dateien erfolgreich entpacken. Da SigGen auf Basis der vorklassifizierten – und somit bereits entpackten – Disassembler-Datenbanken

arbeitet, können bei der Verwendung der erzeugten Signaturen auf gepackten Dateien falsche Negative auftreten.

5.4 Experimente

In diesem Abschnitt werden zwei Experimente vorgestellt, die das Herausarbeiten von weiteren Eigenschaften des Signaturgenerierungsverfahrens ermöglichen.

5.4.1 Unbekannte Mitglieder einer Malware-Familie

Um die Übertragbarkeit der für eine Malware-Familie erzeugten Signaturen auf noch unbekannte Mitglieder dieser Familie empirisch nachzuweisen, wurden aus zwei Familien mit je 22 und 30 Mitgliedern jeweils vier Schadprogramme ausgewählt und mit SigGen Signaturen für die Auswahlen erzeugt. Anschließend wurden die beiden erzeugten Signaturen mit ClamAV auf den ausführbaren Dateien der jeweiligen Familie getestet. Das Ergebnis dieses Experiments zeigt Tabelle 5.2. Alle Schadprogramme der Familie wurden erkannt.

	Auswahl 1	**Auswahl 2**
Gescannte Dateien	22	30
Gescannte Daten	2,75 MiB	4,25 MiB
Zeit	0,25 s	0,81 s
Als infiziert erkannt	22	30
Signaturlänge	1 876	2 065

Tabelle 5.2: Anwendung einer erzeugten Signatur für vier Schadprogramme auf die gesamte Malware-Familie

Ein Scan des oben erwähnten Rechensystems „*Entwickler*“ führte zu keinen falschen Positiven. Basierend auf der kleinen Testmenge scheinen sich die von SigGen erzeugten Signaturen für die Erkennung noch unbekannter Mitglieder einer Malware-Familie zu eignen.

5.4.2 Einfluss der Kürzungsstrategien

Der Einfluss der verschiedenen Kürzungsstrategien (siehe Abschnitt 3.6) auf die Erkennungsleistung der generierten Signaturen wurde nicht untersucht, da dies den Rahmen dieser Arbeit übersteigen würde.

5.5 Performanz des Verfahrens

Für das Testen in Bezug auf falsche Positive wurden in Abschnitt 5.2 Signaturen für zehn Malware-Familien mit jeweils 5 bis 30 Mitgliedern erzeugt. Die Laufzeiten und die bearbeiteten Datenmengen werden in Tabelle 5.3 gezeigt[7]. Alle Signaturen wurden auf einem Rechensystem mit einem Intel Core2-Duo-Prozessor mit 2,13 GHz und 3,5 GiB Hauptspeicher unter dem Microsoft-Windows-Betriebssystem erzeugt. Die IDA-Pro-Datenbanken und die BinDiff-Ausgaben wurden auf einem eigenen Datenträger gespeichert. Der maximal benötigte Hauptspeicher wurde bei der Erzeugung der Signatur für Familie 8 erreicht, dabei wurden von SigGen etwa 1,3 GiB Hauptspeicher benötigt.

Familie	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Mitglieder	14	14	5	10	5	30	5	12	22	5
Laufzeit (s)	66,1	162,5	22,4	168,6	49,2	484,5	23,7	197,2	365,3	208,4
Eing.-Dateien	1 253	37 534	554	12 404	6 313	8 649	185	32 543	6 938	12 930
Größe (MiB)	72,7	219,9	26,8	208,9	55,3	542,9	22,4	156,1	484,1	25,8
Sig.-Länge	1 189	974	786	1 458	1 262	1 327	1 766	871	1 631	1 216

Tabelle 5.3: Laufzeiten bei der Erzeugung von Signaturen für zehn verschiedene Malware-Familien

Der recht große Speicherbedarf bei der Erzeugung der Signatur von Familie 8 lässt sich durch die dafür notwendige Berechnung der längsten gemeinsamen Teilsequenz von 12 783 Kandidaten-Basic-Blocks aus allen 12 Schadprogrammen der Familie erklären.

