Musterbasiertes Filtern von Schadprogrammen und Spam

Inauguraldissertation
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
der Universität Mannheim

vorgelegt von

Philipp Trinius
aus Meißen

Mannheim, 2011

Dekan : Professor Dr. Heinz Jürgen Müller, Universität Mannheim
Referent : Professor Dr. Felix C. Freiling, Friedrich-Alexander-Universität Erlangen-Nürnberg
Korreferent : Professor Dr. Guido Schryen, Universität Regensburg

Tag der mündlichen Prüfung: 21. Oktober 2011

Zusammenfassung

In der vorliegenden Arbeit werden sprachbasierte Filter für Schadprogramme und Spam-Nachrichten vorgestellt. Die Filter basieren auf eigens dafür entwickelten, kontextfreien Grammatiken, anhand derer sich das Verhalten von Programmen und der Aufbau von Spam-Nachrichten beschreiben lassen. Zum Filtern werden die Grammatiken um spezifische Verhaltensmuster beziehungsweise Spam-Inhalte erweitert. Die Evaluierung der Schadprogramm- und Spam-Filter erfolgt an realen Datensätzen und liefert für beide Filtersysteme sehr gute Ergebnisse.

Durch den Einsatz der sprachbasierten Filter für Schadprogramme wird die dynamische Programmanalyse für bekannte Programmfamilien vollständig automatisiert. Das entwickelte Analyse- und Filterungsystem kombiniert dabei die dynamische Sandbox-Analyse mit Methoden des maschinellen Lernens. Die Verbindung der beiden Analysemethoden erfolgt über das *Malware Instruktion Set*. Diese zur Kodierung von Programmverhalten entwickelte Metasprache ist in ihrem Aufbau und ihrer Notation auf eine maschinelle Weiterverarbeitung hin optimiert und trägt entscheidend zu den sehr guten Gesamtergebnissen des Cluster- und Klassifikationssystems bei. Zudem bildet das *Malware Instruktion Set* die Basis der sprachbasierten Verhaltensfilter für Schadprogramme. Durch die Angabe bestimmter Verhaltensmuster in Form von Instruktionsfolgen lassen sich in der Sprache Filter für Programmfamilien definieren. Damit können Programme, allein auf Basis ihres Verhaltens, auf die Zugehörigkeit zu einer Programmfamilie hin untersucht werden. Die durch das *Malware Instruktion Set* vorgegebene Grundgrammatik wird dazu lediglich um die für die Programmfamilien charakteristischen Instruktionsfolgen erweitert. Diese Familien-spezifischen Grammatiken lassen sich anschließend zum Filtern der Verhaltensreporte heranziehen.

In dem proaktiven Filtersystem für Spam-Nachrichten werden ausgewählte Spambots in einer *Sandnet* Umgebung ausgeführt. Die während der Analysephase durch die Bots verschickten Spam-Nachrichten werden auf einen lokalen Server umgeleitet und anschließend an einen Filtergenerator übergeben. Dieser analysiert die Nachrichten und ordnet sie den Spam-Kampagnen zu. Die Zuordnung erfolgt anhand des Nachrichteninhalts, da dieser innerhalb einer Kampagne relativ konstant bleiben muss. Für jede zugeordnete Spam-Nachricht wird die entwickelte kontextfreie Grundgrammatik für E-Mail-Nachrichten um die in der Nachricht enthaltenen Daten (Wörter, Sätze, URLs, etc.) erweitert. Das Resultat ist eine Grammatik, die zum Filtern von Nachrichten aus einer spezifischen Spam-Kampagne verwendet werden kann. Die Filterergebnisse dieses Ansatzes sind sehr gut: Teilweise erreichen aus einer einzelnen Nachricht erstellte Filter bereits Erkennungsraten von 99 Prozent.

Danksagung

An dieser Stelle möchte ich mich bei all denen bedanken, die mich während meiner Promotion unterstützt haben.

Mein Dank gilt ganz besonders Professor Dr. Felix C. Freiling, der es mir ermöglicht hat, meine Promotion bei ihm abzulegen. Seine umfassende, von zahlreichen Gesprächen und zielführenden Anregungen gekennzeichnete Betreuung, war mir eine wertvolle Unterstützung.

Meinem Zweitgutachter Professor Dr. Guido Schryen danke ich für seine Kommentare und Anmerkungen zu dem Thema und meiner Arbeit. Auch seine Rückmeldungen haben die Arbeit in die vorliegende Form gebracht.

Besonders danke ich Jan Göbel, Thorsten Holz, Carsten Willems und Konrad Rieck. Die gemeinsam mit ihnen erstellten Veröffentlichungen und in diesem Rahmen geführten Diskussionen bilden die Grundlage dieser Arbeit. Insbesondere Jan Göbel hat mich, auch über die gemeinsamen Veröffentlichungen hinaus, mit seinen Fragen, Anmerkungen und Ideen stets unterstützt. Außerdem danke ich Andreas Pitsillidis für die Bereitstellung seiner Spam-Datensätze.

Michael Becher, Andreas Dewald, Markus Engelberth, Christian Gorecki, Jan Göbel, Thorsten Holz, Ralf Hund, Carsten Willems und allen anderen Mitarbeitern des Lehrstuhl für Praktische Informatik 1 der Universität Mannheim danke ich für die freundschaftliche Atmosphäre und die fruchtbaren Diskussionen, von denen die Zusammenarbeit geprägt war. Des Weiteren danke ich allen Studenten, die ich betreuen durfte und die ihren Teil zu dieser Arbeit beigetragen haben. Martin Gräßlin, Matthias Luft und Ben Stock sind unter diesen hervorzuheben, da von ihnen implementierte Werkzeuge und Systeme in dieser Arbeit Anwendung finden.

Meinen Korrekturlesern Konrad Trinius und Vanessa Krohn danke ich für ihren Blick für eingeschlichene Fehler. Ihre Anmerkungen und Fragen haben mir zudem geholfen, die Inhalte verständlicher darzustellen.

Bedanken möchte ich mich auch bei meiner Familie, ohne deren Rückhalt und Zutrauen ich diese Arbeit wohl weder begonnen, noch abgeschlossen hätte.

Inhaltsverzeichnis

1 Einführung **1**
- 1.1 Einführung . . . 1
- 1.2 Motivation . . . 2
- 1.3 Ergebnisse . . . 3
  - 1.3.1 Verhaltensanalyse und -filterung von Schadprogrammen . . . 3
  - 1.3.2 Sprachbasierte Filter für Spam-Nachrichten . . . 4
- 1.4 Gliederung . . . 5
- 1.5 Kooperation und Eigenleistung . . . 6
- 1.6 Publikationen . . . 7

2 Grundlagen **9**
- 2.1 Schadprogramme . . . 9
  - 2.1.1 Klassifikation von Schadprogrammen . . . 11
  - 2.1.2 Analyse von Schadprogrammen . . . 15
- 2.2 Spam . . . 21
  - 2.2.1 Ausgewählte Spam-Techniken . . . 23
  - 2.2.2 Technische Spam-Filter . . . 26
  - 2.2.3 Aktuelle Spambots . . . 28
- 2.3 Maschinelles Lernen . . . 29
  - 2.3.1 Clusteranalyse . . . 30
  - 2.3.2 Klassifikation . . . 34
- 2.4 Formale Sprachen . . . 37
- 2.5 Lex und Yacc . . . 41
  - 2.5.1 Lexikalische Analyse . . . 42
  - 2.5.2 Syntaktische Analyse . . . 42
  - 2.5.3 Ableiten mit PLY . . . 43
- 2.6 Zusammenfassung . . . 45

3 Musterbasierte Filter für Schadprogramme **47**
- 3.1 Verwandte Arbeiten . . . 49
- 3.2 Das *Malware Instruction Set* - MIST . . . 52
  - 3.2.1 Verhaltensrepräsentation . . . 53
  - 3.2.2 Beispiele . . . 55
  - 3.2.3 Empirische Bewertung von MIST . . . 61
  - 3.2.4 Zusammenfassung . . . 64

3.3 Automatische Analyse von Schadprogrammen mit MALHEUR und MIST 64
3.3.1 Systementwurf . . . 65
3.3.2 Referenzdatensatz . . . 66
3.3.3 Vergleich mit konkurrierenden Systemen . . . 68
3.3.4 Zusammenfassung . . . 70
3.4 Musterbasiertes Filtern von Schadprogrammen mit MIST . . . 71
3.4.1 Eine kontextfreie Grammatik für MIST . . . 71
3.4.2 Auf LEX und YACC basierende MIST-Filter . . . 75
3.4.3 Auswertung der MIST-Filter . . . 85
3.4.4 Zusammenfassung . . . 89
3.5 Zusammenfassung und Ausblick . . . 90