Die oben gegebenen Laufzeit-Kennzahlen bestätigen die erhoffte praktische Einsetzbarkeit des Verfahrens auf modernen[8] Rechensystemen.

[7]Da die verschiedenen Antivirenprogramme kein einheitliches Namensschema für Malware verwenden, wurden die einzelnen Familien durchnummeriert.
[8]Stand: 2008

6 Schlussbetrachtungen

Das in dieser Arbeit vorgestellte Verfahren zur automatisierten Generierung von Signaturen für Stämme von Malware und dessen prototypische Implementierung erfüllen die in Kapitel 1 und in Abschnitt 3.1.1 gestellten Anforderungen. Die vollständige Automatisierung des Verfahrens erlaubt eine Reduzierung der Zeitspanne von der Entdeckung eines neuen Schadprogramms bis zur Verteilung einer Signatur für die neu entdeckte Malware, da die zeitaufwändige und fehleranfällige manuelle Analyse entfällt.

In dieser Arbeit wurden in Kapitel 2 zunächst verwendete Begriffe eingeführt und ein Überblick über die Analyse von Malware gegeben und die dafür verwendeten Softwarewerkzeuge näher vorgestellt. Die Beschäftigung mit Signaturbeschreibungssprachen für Antivirenprogramme und der Vergleich mit Signaturen für Programme zur Einbruchs- und Missbrauchserkennung führten zu der Einführung der beiden Kategorien Online- und Offline-Signaturen.
Aufbauend auf den eingeführten Konzepten wurde in Kapitel 3 eine Idee für ein Verfahren zur automatisierten Signaturgenerierung grob skizziert und Anforderungen an eine solche Methode gestellt. Abschnitt 3.2 formalisierte die Verfahrensidee unter Verwendung eines Polynomialzeitalgorithmus für das k-LCS-Problem auf Permutationen, dessen Korrektheit bewiesen und dessen Zeitkomplexität abgeschätzt wurden. Im weiteren Verlauf zeigte sich, dass der verwendete Algorithmus in eine Heuristik für den allgemeinen Fall des Findens gemeinsamer Teilsequenzen überführt werden kann.
Ein weiterer vorgestellter Algorithmus erlaubt zudem die Konstruktion eines regulären Ausdrucks aus einer gemeinsamen Teilsequenz einer Menge von Sequenzen, der alle Sequenzen der Menge akzeptiert. Auch hier wurden einige Eigenschaften formal bewiesen.

Bei der Implementierung des formal beschriebenen Verfahrens enstand – sozusagen als „Nebenprodukt“ – mit IDAJava ein leistungsfähiges Softwarewerkzeug, mit dem sich der Disassembler IDA Pro vollständig automatisisieren lässt und unabhängig von dieser Arbeit nutzbar ist.

Als Ergebnis der prototypischen Implementierung wurde das kommandozeilenorientierte Programm SigGen vorgestellt und dessen Softwarearchitektur näher beschrieben. Es wurde gezeigt, dass die Implementierung von SigGen stark an die formale Beschreibung des Signaturgenerierungsverfahrens angelehnt ist. Trotz Verwendung der objektorientierten Programmiersprache Java ergab sich so eine prozedural ausgerichtete Programmstruktur.

Im Verlauf von Kapitel 5 zeigte sich die Praxistauglichkeit von SigGen in Hinblick auf die Verwendung der erzeugten Signaturen, und es wurde gezeigt, dass der Einsatz des Programms auf üblichen Rechensystemen mit akzeptablen Antwortzeiten möglich ist.
Die Ergebnisse von den weiteren Untersuchungen in Kapitel 5 waren die Übertragbarkeit der generierten Signaturen auf noch unbekannte Mitglieder einer Malware-Familie und die Resistenz des Verfahrens in Bezug auf die Erzeugung von falschen Positiven und falschen Negativen.