4 Musterbasierte Filter für Spam-Nachrichten 93
4.1 Verwandte Arbeiten . . . 95
4.2 Aufbau des Filtersystems . . . 96
4.2.1 Die Monitorumgebung – das *Sandnet* . . . 97
4.2.2 Filtergenerator . . . 99
4.2.3 Filterplugin für E-Mail-Clients . . . 99
4.3 Filtern mit regulären Ausdrücken . . . 100
4.3.1 Filter-Generierung . . . 101
4.3.2 Auswertung der Filter mit regulären Ausdrücken . . . 105
4.4 Kontextfreie Grammatiken für E-Mail-Nachrichten . . . 111
4.4.1 Grundgrammatik für E-Mail-Nachrichten . . . 114
4.4.2 Erweiterte Grundgrammatik für E-Mail-Nachrichten. . . . 115
4.5 Lernen von Spam-Vorlagen mit LEX und YACC . . . 120
4.5.1 Lernen neuer Token . . . 122
4.5.2 Lernen neuer Sätze . . . 123
4.5.3 Lernen von Modifikationen in bekannten Sätzen . . . 124
4.5.4 Fehlerbehandlungsroutine . . . 126
4.5.5 Zuordnung der Nachrichten zu Spam-Kampagnen . . . 129
4.6 Filtern mit LEX und YACC . . . 130
4.6.1 Auswertung der *kontextfreien* Spam-Filter . . . 130
4.7 Zusammenfassung und Ausblick . . . 138

5 Zusammenfassung und Ausblick 141
5.1 Zusammenfassung . . . 141
5.1.1 Musterbasierte Filter für Schadprogramme . . . 141
5.1.2 Musterbasierte Filter für Spam-Nachrichten . . . 142
5.2 Ausblick . . . 144
5.2.1 Optimierung und Anpassung der Implementierung . . . 144
5.2.2 Programmanalyse auf Prozess- und Thread-Ebene . . . 144
5.2.3 *Multipath*-Unterstützung für MIST . . . 145
5.2.4 Erweiterte E-Mail-Grammatik . . . 145

Literaturverzeichnis 147

A PLY 157

B Antiviren-Scanner **165**

C MIST-Lexer, -Parser und -Filter **169**

D Filtern mit regulären Ausdrücken **177**

E Filtern mit kontextfreien Grammatiken **189**

Tabellenverzeichnis

2.1 Die zentralen Eigenschaften der 10 aktivsten Spambots. 29
2.2 Ableitung eines Satzes mit PLY. 44

3.1 Die einzelnen MIST-Kategorien inklusive Codierungen. 54
3.2 Codierung des `Flags`-Attributs in MIST. 59
3.3 Zur Klassifizierung verwendete Virenscanner. 67
3.4 Zusammensetzung des Referenzdatensatzes. 67
3.5 Vergleich der Analysemethoden auf dem Referenzdatensatz. 68
3.6 Die Ergebnissen von MALHEUR auf den Referenzdaten. 69
3.7 Der Aufbau der auf AV-Labels trainierten MIST-Filter. 85
3.8 Ergebnisse der auf den AV-Labels trainierten MIST-Filter. 86
3.9 Details zu den auf den MALHEUR-Daten trainierten MIST-Filter. . . . 87
3.10 Ergebnisse der auf den MALHEUR-Daten trainierten MIST-Filter. . . . 88

4.1 Erkennungsraten der *regex*-basierten Spam-Filter für STORM. 107
4.2 Erkennungsraten der *regex*-basierten Spam-Filter für GHEG. 107
4.3 Ergbnisse der einzelnen *regex*-basierten Filter für SRIZBI. 108
4.4 Erkennungsraten der *regex*-basierten Spam-Filter für SRIZBI. 110
4.5 Token der erweiterten Grundgrammatik. 117
4.6 Zur Evaluierung herangezogene Spam-Läufe. 130
4.7 Analyseläufe zur Filtergenerierung. 131
4.8 Filterergebnisse für das STORM-Botnetz. 131
4.9 Gelernte Wörter und Sätze für die STORM-Spam-Läufe. 132
4.10 Filterergebnisse auf den GHEG-Spam-Daten. 133
4.11 Gelernte Wörter und Sätze für den GHEG-Spam-Lauf. 133
4.12 Filterergebnisse auf den SRIZBI-Spam-Daten. 134
4.13 Erkennungsraten der SRIZBI-Filter. 135
4.14 Basis der einzelnen SRIZBI-Filter. 136
4.15 Gelernte Wörter und Sätze für den SRIZBI Spam-Lauf. 137

B.1 Performanz der Anti-Virus Lösungen auf einem Kandidatendatensatz. 166
B.2 Performanz der Anti-Virus Lösungen auf dem Referenzdatensatz. . . 167

D.1 Vollständige Filterergebnisse des STOM-Botnetz. 177
D.2 Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung). 178
D.3 Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung). 179

D.4 Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung). 180
D.5 Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung). 181
D.6 Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung). 182
D.7 Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung). 183
D.8 Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung). 184
D.9 Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung). 185
D.10 Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung). 186
D.11 Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung). 187
D.12 Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung). 188

E.1 Ergebnisse für das GHEG-Filter basierend auf 3 Nachrichten. 189
E.2 Ergebnisse für das GHEG-Filter basierend auf 26 Nachrichten. 190
E.3 Ergebnisse für das GHEG-Filter basierend auf 256 Nachrichten. 190
E.4 Ergebnisse für das GHEG-Filter basierend auf 2.556 Nachrichten. . . . 191
E.5 Ergebnisse für das GHEG-Filter basierend auf 25.459 Nachrichten. . . 191
E.6 Ergebnisse für das GHEG-Filter basierend auf 254.591 Nachrichten. . 191
E.7 Erkennungsraten der SRIZBI-Filter. 192

Abbildungsverzeichnis

2.1 Übersicht über die Infektionsraten nach Ländern/Regionen. 10
2.2 Die Anzahl jährlich entwickelter Virensignaturen. 11
2.3 Schwachstellenanzahl in Produkten von Microsoft und Drittanbietern. 13
2.4 Klassischer Aufbau eines zentralisierten Botnetzes. 14
2.5 Aufbau des Analysesystems mit Wiederherstellungsmechanismus. . . 18
2.6 Aufbau der im Analysesystem verwendeten Festplatten. 18
2.7 Ablauf des API-Hooking. 19
2.8 Entwicklung des Spam-Volumens. 22
2.9 Weltweite Verteilung der Spam-Quellen. 23
2.10 Das auf einem *SOCKS-Proxy* basierte Spam-Verfahren. 24
2.11 Das musterbasierte Spam-Verfahren. 25
2.12 Spam-Anteile der zehn aktivsten Botnetze. 28
2.13 Spam-Nachrichten-Ausstoß des RUSTOCK-Botnetz im September 2010. 29
2.14 Beispiele für Clusterings von Datenpunkten. 31
2.15 Beispiele für dichtebasierte Cluster. 32
2.16 Single-Link-Dendrogramm. 33
2.17 Die Konfusionsmatrix für ein Drei-Klassen-Problem. 35
2.18 Die Grammatik einer einfachen natürlichen Sprache. 38
2.19 Syntaxbaum zur Ableitung eines Satzes. 38
2.20 Grammatik-Typen nach Chromsky. 39
2.21 Die BNF einer einfachen natürlichen Sprache. 41

3.1 Schematischer Aufbau einer MIST-Instruktion. 53
3.2 Repräsentation des Systemaufrufs `load_dll`. 56
3.3 Repräsentation des Systemaufrufs `connect_socket`. 58
3.4 Repräsentation des Systemaufrufs `move_file`. 59
3.5 Aufbau der Infozeile in MIST-Reporte. 61
3.6 Vergleich ausgewählter Notationen zur Verhaltensrepräsentation. . . . 63
3.7 Vergleich der Größe von Verhaltensinstrukionen und -reporte. 64
3.8 Schematischer Überblick über das Analysesystem. 65

4.1 Systemaufbau zur Analyse von Spambots. 97
4.2 Beispiel für musterbasierte Spam-Nachrichten. 100
4.3 Gegenüberstellung von zwei E-Mail-Headern einer Spam-Kampagne. 105
4.4 Permutationen in Spam-Kampagnen. 112

4.5 HTML-Formatierungen in Spam. 113
4.6 Schematischer Ablauf des Lernens von Mustern. 121
4.7 Ableiten und Lernen eines unbekannten Wortes. 122
4.8 Modifikation von Sätzen. 125
4.9 Schematischer Ablauf der Fehlerbehandlungsroutine. 127

C.1 Datawarehouse zur Speicherung der n-Gramme. 170

Listingsverzeichnis

2.1 Beispiel einer infizierten `autorun.inf` Datei. 12
2.2 Der Aufbau eines CWSandbox-Reports. 20
2.3 Definition der Tokenliste und der einzelnen Token in PLY. 42
2.4 Definition von Grammatikregeln in PLY. 43

3.1 Auszüge aus dem Level 2 MIST-Report eines Schadprogramms. . . . 60
3.2 Charakteristische 2-Gramme der VIRUT-Familie. 72
3.3 Ein regulärer Ausdruck zum Filtern auf fünf überlappende 2-Gramme. 73
3.4 Ein optimierter regulärer Ausdruck für fünf überlappende 2-Gramme. 74
3.5 Die kontextfreie Grammatik für MIST. 74
3.6 Die PYTHON-Klasse *MISTLexer*. 76
3.7 Die PYTHON-Klasse *MISTParser*. 77
3.8 Die für die VIRUT-Familie abgeleitete PYTHON-Klasse *VIRUTLexer*. 79
3.9 Erweiterungen der PYTHON-Klasse *VIRUTParser* zum Ableiten von 2-Grammen. 80
3.10 Erweiterungen des *VIRUTParser* zum Ableiten von überlappenden 2-Grammen. 80
3.11 Die Klassen-Vorlage MALWARELEXER. 82
3.12 Die Klassen-Vorlage *MALWAREParser*. 83
3.13 Die Skript-Vorlage *MALWAREFilter*. 84

4.1 Ein *regex* Filter für eine Spam-Kampagne des BOBAX-Botnetzes. . . 104
4.2 Die Token der Grundgrammatik für E-Mail-Nachrichten. 114
4.3 Die Ableitungsregeln der Grundgrammatik für E-Mail-Nachrichten. . 115
4.4 Erste Anpassung der Grundgrammatik für E-Mail-Nachrichten. . . . 116
4.5 Zweite Anpassung der Grundgrammatik für E-Mail-Nachrichten. . . . 117
4.6 Die Regeln der erweiterten Grundgrammatik. 119
4.7 Die Grammatikregel und Funktion zum Ableiten einer gelernten URL. 123
4.8 Die Grammatikregel und Funktion zum Ableiten eines gelernten Satzes. 123
4.9 Die Grammatikregel und Funktion zum Ableiten eines gelernten Wortes. 130

A.1 Der Parser für eine einfache natürliche Grammatik. 157
A.2 Der Parser für eine einfache natürliche Grammatik (Fortsetzung). . . . 158
A.3 Patch für die Klasse *yacc.py* des PLY-Modules. 161

C.1 Die PYTHON-Klasse *MISTLexer*. 171

C.2 Die PYTHON-Klasse *MISTParser*. 172
C.3 Die PYTHON-Klasse *MISTParser* (Fortsetzung). 173
C.4 Das PYTHON-Skript *MISTLexYacc_ParserGenerator*. 174
C.5 Das PYTHON-Skript *MISTLexYacc_ParserGenerator* (Fortsetzung). . 175

KAPITEL I

Einführung

1.1 Einführung

Informationstechnologie (IT) und Internet haben in kürzester Zeit eine entscheidende Rolle in Wirtschaft und Gesellschaft der Industrienationen eingenommen. Sowohl die computergestützten Design- und Fertigungsprozesse als auch die Verwaltung und Kommunikation der Unternehmen hängen unmittelbar von IT-Systemen ab. Somit haben IT-Systeme zentrale Bedeutung für den wirtschaftlichen Erfolg der Unternehmen erlangt und müssen deshalb besonders gegen Ausfälle und Angriffe abgesichert werden.

Mit der steigenden Vernetzung der IT-Systeme verschwinden zunehmend physische Netzgrenzen in Unternehmen, und selbst Systeme mit sensitiven Daten oder Funktionen werden – wenigstens indirekt – „unternehmensweit“ erreichbar. Dadurch kann selbst die Infektion eines als unkritisch eingestuften Systems gravierende Auswirkungen haben, weil sich Angreifer oder Schadprogramme im internen Netz befinden, die damit Zugriff auf kritische Systeme und Daten erlangen können. Das durch die Vernetzung entstehende Problem lässt sich deutlich an dem Infektionsweg des STUXNET Wurms nachvollziehen [Falliere et al., Februar 2011]: nachdem Mitarbeiter infizierte USB-Sticks an ihren Arbeitsplatzrechnern anschlossen, nutzte der Wurm Schwachstellen in Windows Systemen aus und infizierte die Rechner. Im internen Netz suchte der Wurm nach weiteren Systemen und infizierte neben weiteren Windows Systemen insbesondere Siemens-Steueranlagen. Erst auf einem bestimmten Anlagentyp und bei einer ausgewählten Konfiguration führte der Wurm die eigentliche Schadfunktion aus und manipulierte die Steuerung der zur Anreicherung von Uran verwendeten Zentrifugen.

Die von dem Wurm ausgenutzten Schwachstellen lassen sich in drei Kategorien einteilen: *Social Engineering*, Programmierfehler, Architekturfehler. Während sich der Fehler in der Netzarchitektur – zum Beispiel durch eine physische Trennung der Netze – vermeiden ließ, sind sowohl das Fehlverhalten von Menschen, als auch die von diesen beim Programmieren gemachten Fehler nie ganz auszuschließen. Eine hundertprozentige Sicherheit gibt es somit nicht. Solange Angreifer existieren, die versuchen

diese Schwachstellen zu ihrem eigenen Vorteil auszunutzen, sind präventive, als auch reaktive Schutzmaßnahmen notwendig, um die Angriffe zu verhindern oder deren Auswirkungen so weit wie möglich zu begrenzen.

1.2 Motivation

Heute kommen bei den meisten Angriffen auf IT-Systeme Schadprogramme zum Einsatz. Die Bandbreite reicht dabei von kleinen Skripten, die das Ausnutzen einer Schwachstelle automatisieren, bis hin zu komplexen Programmen, die sich automatisiert im Netz verbreiten und die kompromittierten Systeme zu einem Netzwerk – sogenannte Botnetze – zusammenschalten. Dieses Netzwerk kann zentral durch den Angreifer gesteuert werden. Schadprogramme besitzen eine Vielzahl verschiedener Verbreitungswege und können auf einem befallenen System – mit entsprechenden Rechten – nahezu jede Funktion ausführen. Damit stellen sie die größte Gefahr für Computersysteme dar und lassen sich zudem nur schwer bekämpfen. Insbesondere für Schadprogrammen mit unbekannten Funktionen oder Verbreitungsvektoren, ist eine klare Abgrenzung zu „normalen“ Programmen nur bedingt möglich.

Antiviren-Programme identifizieren Schadprogramme in der Regel anhand von Codesignaturen. Diese werden für bereits bekannte Schadprogramme oder -funktionen generiert, und alle neu auf Systeme geladenen oder zur Ausführung ausgewählten Programme werden auf die Signaturen hin untersucht. Um ein Schadprogramm erkennen zu können, muss somit immer erst eine passende Signatur für das Programm oder die Programmfamilie existieren. Das heißt, es muss mindestens ein ähnliches Schadprogramm bekannt und analysiert sein. Das größte Problem stellen hierbei die Vielfalt und Menge der Schadprogramme dar. Täglich werden tausende neue Schadprogramme entdeckt, für die geeignete Signaturen entwickelt oder bestehende Signaturen angepasst werden müssen [Microsoft, 2009, Symantec, 2009]. Selbst wenn es sich bei vielen der Programme nur um leichte Variationen bekannter Schadprogrammfamilien handelt, müssen diese erst durch eine Analyse erkannt werden. Um zeitnah zu fundierten Aussagen über die neuen Schadprogramme zu kommen und damit den Ressourceneinsatz für eine detaillierte Analyse richtig zu steuern, sind Analysewerkzeuge und -systeme notwendig, die mit keiner oder nur minimaler Anwenderinteraktion bekannte und unbekannte Programme voneinander abgrenzen. Einen vielversprechenden Ansatz stellt hier die Kombination der dynamischen Verhaltensanalyse mit Methoden des maschinellen Lernens dar. Über die Klassifikation von Verhaltensreporten können Programme bekannten Familien zugeordnet werden. Für die übrigen, nicht klassifizierbaren Verhaltensreporte kann die Clusteranalyse erste Informationen über enthaltenen Programmfamilien liefern. Der hohe Automatisierungsgrad der dynamischen Analyse ermöglicht, ein entsprechendes Analysewerkzeug auch bei sehr großen Mengen neuer Schadprogramme anzuwenden.

Mit rund 90 Prozent aller verschickten E-Mail-Nachrichten stellt *Spam* die zweite große Herausforderung im Internet dar. Eine von MCAFEE und ICT International veröffentlichte Studie schätzt die jährlich für das Weiterleiten, Zustellen und (manuelle) Filtern der Spam-Nachrichten verbrauchte Energie auf 33 Milliarden Kilowattstunden [McAfee, 2009]. Dies entspricht dem jährlichen Stromverbrauch von

2.4 Millionen Haushalten in den USA. Spam verursacht damit einen hohen materiellen und ökologischen Schaden. Zudem stellen Spam-Nachrichten zunehmend ein Sicherheitsrisiko dar. Mit der fortschreitenden Härtung von Betriebssystemen, werden Spam-Nachrichten als Verbreitungsvektor für Schadprogramme und -dokumente immer wichtiger. Die Schadprogramme lassen sich zum Beispiel als Dateianhang direkt in der Nachricht verteilen, oder verbergen sich auf verlinkten Internetseiten.