Die starke Abhängigkeit von SigGen von den verwendeten Softwarewerkzeugen BinDiff und VxClass lässt eine mögliche zukünftige Integration von SigGen als Komponente von VxClass sinnvoll erscheinen.
Weitere mögliche Verbesserungen des vorgestellten Verfahrens betreffen die bessere Ausnutzung der Sprachmittel der zugrundeliegenden Signaturbeschreibungssprache und die Erzeugung von Signaturen auch für andere Antivirenprogramme als ClamAV.

Eine Verallgemeinerung des Verfahrens zur automatisierten Erzeugung von Online-Signaturen erscheint ebenfalls lohnenswert.

Abkürzungsverzeichnis

API Application Programming Interface

BCM Business Continuity Management

CG Call Graph

CGISO Call Graph Isomorphism

CFG Control Flow Graph

CFGISO Control Flow Graph Isomorphism

ClamAV Clam Antivirus

CRC Cyclic Redundancy Check

CVD ClamAV Virus Database

EDL Event Description Language

ELF Executable and Linking Format

EICAR European Expert Group for IT-Security

FLIRT Fast Library Identification and Recognition Technology

GNU Gnu's Not Unix

GUI Graphical User Interface

IT- informationstechnisch- (Kontext)

IDA Interactive Disassembler

IDB IDA Database

IDC-Script IDA Command Script

IDS Intrusion-Detection System

Java-VM Java Virtual Machine

JDK Java Development Kit

JNI Java Native Interface

JRE Java Runtime Environment

JVM Java Virtual Machine

LCS Longest Common Subsequence

MD5 Message-Digest Algorithm 5

NCD Normalized Compression Distance

PE Portable Executable

PCRE Perl-compatible regular expression

RPC Remote Procedure Call

RMI Remote Method Invocation

SAX Simple API for XML

SDK Software Development Kit

SPP Small Primes Product

SHEDEL Simple Hierarchcal Event Description Language

STL Standard Template Library

SWIG Simplified Wrapper and Interface Generator

UML Unified Modeling Language

VBA Visual Basic for Applications

XML Extensible Markup Language

XML-RPC XML Remote Procedure Call

Abbildungsverzeichnis

2.1 Ein einfaches Modell getarnter Malware 8
2.2 IDA Hauptfenster mit geöffnetem Aufrufgraph 28
2.3 Visuelle Darstellung von Unterschieden zweier Funktionen in BinDiff . . . 32
2.4 Von VxClass generierte Baumdarstellung der Ähnlichkeiten von Malware 38
2.5 Architektur von VxClass 39

3.1 Zwei mögliche Einbettungen einer Teilsequenz 46
3.2 Symbolische Darstellung der Kandidatenfunktionen von sechs verglichenen Schadprogrammen vor dem Einfügen in F_{cand}. 47
3.3 Berechnung des LCS mit dynamischer Programmierung 50
3.4 Ein Algorithmus zur Berechnung von gemeinsamen Teilseqenzen 53
3.5 Konstruktion eines regulären Ausdrucks aus einer gemeinsamen Teilsequenz einer Menge von Sequenzen 59

4.1 Blockdiagramm der von IDAJava bereitgestellten Infrastruktur 65
4.2 UML-Klassendiagramm des IDA-Pro-Plugins von IDAJava 67
4.3 UML-Klassendiagramm der Server- und Automatisierungskomponente von IDAJava 68
4.4 UML-Klassendiagramm einer Auswahl von Klassen in SigGen 70

Tabellenverzeichnis

2.1 Dimensionen, -aspekte und -ausprägungen im Semantikmodell nach Meier 13
2.2 Zieltypen im erweiterten Signaturformat 24
2.3 Offset-Modifikatoren im erweiterten Signaturformat 24
2.4 Überblick über die vorgestellten Signaturbeschreibungssprachen und deren Kategorien 27

5.1 Ergebnisse von kompletten ClamAV-Scandurchläufen 80
5.2 Anwendung einer erzeugten Signatur für vier Schadprogramme auf die gesamte Malware-Familie 82
5.3 Laufzeiten bei der Erzeugung von Signaturen für zehn verschiedene Malware-Familien 83

Listings

2.1 SHEDEL-Signatur für fehlgeschlagene Anmeldeversuche 14
2.2 Spezialisierung einer SHEDEL-Signatur 14
2.3 SHEDEL-Signatur zur Erkennung einer Sequenz von Zeichen 16
2.4 EDL-Signatur zur Erkennung einer Sequenz von Zeichen 17
2.5 MD5-Signatur für die Standard EICAR Anti-Malware-Testdatei [17] 22
2.6 Archiv-Metadaten-Signatur für eine Variante des Bagle-Wurms 23
2.7 Erweiterte Signatur für eine Variante des Trojanischen Pferds Crypted . 23
2.8 IDC-Script für die Ausgabe einer Funktionsliste 29

4.1 Ein Beispiel für die Verwendung von IDAJava 69
4.2 Ausschnitt aus dem Rahmenprogramm 71
4.3 Ausschnitt aus der Methode `generate()` 72
4.4 SigGen Kommandozeilen-Optionen 74

5.1 Signatur aus der mitgelieferten ClamAV-Signaturdatenbank für eine Variante des Trojanischen Pferds „Fenomen Downloader“ 80
5.2 Von SigGen erzeugten Signatur für 22 Mitglieder der Malware-Familie „Fenomen Downloader“ 81

Literaturverzeichnis

[1] AHO, Alfred V. ; CORASICK, Margaret J.: Efficient String Matching: An Aid to Bibliographic Search. In: *Commun. ACM* 18 (1975), Nr. 6, S. 333–340

[2] AYCOCK, John: *Computer Viruses and Malware*. 1st Edition. Verlin : Springer, 2006 (Advances in Information Security). – 227 S. – ISBN 978–0–387–30236–2

[3] BAECHER, Paul ; KOETTER, Markus ; HOLZ, Thorsten ; DORNSEIF, Maximillian ; FREILING, Felix C.: The Nepenthes Platform: An Efficient Approach to Collect Malware. In: ZAMBONI, Diego (Hrsg.) ; KRÜGEL, Christopher (Hrsg.): *RAID* Bd. 4219, Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–39723–X, S. 165–184

[4] BAILEY, Michael ; OBERHEIDE, Jon ; ANDERSEN, Jon ; MAO, Zhuoqing M. ; JAHANIAN, Farnam ; NAZARIO, Jose: Automated Classification and Analysis of Internet Malware. In: KRÜGEL, Christopher (Hrsg.) ; LIPPMANN, Richard (Hrsg.) ; CLARK, Andrew (Hrsg.): *RAID* Bd. 4637, Springer, 2007 (Lecture Notes in Computer Science). – ISBN 978–3–540–74319–4, S. 178–197

[5] BAYER, Ulrich ; KRÜGEL, Christopher ; KIRDA, Engin: TTAnalyze: A Tool for Analyzing Malware. In: *EICAR*, 2006

[6] BEAZLEY, David M.: SWIG: an easy to use tool for integrating scripting languages with C and C++. In: *TCLTK'96: Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996*. Berkeley, CA, USA : USENIX Association, 1996, S. 15–15

[7] BELLARD, Fabrice: QEMU, a Fast and Portable Dynamic Translator. In: *USENIX Annual Technical Conference, FREENIX Track*, USENIX, 2005, S. 41–46

[8] CHEN, Yixin ; WAN, Andrew ; LIU, Wei: A fast parallel algorithm for finding the longest common sequence of multiple biosequences. (2006), December. http://www.pubmedcentral.nih.gov/articlerender.fcgi?artid=1780122

[9] CILIBRASI, Rudi ; VITÁNYI, Paul M. B.: Clustering by compression. In: *IEEE Transactions on Information Theory* 51 (2005), Nr. 4, S. 1523–1545

[10] CLAMAV-PROJEKT: *ClamAV Official FAQ*. http://www.clamav.org/support/faq/. Version: Februar 2007. – [Online; Stand 15. Mai 2008]

[11] CLAMAV-PROJEKT: *About ClamAV*. http://www.clamav.net/about/. Version: Mai 2008. – [Online; Stand 13. Mai 2008]