Auch wenn die meisten Spam-Nachrichten von aktuellen Filtern zuverlässig erkannt werden, existieren doch keine hundertprozentigen Filter. Die bei allen Filtern vorkommenden *false postives*, das heißt normale Nachrichten werden fälschlicherweise als Spam klassifiziert, haben zur Folge, dass die meisten Filter die Nachrichten nur markieren oder in einen Spam-Ordner verschieben. Für das Löschen der Nachrichten bleibt weiterhin der Anwender verantwortlich. Ein weiteres Problem aktueller Spam-Filter besteht darin, dass Filter – ähnlich den Signaturen für Schadprogramme – nur für die Nachrichten erstellt werden können, die auf ausreichend vielen Systemen empfangen wurden. Jede Spam-Kampagne hat damit zumindest kurzfristig Erfolg. Das heißt, es werden Spam-Nachrichten zugestellt, bevor ein passender Filter erstellt werden kann.

Der überwiegende Anteil an Spam wird aktuell über Botnetze versendet, die man analysieren kann. Mit dem sich so bietenden Einblick lässt sich ein proaktives Analyse- und Filtersystem für Spambots entwickeln, in dem sehr spezifische Filter für einzelne Spam-Kampagnen generiert werden können. Da tausende Spam-Nachrichten einer Kampagne direkt an der Quelle abgegriffen werden, lassen sich in dem System sehr gute Filter zum frühestmöglichen Zeitpunkt generieren. Als Ansatzpunkt für die Filter fungieren die von den meisten Botnetzen eingesetzten musterbasierten Spam-Algorithmen. Die Filter approximieren das Muster einer Kampagne und passen deshalb ausschließlich auf Nachrichten einer Kampagne. Damit ist sichergestellt, dass keine *false positives* existieren und die als Spam klassifizierten Nachrichten können sofort gelöscht werden.

1.3 Ergebnisse

In dieser Arbeit werden zwei Filtersysteme für Schadprogramme und Spam-Nachrichten vorgestellt. Beide Systeme nutzen einen sprachbasierten Ansatz zum Filtern und wurden in Prototypen umgesetzt. Die Effektivität des neuartigen Filteransatzes konnte sowohl für Schadprogramme, als auch für Spam-Nachrichten in umfangreichen Evaluierungen auf realen Datensätzen nachgewiesen werden.

1.3.1 Verhaltensanalyse und -filterung von Schadprogrammen

Das in dieser Arbeit präsentierte Analysesystem ermöglicht es, Schadprogramme entsprechend ihrem tatsächlichen Verhalten in Programmfamilien einzuteilen und für diese Familien Prototypen sowie charakteristische Verhaltensmuster und -filter zu berechnen. Durch die Kombination der dynamische Sandbox-Analyse mit Methoden des maschinellen Lernens können Schadprogramme vollständig automatisiert analysiert und kategorisiert werden. Die dynamische Programmanalyse erfolgt mit Hilfe der CWSANDBOX und zur Analyse der Verhaltensreporte wird mit MALHEUR ein

eigens für das Analysesystem entwickeltes Cluster- und Klassifikationswerkzeug eingesetzt. Die Schnittstelle zwischen der dynamischen Analyse und der Bewertung des Verhaltens stellt das *Malware Instruction Set*, kurz MIST, dar. MIST ist die erste Metasprache zur Kodierung von Programmverhalten, deren Aufbau und Notation zudem auf eine maschinelle Weiterverarbeitung der Verhaltensreporte hin optimiert sind. Bei der Transformation der sehr detaillierten, XML-codierten Verhaltensreporte in die MIST-Repräsentation, werden die Verhaltensreporte auf die zentralen und aussagekräftigen Informationen reduziert. Neben sämtlichen Formatierungsinformationen werden zum Beispiel von der Anaylseumgebung abhängige Informationen und zufällige Daten aus den Reporten entfernt. Eine Neuordnung aller Verhaltensinformationen erlaubt außerdem die Definition unterschiedlicher Detailstufen innerhalb der Reporte, auf denen diese miteinander verglichen werden können.

Insgesamt konnten durch die in MIST umgesetzten Optimierungen die Cluster- und Klassifikationsergebnisse des gesamten Analysesystems entscheidend gesteigert werden. Die optimierte Notation reduzierte zudem den Speicherbedarf pro Verhaltensreport, was sich positiv auf die sehr rechen- und speicherintensiven Verfahren des maschinellen Lernens auswirkt. Das Analysesystem wurde sowohl auf künstlichen als auch realen Datensätzen evaluiert. Es liefert durchweg bessere Ergebnisse, als konkurrierende Systeme und wurde über 13 Monate erfolgreich in einem Produktivsystem eingesetzt.

Mit der Einführung einer eigenen Sprache zur Beschreibung von Programmverhalten wird es möglich, die charakteristischen Verhaltensmuster von Programmfamilien verständlich darzustellen und auch zum Filtern der Verhaltensreporte zu nutzen. Der dafür entwickelte Filtermechanismus drückt das charakteristische Verhalten einer Familie in einer Familien-spezifischen Grammatik aus. Jedes Programm, dessen Verhaltensreport dieser Grammatik genügt, kann der Programmfamilie zugeordnet werden. Zur Klassifikation neuer Verhaltensreporte muss deshalb nicht mehr auf MALHEUR zurückgegriffen werden. Stattdessen werden die Reporte direkt durch entsprechende Verhaltensfilter bekannten Schadprogrammfamilien zugeordnet.

1.3.2 Sprachbasierte Filter für Spam-Nachrichten

Mit dem in dieser Arbeit präsentierten proaktiven Filtersystem für Spam-Nachrichten können sehr schnell Spam-Filter für musterbasierte Spam-Kampagnen berechnet werden, die zudem über ausgezeichnete Erkennungsraten verfügen. Dazu werden die zu analysierenden Spambots in einer *Sandnet* Umgebung ausgeführt und überwacht. Alle während der Analysephase von dem Bot verschickten Spam-Nachrichten werden innerhalb des Sandnet abgegriffen und einem Filtergenerator übergeben. Dieser analysiert die Nachrichten und ordnet sie den verschiedenen Kampagnen zu. Für jede dieser Kampagnen wird dabei das von dem Spambot verwendete Muster berechnet und in einen Spam-Filter für die einzelnen Kampagnen überführt.

Für den Aufbau der Spam-Filter wurden zwei sprachbasierte Ansätze entworfen. Der erste Ansatz basiert auf regulären Ausdrücken und wurde zusammen mit dem proaktiven Filteransatz entwickelt und veröffentlicht. Bei diesem Ansatz wird das Muster einer Kampagne durch die Überlagerung mehrere Spam-Nachrichten berechnet und in einen regulären Ausdruck überführt. Der Ausdruck enthält diejenigen Text-

blöcke, die in allen Nachrichten der Kampagne identisch sind. Nur diese können auch in dem verwendeten Kampagnen-Muster enthalten sein. Die übrigen Textblöcke werden durch Pattern ersetzt, die den an diesen Stellen in den Spam-Nachrichten enthaltenen, variablen Text möglichst genau beschreiben. Der Filter wurde anhand realer Spam-Daten verschiedener Spambots evaluiert und dabei gute bis sehr gute Erkennungsquoten erzielt.

Mit dem auf einer kontextfreien Grammatik für E-Mail-Nachrichten basierenden Filteransatz wird ein zweiter, bisher unveröffentlichter Filteransatz präsentiert. Dabei wird eine eigens entwickelte Grammatik für E-Mail-Nachrichten zugrunde gelegt, die für jede Kampagne um die in den E-Mail-Nachrichten der Kampagne enthaltenen Daten (Wörter, Sätze, URLs, etc.) erweitert wird. Das Resultat ist eine Kampagnenspezifische Grammatik, für die jeweils ein auf LEX und YACC basierender Parser generiert wird. Mit diesem lassen sich neue Spam Nachrichten auf ihre Kampagnen-Zugehörigkeit hin untersuchen. Die Grundannahme für diesen Filteransatz ist, dass dem *Spammer* nur eine begrenzte Anzahl an Wörtern und Sätzen zur Verfügung steht, um sein Produkt zu bewerben. Diese Annahme wird durch hervorragende Filterergebnisse bei der Analyse mehrere Spam-Kampagnen aus unterschiedlichen Botnetzen bestätigt. Aus einer einzelnen Nachricht erstellte Filter erreichen durch die weniger stringente Beschreibung der Kampagnen bereits Erkennungsrate von teilweise über 99 Prozent.