[12] COHEN, Fred: Computer viruses: theory and experiments. In: *Computers and Security* 6 (1987), Nr. 1, S. 22–35. – ISSN 0167–4048

[13] COMPUTER ECONOMICS INC.: 2007 Malware Report: The Economic Impact of Viruses, Spyware, Adware, Botnets, and Other Malicious Code. Irvine, CA, 2007. – Forschungsbericht

[14] CORPORATE UNIX PRESS: *System V application binary interface (3rd ed.)*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1993. – ISBN 0–13–100439–5

[15] DULLIEN, Thomas: *Persönliche Konversation*. Bochum, Februar 2008

[16] DULLIEN, Thomas ; ROLLES, Rolf: Graph-Based Comparison of Executable Objects. In: *SSTIC '05, Symposium sur la Sécurité des Technologies de l'Information et des Communications*, 2005

[17] EICAR E.V.: *The Anti-Virus or Anti-Malware test file*. http://www.eicar.org/anti_virus_test_file.htm. Version: September 2006. – [Online; Stand 15. Mai 2008]

[18] FLAKE, Halvar: Structural Comparison of Executable Objects. In: FLEGEL, Ulrich (Hrsg.) ; MEIER, Michael (Hrsg.): *DIMVA* Bd. 46, GI, 2004 (LNI). – ISBN 3–88579–375–X, S. 161–173

[19] FRIEDL, Jeffrey E. F. ; ORAM, Andy (Hrsg.): *Mastering Regular Expressions*. 2nd Edition. Sebastopol, CA, USA : O'Reilly & Associates, Inc., 2002. – ISBN 0–596–00289–0

[20] GHEORGHESCU, Marius: An Automated Virus Classification System. In: *Virus Bulletin International Conference (VB2005)*, Virus Bulletin, October 2005, S. 294–300

[21] GOSLING, James ; JOY, William N. ; JR., Guy L. S.: *The Java Language Specification*. Addison-Wesley, 1996. – ISBN 0–201–63451–1

[22] GRYAZNOV, Dmitry ; TELAFICI, Joe: What a waste — The anti-virus industry DOS-ing itself / McAfee Avert Labs, USA. 2007. – Forschungsbericht

[23] HAMMING, Richard W.: Error Detecting and Error Correcting Codes. In: *Bell System Technical Journal* XXVI (1950), April, Nr. 2, S. 147–160

[24] HEX-RAYS SA: *Executive Summary: IDA Pro at the cornerstone of IT security*. http://www.hex-rays.com/idapro/ida-executive.pdf. Version: August 2006. – [Online; Stand 15. Mai 2008]

[25] HEX-RAYS SA: *The IDA Pro Disassembler and Debugger*. http://www.hex-rays.com/idapro/. Version: Mai 2008. – [Online; Stand 15. Mai 2008]

[26] HIRSCHBERG, Daniel S.: A Linear Space Algorithm for Computing Maximal Common Subsequences. In: *Commun. ACM* 18 (1975), Nr. 6, S. 341–343

[27] HUANG, Gaofeng ; LIM, Andrew: An Effective Branch-and-Bound Algorithm to Solve the k-Longest Common Subsequence Problem. In: MÁNTARAS, Ramon L. (Hrsg.) ; SAITTA, Lorenza (Hrsg.): *ECAI*, IOS Press, 2004. – ISBN 1–58603–452–9, S. 191–195

[28] HUNT, James W. ; SZYMANSKI, Thomas G.: A Fast Algorithm for Computing Longest Subsequences. In: *Commun. ACM* 20 (1977), Nr. 5, S. 350–353

[29] HURSEY, Neil J. ; MCEWAN, William A.: *Tree pattern system and method for multiple virus signature recognition*. July 2001. – United States Patent 6980992

[30] JIANG, Tao ; LI, Ming: On the Approximation of Shortest Common Supersequences and Longest Common Subsequences. In: *SIAM J. Comput.* 24 (1995), Nr. 5, S. 1122–1139