1.4 Gliederung

Die vorliegende Arbeit ist in fünf Kapitel untergliedert, von denen Kapitel 3 und 4 den Schwerpunkt bilden. Diese beschäftigen sich mit der Analyse und Filterung von Schadprogrammen und Spam. Das Ziel der Analyse besteht darin, geeignete Filter zu entwickeln, die es ermöglichen einzelne Schadprogramme ihren Programmfamilien und E-Mail-Nachrichten den zugehörigen Spam-Kampagnen zuzuordnen. Dazu wurden in beiden Fällen Filter entwickelt, die auf kontextfreien Grammatiken basieren. Diese identische Filtermethodik und die Symbiose, die Schadprogramme und Spam bilden, stellen die Verbindung zwischen den beiden Kapiteln dar. Spam wird überwiegend durch Schadprogramme von infizierten Rechnern aus verbreitet und Schadprogramme nutzen ihrerseits Spam als Verbreitungsvektor, indem sie zum Beispiel eine Kopie von sich im Anhang verschicken. Mit den im vierten Kapitel betrachteten Spam-Filtern werden zudem nur aus Botnetzen – also von Schadprogrammen – verschickte Nachrichten gefiltert.

Im Folgenden werden die in den einzelnen Kapiteln präsentierten Inhalte kurz zusammengefasst:

Im Grundlagenkapitel (Kapitel 2) werden alle zum Verständnis der Arbeit notwendigen Grundlagen geliefert. Den Einstieg bildet eine Beschreibung der verschiedenen Schadprogramme und der Gefahren, die von diesen ausgehen. Anschließend werden die statische und dynamische Programmanalyse vorgestellt. Der Schwerpunkt liegt dabei auf der verhaltensorientierten Analyse von Programmen mit Hilfe der *CWSandbox*. Diese generiert die Verhaltensreporte, auf denen das im dritten Kapitel präsentierte Analysesystem aufbaut. In Abschnitt 2.2 wird eine Definition für den Begriff *Spam*

geliefert, und es werden verschiedene Verfahren zum Senden und Filtern von Spam-Nachrichten vorgestellt. Zudem liefert eine Auflistung der aktivsten Spambots und der von diesen durchschnittlich versendeten Nachrichten die Motivation für den in Kapitel 4 präsentierten neuen Spam-Filter. Mit einer Abgrenzung der Clusteranalyse von der Klassifikation werden die Grundlagen der in Kapitel 3 eingesetzten Methoden des maschinellen Lernens vorgestellt und an praktischen Beispielen nachvollzogen. Das Kapitel 2 endet mit der Einführung in die formalen Sprachen und der Beschreibung von LEX und YACC. Letztere werden zur lexikalischen und syntaktischen Analyse verwendet und bilden die Grundlage der in dieser Arbeit präsentierten Filter.

Die in Kapitel 3 vorgestellten Filter beschreiben Programmfamilien anhand charakteristischer Verhaltensmuster und können verwendet werden, um unbekannte Programme bereits bekannten Familien zuordnen. Die Analyse und Beschreibung des in einer Sandbox aufgezeichneten Verhaltens erfolgt in dem eigens hierfür entwickelten *Malware Instruction Set* (MIST). Diese Sprache optimiert die von Sandboxen generierten Verhaltensreporte für die Weiterverarbeitung mit Techniken des maschinellen Lernens. MIST wird im Detail vorgestellt und evaluiert, bevor ein auf der CWSANDBOX, MIST und dem Cluster- und Klassifikationswerkzeug MALHEUR basierendes Analysesystem für Schadprogramme erläutert und mit konkurrierenden Ansätzen verglichen wird. Abschließend werden die sich aus MIST ableitende Grammatik für Verhaltensreporte und die auf dieser kontextfreien Grammatik basierenden Filter für Schadprogramme beschrieben. Die Evaluierung des Filteransatzes erfolgt an einem Datensatz mit rund 3.000 Verhaltensreporten.

Kapitel 4 befasst sich mit dem Filtern von Spam-Nachrichten. Nach einer Abgrenzung der Arbeit von ähnlichen Filtersystemen, wird der Aufbau des dreistufigen Analysesystems aus *Sandnet*, Filtergenerator und Filter-Plugin vorgestellt. Als *Sandnet* wird in diesem Zusammenhang eine Analyseumgebung zum kontrollierten Ausführen von Spambots bezeichnet. Die während der Analyse verschickten Nachrichten werden innerhalb dieses Sandnets auf einen lokalen Mail-Server umgeleitet und zum Aufbau passender Spam-Filter herangezogen. Mit den auf einem regulären Ausdruck oder einer kontextfreien Grammatik basierenden Filtern werden zwei Filtermethoden für musterbasierte Spam-Kampagnen vorgestellt. Beide Filteransätze werden im Detail beschrieben und an praktischen Beispielen motiviert. Auf die für den zweiten Filteransatz notwendige Grammatik für E-Mail-Nachrichten wird an dieser Stelle ebenso eingegangen, wie auf die konkrete Implementierung des Filtergenerator. Die Güte der Filtermethoden wird jeweils an mehreren Spam-Kampagnen verschiedener Spambots bestimmt und miteinander verglichen.

Den Abschluss dieser Arbeit bildet Kapitel 5, das die Arbeit nochmals zusammenfasst und mögliche Erweiterungen der einzelnen Systeme anspricht.

1.5 Kooperation und Eigenleistung

Die in dieser Arbeit vorgestellten Analysesysteme und Filter kombinieren Methoden und Werkzeuge aus unterschiedlichen Bereichen der Informatik. Neben der optimalen Kombination sind auch die Werkzeuge selbst – beispielsweise CWSANDBOX zur verhaltensbasierten Analyse von Schadprogrammen und MALHEUR zur Cluster- und

Klassifikationsanalyse der Verhaltensreporte – entscheidend für die Güte und Effizienz der entwickelten Systeme zur Analyse und Filterung von Schadprogrammen und Spam. Erst durch die Zusammenarbeit mit Kollegen aus den verschiedenen Bereichen konnten die vorgestellten Systeme mit den durchweg sehr guten Ergebnissen realisiert werden.

Das im dritten Kapitel vorgestellte Analysesystem für Schadprogramme wurde in Zusammenarbeit mit Thorsten Holz, Konrad Rieck und Carsten Willems erarbeitet. Eine zentrale Rolle in diesem System besitzt das von mir entwickelte *Malware Instruktion Set* (MIST). Diese abstrakte Sprache zur Beschreibung von Programmverhalten stellt das Bindeglied zwischen der verhaltensbasierten Analyse in der CWSANDBOX und der Cluster- und Klassifikationsanalyse in MALHEUR dar. Die in MIST realisierte Reduktion der Verhaltensberichte und die optimierte Darstellung der Informationen hat entscheidenden Einfluss auf die Güte des Gesamtsystems und eröffnet zudem die Möglichkeit, Verhaltensberichte verschiedener Analysesysteme miteinander zu vergleichen. Das Filtersystem für Verhaltensberichte wurde ebenfalls von mir entworfen. Das noch unveröffentlichte System greift die Eigenschaften von MIST als kontextfreie Sprache auf und stellt neben der Klassifikationsanalyse mit MALHEUR eine weitere Möglichkeit dar, Verhaltensberichte von Programmen zu klassifizieren. Da die charakteristischen Verhaltensmuster der Programmfamilien direkt in die Filter einfließen, sind die MIST-Filter, im Gegensatz zu MALHEUR, leicht verständlich und liefern eine lesbare Beschreibung der Programmfamilien.

Das einfache Filtersystem für Spam (Kapitel 3) wurde zusammen mit Jan Göbel und Thorsten Holz entwickelt. Insbesondere der in diesem System verwendete Algorithmus zur Filtergenerierung wurde von mir entworfen und zusammen mit Jan Göbel umgesetzt. Die kontextfreie Grammatik für E-Mail-Nachrichten und die auf dieser aufbauende Filtermethodik wurden unabhängig von dem einfachen Filtersystems von mir entwickelt. Weder die Grammatik noch die Filtermethodik wurden bisher veröffentlicht. Obwohl auch hier Spam-Bots in einer kontrollierten Umgebung ausgeführt und die abgegriffenen Nachrichten zum Lernen eines passenden Filters verwendet werden, setzt die auf einer kontextfreien Grammatik für E-Mail-Nachrichten basierenden Filtermethode auf einer anderen, neuartigen Annahme auf: Jeder Spam-Kampagne liegt nur eine begrenzte Anzahl an Wörtern und Formulierungen zugrunde. Folglich wird bei diesem Ansatz nicht mehr der Aufbau der Nachricht, sondern deren „Inhalt“ als Filterkriterium herangezogen. Der Filteransatz versucht nicht mehr nur das von den Spammern verwendete Muster zu extrahieren, sondern lernt mit den Wörtern und Sätzen den „Inhalt“ der Kampagne. Die Filtermethodik ist damit mächtiger als alle konkurrierenden Ansätze und überwindet einen Großteil der diesen innewohnenden Beschränkungen.

1.6 Publikationen

Die vorliegende Arbeit basiert auf den im Folgenden aufgelisteten Veröffentlichungen, stellt aber auch vollkommen neue und bisher unveröffentlichte Ideen und Ergebnisse vor.

Kapitel 3 basiert in erster Linie auf den Arbeiten „A Malware Instruction Set for Behavior-Based Analysis“ [Trinius et al., 2010] und „Automatic Analysis of Malware Behavior using Machine Learning“ [Rieck et al., 2011]. Beide Arbeiten wurden in Zusammenarbeit mit Konrad Rieck, Carsten Willems und Thorsten Holz erstellt und beschreiben das *Malware Instruction Set* und das dieses integrierende System zur verhaltensbasierten Analyse von Schadprogrammen. Obwohl die Idee, *Mist* als Sprache zu konstruieren, schon 2008 in „Malwareklassifizierung und -clustering mit MIST“ [Trinius, 2008] vorgestellt wurde, handelt es sich bei der aus MIST abgeleiteten kontextfreien Grammatik und den darauf basierenden Verhaltensfilter um bisher unveröffentlichte Forschungsergebnisse.

Die erste Version des proaktiven Filtersystems für Spam-Nachrichten wurde in „Towards Proactive Spam Filtering“ [Göbel et al., 2009] vorgestellt. Dieses in Zusammenarbeit mit Jan Göbel und Thorsten Holz entwickelte Filtersystem nutzt die in Abschnitt 4.3 beschriebenen, auf regulären Ausdrücken basierenden Filter. Die kontextfreie Grammatik und die daraus resultierenden Spam-Filter stellen neue und unveröffentlichte Forschungsideen und -ergebnisse dar.

Im Folgenden sind ausgewählte Veröffentlichungen aufgeführt, die thematisch nicht im Fokus dieser Arbeit liegen, diese aber trotzdem an verschiedenen Stellen direkt, oder indirekt beeinflusst haben.

Hinter den mit Markus Engelberth, Felix Freiling, Christian Gorecki, Jan Göbel, Thorsten Holz, Ralf Hund und Carsten Willems erstellten Veröffentlichungen zum *Internet-Malware-Analyse-System* (InMAS) [Engelberth et al., 2010a,b, 2011] steht ein mehrjähriges, durch das Bundesamt für Sicherheit in der Informationstechnik (BSI) gefördertes Forschungsprojekt. Das in Kapitel 3 präsentierte Analyse- und Filtersystem für Schadprogramme löst eine im Rahmen der InMAS-Entwicklung aufgetretene Fragestellung und wurde in das InMAS integriert. Zudem flossen einzelne Entwicklungen, wie das in Abschnitt 2.1.2 beschriebene System zum Zurücksetzen infizierter Analysesysteme, direkt in diese Arbeit ein. Eine saubere Analyseumgebung ist für die in Kapitel 3 vorgestellte verhaltensbasierte Analyse zwingend erforderlich.

Basierend auf dem *Malware Instruction Set* wurden in Zusammenarbeit mit Thorsten Holz, Jan Göbel und Felix Freiling verschiedene Methoden zur Visualisierung von Programmverhalten untersucht. Auf die in „Visual Analysis of Malware Behavior Using Treemaps and Thread Graphs“ [Trinius et al., 2009a] veröffentlichten Darstellungsformen wurde während der Entwicklung des in Kapitel 3.3 präsentierten Analysesystems mehrfach zurückgegriffen, um die von MALHEUR berechneten Cluster greifbar zu machen und den Algorithmus weiter zu optimieren.

KAPITEL II

Grundlagen

In diesem Kapitel werden die Grundlagen diskutiert, die zum Verständnis der Arbeit notwendig sind. Dazu werden im ersten Abschnitt die verschiedenen Schadprogramm-Klassen eingeführt und die von diesen ausgehenden Gefahren exemplarisch diskutiert. Anschließend werden die Konzepte der *statischen* und *dynamischen* Programmanalyse vorgestellt. Die dynamischen Ansätze werden dabei eingehender betrachtet, da die im Rahmen der Arbeit präsentierten Filteransätze auf den Ergebnissen dynamischer Schadprogrammanalysen aufbauen. Abschnitt 2.2 führt den Begriff *Spam* ein. Neben einem Überblick über die verschiedenen Spam-Techniken werden auch *klassischen* Filtermechanismen vorgestellt. Mit der Clusteranalyse und der Klassifikationen werden in dieser Arbeit zwei grundlegenden Techniken des maschinellen Lernens angewendet. Beide Verfahren werden in Abschnitt 2.3 beschrieben und voneinander abgegrenzt. Die beiden Konzepte werden dabei jeweils an konkreten Beispielen nachvollzogen. Abschnitt 2.4 liefert eine Einführung in die formalen Sprachen. Zum besseren Verständnis werden die notwendigen Definitionen dabei an einer einfachen Sprache nachvollzogen. Auf diese Sprache wird auch bei der Vorstellung der beiden Werkzeuge LEX und YACC in Abschnitt 2.5 zurückgegriffen, um die Grundlagen der lexikalischen und syntaktischen Analyse vorzustellen und die Umsetzung dieser in PLY zu diskutieren. Den Abschluss des Kapitels bildet eine Zusammenfassung der präsentierten Grundlagen.

2.1 Schadprogramme

Schadprogramme stellen eine der größten Gefahren im Internet dar. Die große Masse an täglich in Umlauf gebrachten Schadprogrammen und deren Vielfalt macht den Einsatz spezieller Schutzprogramme, wie Virenscanner, Firewalls oder *Intrusion Detection* Systeme für Privatanwender und Firmen unerlässlich. Zusätzlich versuchen Regierungen die Aktivitäten der Schadprogramme mit sogenannten *Frühwarnsystemen* zu überwachen, um möglichst auf ein ansteigendes Gefahrenlevel reagieren zu können [Apel et al., 2010, Engelberth et al., 2010a,b]. In diesen werden beispielweise *Honeypots* [Bächer et al., 2006, Göbel, 2010, Göbel and Trinius, 2010, Trinius, 2007]

eingesetzt, um die Geschwindigkeit, mit der sich die Schadprogramme verbreiten, zu messen und die verwendeten Angriffsvektoren zu protokollieren.

Das von MICROSOFT betriebene *Malware Protection Center* hat die in Abbildung 2.1 dargestellte Statistik für das zweite Quartal 2010 veröffentlicht. Darin ist die *Computer Cleaned per Mille (CCM)* Rate aufgetragen. Die CCM-Rate entspricht der Anzahl gesäuberter Systeme pro 1.000 Ausführungen des MALICIOUS SOFTWARE REMOVAL TOOLS (MSRT) [Microsoft, 2010a].

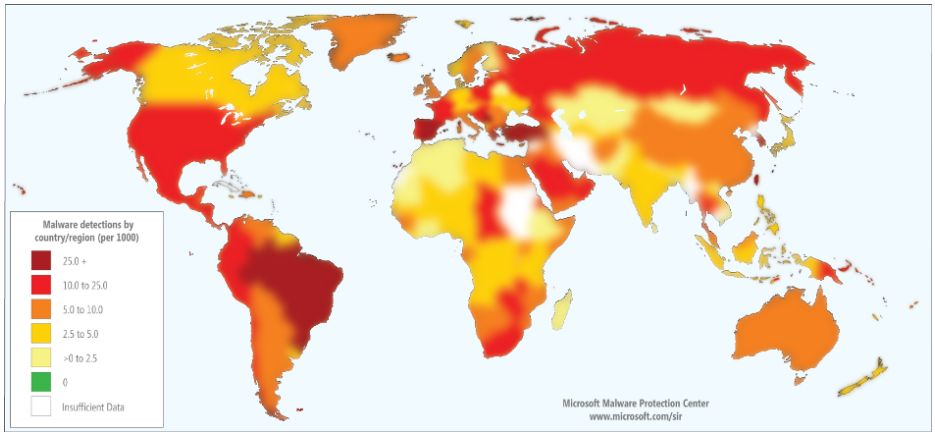


Abbildung 2.1: Übersicht über die Infektionsraten nach Ländern/Regionen für das 2. Quartal 2010, gemessen in CCM [Microsoft, 2010b].

Gerade die Vielfalt der verbreiteten Schadprogramme stellt eine große Herausforderung für die Hersteller zum Beispiel von Virenscannern dar. In Abbildung 2.2 ist die Anzahl der jährlich von SYMANTEC neu entwickelten Virensignaturen dargestellt. Während im Jahr 2002 noch 20.254 Signaturen ausreichten, um die von Schadprogrammen ausgehenden Gefahren abzuwehren, mussten 2008 schon über 1,5 Millionen und 2009 fast 2,8 Millionen neue Signaturen entwickelt werden. Da Signaturen in der Regel nicht für ein einzelnes Schadprogramm entwickelt werden, sondern – soweit möglich – Schadprogrammfamilien von einer Signatur abgedeckt werden sollen, ist die Anzahl neuer Schadprogramme für die jeweiligen Jahre folglich weit höher anzusetzen.