[31] KARIM, Md. E. ; WALENSTEIN, Andrew ; LAKHOTIA, Arun ; PARIDA, Laxmi: Malware phylogeny generation using permutations of code. In: *Journal in Computer Virology* 1 (2005), Nr. 1-2, S. 13–23

[32] KASPERSKY, Eugene: *Malware: Von Viren, Würmern, Hackern und Trojanern und wie man sich vor ihnen schützt*. 1. Auflage. Hanser, 2008. – 254 S. – ISBN 3–44641–500–9

[33] KERNIGHAN, Brian W. ; RITCHIE, Dennis: *The C Programming Language*. Prentice-Hall, 1978. – ISBN 0–13–110163–3

[34] KNUTH, Donald E. ; JR., James H. M. ; PRATT, Vaughan R.: Fast Pattern Matching in Strings. In: *SIAM J. Comput.* 6 (1977), Nr. 2, S. 323–350

[35] KOJM, Tomasz: *Creating signatures for ClamAV*. http://www.clamav.net/doc/latest/signatures.pdf. Version: April 2008. – [Online; Stand 03. Mai 2008]

[36] KRAUS, Jürgen: *Selbstreproduktion bei Programmen*. Februar 1980. – Unveröffentlicht —Universität Dortmund, Fachschaft Informatik, S. 94ff.

[37] LAWTON, Kevin P.: Bochs: A Portable PC Emulator for Unix/X. In: *Linux Journal* 29 (1996), September. – ISSN 1075–3583

[38] LEE, Tony ; MODY, Jigar J.: Behavioral classification. In: *EICAR*, 2006

[39] LEE, Tsern-Huei: Generalized Aho-Corasick Algorithm for Signature Based Anti-Virus Applications. In: *ICCCN*, IEEE, 2007. – ISBN 978–1–4244–1251–8, S. 792–797

[40] MAIER, David: The Complexity of Some Problems on Subsequences and Supersequences. In: *J. ACM* 25 (1978), Nr. 2, S. 322–336. – ISSN 0004–5411

[41] MASEK, William J. ; PATERSON, Mike: A Faster Algorithm Computing String Edit Distances. In: *J. Comput. Syst. Sci.* 20 (1980), Nr. 1, S. 18–31

[42] MEIER, Michael: A Model for the Semantics of Attack Signatures in Misuse Detection Systems. In: ZHANG, Kan (Hrsg.) ; ZHENG, Yuliang (Hrsg.): *ISC* Bd. 3225, Springer, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–23208–7, S. 158–169

[43] MEIER, Michael: *Intrusion Detection effektiv! Modellierung und Analyse von Angriffsmustern.* Berlin : Springer, 2007. – ISBN 978–3540482512

[44] MEIER, Michael ; BISCHOF, Niels ; HOLZ, Thomas: SHEDEL-A Simple Hierarchical Event Description Language for Specifying Attack Signatures. In: GHONAIMY, Adeeb (Hrsg.) ; EL-HADIDI, Mahmoud T. (Hrsg.) ; ASLAN, Heba K. (Hrsg.): *SEC* Bd. 214, Kluwer, 2002 (IFIP Conference Proceedings). – ISBN 1–4020–7030–6, S. 559–572

[45] MEIER, Michael ; SCHMERL, Sebastian ; KÖNIG, Hartmut: Improving the Efficiency of Misuse Detection. In: JULISCH, Klaus (Hrsg.) ; KRÜGEL, Christopher (Hrsg.): *DIMVA* Bd. 3548, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 3–540–26613–5, S. 188–205

[46] MYSQL AB: *What is MySQL?* http://dev.mysql.com/doc/refman/5.1/en/what-is-mysql.html. Version: 2008. – [Online; Stand 15. Mai 2008]

[47] NACHENBERG, Carey: *Signature driven cache extension for stream based scanning*. April 2004. – United States Patent 7130981

[48] NATVIG, Kurt: Sandbox technology inside AV scanners. In: *Virus Bulletin International Conference (VB2001)*, Virus Bulletin, September 2001, S. 475–487