Neben der verloren gegangenen Integrität des infizierten Systems geht von den Schadprogrammen eine Vielzahl weiterer Gefahren für das Opfer oder auch Dritte aus. In der Regel hängen diese direkt mit dem eigentlichen Ziel des Autors oder Verbreiters zusammen. Die wenigsten Angreifer begnügen sich hierbei mit dem Löschen oder anderweitigem Zerstören von Daten. Schadprogramme werden stattdessen zum Beispiel dazu eingesetzt, Geld von Opfern zu erpressen [Kirk, 2006], oder können die auf dem infizierten System durchgeführten Online-Überweisungen *on-the-fly* abändern und das Geld so auf andere Konten umleiten [Utakrit, 2009]. Durch den Zusammenschluss von tausenden Schadprogrammen (Bots) zu sogenannten Botnetzen verfügen Angreifer über genug Ressourcen, um massenhaft Spam-Nachrichten zu versenden [Kreibich et al., 2008, Xie et al., 2008], oder die komplette Informations-

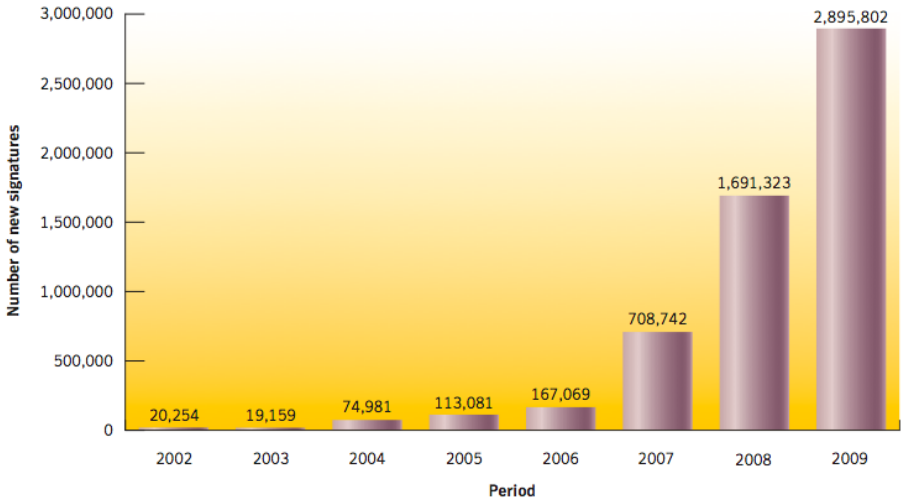


Abbildung 2.2: Die Anzahl der von SYMANTEC jährlich neu entwickelter Virensignaturen [Symantec, 2010b].

und Telekommunikations-Infrastruktur eines ganzen Landes zum Zusammenbruch zu bringen [Lesk et al., 2007]. Angriffe dieser Art werden als verteilte *Denial-of- Serive*-Angriffe bezeichnet und lassen sich nur sehr eingeschränkt abwehren. Mit dem STUXNET-Wurm wurde 2010 der erste großflächige Angriff auf Produktionsanlagen beobachtet. Zwar verbreitete sich der Wurm auch über Windows-Systeme, richtete auf diesen aber keinen „Schaden“ an. Das eigentlichen Ziel des Wurms waren Steuerungsanlagen von SIEMENS [Falliere et al., Februar 2011].

Schon diese kleine Auswahl an Beispielen zeigt die Vielfalt heutiger Schadprogramme, und spätestens mit dem STUXNET-Wurm wird deutlich, dass Schadprogramme inzwischen von sehr versierten Entwicklern geschrieben werden, die über ausreichend finanzielle Ressourcen und den Zugang zu internen und teils geheimen Informationen verfügen.

2.1.1 Klassifikation von Schadprogrammen

Im Folgenden werden mit *Viren*, *Würmern*, *Trojanern* und *Bots* die vier bei der Klassifikation von Schadprogrammen am häufigsten verwendeten Kategorien vorgestellt. Die Arten werden – soweit möglich – voneinander abgegrenzt und es wird zusätzlich auf den typischen Verbreitungsweg der jeweiligen Art eingegangen.

Viren. Schadprogramme die sich selbst reproduzieren, aber nicht autonom über das Internet verbreiten können, werden als Viren (Computerviren) bezeichnet. Viren sind immer auf den lokalen Rechner und die in das System lokal eingebundenen (externen) Datenträger und Netzlaufwerke beschränkt. Zur Infektion entfernter Systeme kommt es erst, wenn ein mit dem Virus infiziertes Programm oder Dokument – man spricht in diesem Zusammenhang auch von *Wirts*programmen – von dem Anwender verbrei-

tet wird. Die meisten Viren werden in E-Mail-Anhängen verbreitet, allerdings bieten externe Datenträger wie USB-Sticks und externe Festplatten, kombiniert mit der in vielen Betriebssystemen aktivierten *Autostart*-Funktion, einen ebenfalls zuverlässigen Verbreitungsvektor. Listing 2.1 zeigt die auf einem USB-Stick gefundene, von einem Virus manipulierte Datei *autorun.inf*.

```
1 [autorun]
2 open=RECYCLER\S-1-5-21-1482476501-1644491937-682003330-1013\windows32.exe
3 icon=%SystemRoot%\system32\SHELL32.dll,4
4 action=Open folder to view files
5 shell\open=Open
6 shell\open\command=RECYCLER\S
      -1-5-21-1482476501-1644491937-682003330-1013\windows32.exe
7 shell\open\default=1
```

Listing 2.1: Beispiel einer infizierten `autorun.inf` Datei.

Beim Einstecken des USB-Sticks in ein WINDOWS-System werden die in der *autorun.inf* Datei enthaltenen Informationen ausgewertet. Bei aktiviertem Autostart wird die infizierte Datei `RECYCLER\...\windows32.exe` ausgeführt. Ist die Funktion deaktiviert, wird dem Anwender stattdessen ein Dialog angezeigt, mit dem ihm die Option *Open folder to view files* unter dem entsprechenden *Icon* angeboten wird. Wählt der Anwender die Option aus, wird statt des WINDOWS EXPLORERS ebenfalls die infizierte Datei gestartet und das System infiziert.

Würmer. Würmer sind selbst replizierende Schadprogramme, die sich autonom in lokalen Netzen und dem Internet verbreiten und dabei Schwachstellen in Betriebssystemen und Anwendungen ausnutzen. Den Würmern stehen zur Verbreitung unter anderem dieselben Dienste zur Verfügung, die von Anwendern zur Kommunikation und zum Austausch von Daten untereinander verwendet werden. Dazu gehören der E-Mail-Dienst, *Instant Messenger*, *Peer-to-peer (p2p)*-Tauschbörsen, soziale Netzwerke oder auch der *Twitter*-Dienst. Im Unterschied zu Viren – diese lassen sich prinzipiell auch über diese Dienste verbreiten – benötigen Würmer keine Anwenderinteraktion, sondern bringen die notwendige Funktionalität entweder selbst mit oder können Anwendungen, wie zum Beispiel MIRCOSOFT OUTLOOK, entsprechend instrumentalisieren. Dazu durchsuchen sie beispielsweise auf dem System vorhandene Adressbücher nach E-Mail-Adressen, um sich anschließend im Namen des Opfers an diese zu verschicken.

In den meisten Fällen bauen die Würmer für ihre Verbreitung aber nicht auf existierende Dienste auf sondern durchsuchen das Internet selbstständig nach verwundbaren Diensten, die von anderen Rechnern angeboten werden. Durch das Ausnutzen der in den Diensten gefundenen Schwachstellen – dem *Exploit* der Schwachstelle – gelangt der Wurm auf das System. Auf diesem installiert, kann er sich anschließend weiter verbreiten [Moore et al., 2002].

Mit den immer sicherer werdenden Betriebssystemen hat sich der Fokus der Angreifer in den letzten Jahren in Richtung der Anwendungen von Drittanbietern ver-

schoben [Secunia, 2010]. Viele weit verbreitete Anwendungen wie Internet Browser, Mediaplayer und Dokumentbetrachter besitzen zahlreiche Schwachstellen, die leicht zu finden und auszunutzen sind. Abbildung 2.3 zeigt, wie sich das Verhältnis zwischen Schwachstellen in MICROSOFT-Produkten und den Produkten von Drittanbietern in den letzten Jahren verschoben hat.