[49] PIETREK, Matt: Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. In: *Microsoft Systems Journal* 9 (1994), March, Nr. 3, S. 15 ff.. – ISSN 0889–9932

[50] RIVEST, Ron: The MD5 Message-Digest Algorithm / Internet Engineering Task Force. 1992 (1321). – Internet Request for Comment RFC

[51] ROMAIN, Raboin ; DAVID, Morel: *Enhance ClamAV's signatures using BinDiff*. http://www.stormav.org/publications/enhance_clamavs_signatures.pdf. Version: 2007. – [Online; Stand 15. Mai 2008]

[52] RUTKOWSKA, Joanna: *Introducing Stealth Malware Taxonomy*. http://invisiblethings.org/papers/malware-taxonomy.pdf. Version: November 2006. – [Online; Stand 03. Mai 2008]

[53] SALOMAA, Arto: *Computation and automata*. New York, NY, USA : Cambridge University Press, 1985. – ISBN 0–521–30245–5

[54] SHAPIRO, Jonathan S. ; ADAMS, Jonathan: Design Evolution of the EROS Single-Level Store. In: ELLIS, Carla S. (Hrsg.): *USENIX Annual Technical Conference, General Track*, USENIX, 2002. – ISBN 1–880446–00–6, S. 59–72

[55] SPITZNER, Lance: *Honeypots. Tracking Hackers*. Amsterdam : Addison-Wesley Longman, 2002. – ISBN 978–0321108951

[56] STROUSTRUP, Bjarne: *The C++ Programming Language, Second Edition*. Addison-Wesley, 1991. – ISBN 0–201–53992–6

[57] SUN MICROSYSTEMS, INC.: *VirtualBox – professional, flexible, open*. http://www.virtualbox.org/wiki/VirtualBox. Version: 2008. – [Online; Stand 15. Mai 2008]

[58] SZÖR, Péter ; FERRIE, Peter: Hunting for metamorphic. In: *Virus Bulletin International Conference (VB2001)*, Virus Bulletin, September 2001, S. 123–144

[59] USERLAND SOFTWARE: *XML-RPC Specification*. http://www.xmlrpc.com/spec. Version: 2003. – [Online; Stand 15. Mai 2008]

[60] WAGNER, Robert A. ; FISCHER, Michael J.: The String-to-String Correction Problem. In: *J. ACM* 21 (1974), Nr. 1, S. 168–173

[61] WALTERS, Brian: VMware Virtual Platform. In: *Linux Journal* 63 (1999), July. – ISSN 1075–3583

[62] WEGENER, Ingo: *Effiziente Algorithmen für grundlegende Funktionen (2. Auflage)*. Teubner, 1996

[63] WIECZOREK, Martin ; NAUJOKS, Uwe ; BARTLETT, Bob: *Business Continuity: Notfallplanung für Geschäftsprozesse*. Berlin : Springer, 2002. – ISBN 978–3540442851

[64] WILLEMS, Carsten ; HOLZ, Thorsten ; FREILING, Felix: Toward Automated Dynamic Malware Analysis Using CWSandbox. In: *IEEE Security and Privacy* 5 (2007), Nr. 2, S. 32–39. – ISSN 1540–7993

[65] ZYNAMICS GMBH: *VxClass — Automatic classification of malware and trojans into „families“*. http://www.vxclass.com/. – [Online; Stand 05. Mai 2008]

[66] ZYNAMICS GMBH: *Zynamics BinDiff*. http://www.zynamics.com/index.php?page=bindiff. – [Online; Stand 05. Mai 2008]

Einverständniserklärung des Urhebers

Ich, Christian Blichmann, erkläre mich einverstanden, dass meine Diplomarbeit nach §6 (1) des URG der Öffentlichkeit durch die Übernahme in die Bereichsbibliothek zugänglich gemacht wird. Damit können Leser der Bibliothek die Arbeit einsehen und zu persönlichen wissenschaftlichen Zwecken Kopien aus dieser Arbeit anfertigen. Weitere Urheberrechte werden nicht berührt.

Dortmund, 03.06.2008