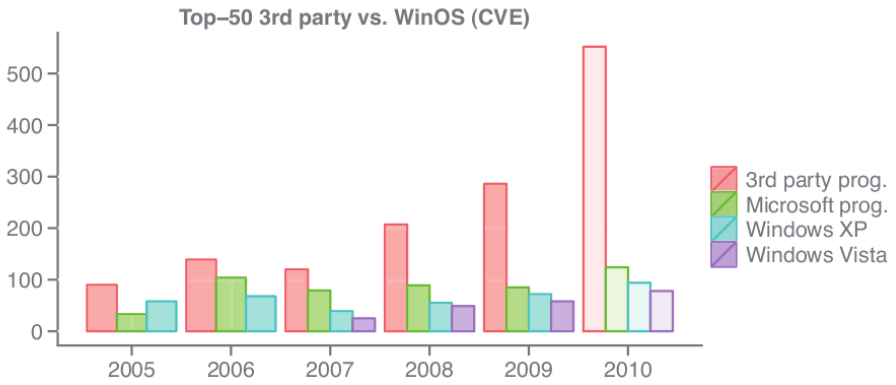


Abbildung 2.3: Aufschlüsselung der jährlich gefundenen Schwachstellen nach Drittanbieter- und MICROSOFT-Programmen, sowie den beiden MICROSOFT Betriebssystemen WINDOWS XP und WINDOWS VISTA [Secunia, 2010]. Die Schätzungen für 2010 basieren auf Daten der ersten Jahreshälfte.

Neben mangelnder Sorgfalt beim Programmieren stellt der bei vielen Produkten von Drittanbietern fehlende automatische Aktualisierungsmechanismus hier die größte Gefahr dar. Selbst wenn seitens des Anbieters Sicherheitsupdates bereitgestellt werden, werden diese von vielen Anwendern nicht wahrgenommen und somit auch nicht installiert.

Trojaner. Als Trojaner werden Programme bezeichnet, die neben ihrer „offiziellen“ Funktionalität noch weitere, dem Anwender nicht bekannte Funktionen auf dem System ausführen. Sie besitzen keinen eigenen Verbreitungsmechanismus und müssen durch Würmer oder Anwender verbreitet werden. Insbesondere bei der Verbreitung durch Anwender hilft dem Trojaner die Tarnung als nützliches Programm, beziehungsweise die Koppelung mit einem solchen. Der Trojaner wird dadurch von Anwendern „freiwillig“ auf das System geladen und dort mit den Rechten des Anwenders ausgeführt.

Mit Hilfe von Trojanern lassen sich viele unterschiedlicher Schadprogramme in einen Rechner einschleusen. Abhängig davon, ob der Trojaner die Schadfunktion enthält oder diese erst nach seiner Installation von einem entfernten Server lädt, wird zwischen *Trojan-Dropper* und *Trojan-Downloader* unterschieden. Die Bandbreite der existierenden Trojaner reicht von relativ ungefährlichen *Internet-Klickern* über *Banking-Trojaner* bis hin zu *Backdoor-Trojanern* [Lab, 2005]. Das Kategorisieren von Trojanern ist dabei nicht eindeutig, da die meisten Trojaner sich nicht nur auf Funktionen einer Kategorie beschränken. Beispielsweise enthalten viele Trojaner sogenannte

Trojan-Notifier, die den Angreifer über die erfolgreiche Installation des Trojaners auf dem attackierten System informieren.

Bots. Der Begriff „Bot“ leitet sich von *robot* ab. Analog zu Robotern stellen Bots Programme dar, die ohne menschlichen Eingriff Aufgaben wiederholt durchführen. Ein Bot ist damit nicht zwangsläufig auch ein Schadprogramm. Zum Beispiel setzen die Betreiber von Internet-Suchmaschinen sogenannte *Webcrawler* ein. Diese Bots durchsuchen das Internet selbstständig nach Internetseiten und indizieren diese. Auf den Seiten eventuell vorhandene Links werden dabei automatisch verfolgt, um im Anschluss den Inhalt der Seite auswerten zu können. Ohne Webcrawler wäre die vorhandene, breite Indizierung und Klassifizierung von Internetseiten nicht möglich. Ein weiterer oft eingesetzter Bot ist *Eggdrop* [Pointer, 2006]. Dieser *Internet Relay Chat* (IRC)-Bot wurde zur Automatisierung von IRC-Funktionen entwickelt.

Im Zusammenhang mit Schadprogrammen bezeichnet Bot ein Programm, dass es einem Angreifer ermöglicht, das infizierte System fernzusteuern. Nachdem sie auf dem kompromittierten System installiert sind, können Bots zudem zu sogenannten *Botnetzen* zusammengeschlossen werden, die sich zentral durch den *Botmaster* steuern und kontrollieren lassen. Zentralisierte Botnetze werden, wie in Abbildung 2.4 dargestellt, um einen zentralen *Command & Control (C&C)*-Server herum aufgebaut. Neben der dargestellten zentralen Topologie, nutzen Botnetze auch Topologien mit mehreren C&C-Servern, hierarchische Strukturen, oder auch *peer-to-peer*-Strukturen [Ollman, 2010].

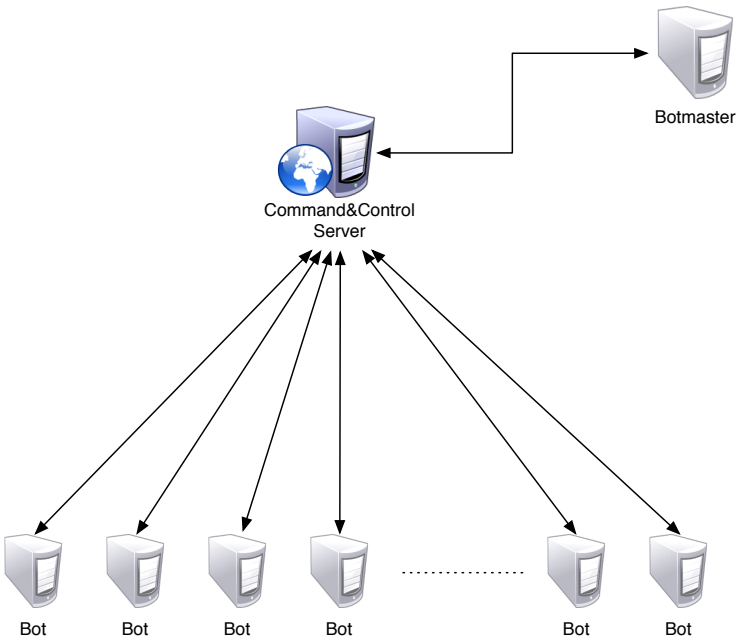


Abbildung 2.4: Klassischer Aufbau eines zentralisierten Botnetzes. Der Botmaster kommuniziert hier nur über den C&C-Server mit den einzelnen Bots.

Die durch Bots kontrollierten Rechner werden auch als *Zombies* bezeichnet. Nachdem der Angreifer einen Rechner kompromittiert hat, installiert er auf diesem zum Beispiel einen zum Bot modifizierten IRC-Client. Dieser verbindet sich zu einem voreingestellten IRC-Server, dem C&C-Server, auf einem festen *Channel* und wartet auf Kommandos. Die Kommandos, die ein Bot unterstützt, sind jeweils abhängig von seiner Entwicklungsstufe und seinem Einsatzgebiet.

Botnetze werden bevorzugt bei Angriffen eingesetzt, für die hohe Bandbreiten oder Rechenleistung notwendig sind [Banday et al., 2009]. Diese Ressourcen stehen dem Angreifer erst durch die Zusammenschaltung aller infizierten Systeme zur Verfügung und können beispielsweise für einen verteilten *Denial-of-Service*-Angriff auf einen Dienst, oder auf die IT-Infrastruktur eines Landes [Rötzer, 2007] eingesetzt werden. Auch beim Ausspionieren von Daten und dem Verbreiten von Schadprogrammen oder Spam-Nachrichten steigt die Effektivität mit der Größe des Botnetzes [Govil, 2007].

2.1.2 Analyse von Schadprogrammen

Um gegen die Verbreitung von Schadprogrammen vorgehen zu können und die von diesen ausgehenden Sicherheitsrisiken zu minimieren, müssen die Schadprogramme detailliert analysiert werden. Bei dieser Analyse unterschiedet man zwischen statischen und dynamischen Ansätzen. Im Folgenden werden beide Analyseformen kurz vorgestellt. Die dynamische Analyse von (Schad-) Programmen und das sich dabei ergebende Problem, die Analyseumgebung in einen sauberen Zustand zurückzusetzen, werden etwas detaillierter betrachtet, da die in den Kapiteln 3 und 4 vorgestellten Filtermethoden auf Ergebnissen dynamischer Programmanalysen basieren.

Statische Ansätze

Bei statischen Ansätzen zur Programmanalyse wird versucht, mit Hilfe von *Disassemblern* oder *Decompilern* den Binär- beziehungsweise Quellcode des Schadprogramms wiederherzustellen. Dabei kommt das Schadprogramm selbst nicht zur Ausführung.

Im Idealfall, das heißt, wenn der komplette Binärcode wiederhergestellt werden konnte, liefert die statische Analyse einen vollständigen Einblick in das Programm. Die sich an die Wiederherstellung des Codes anschließende Codeanalyse zeigt die möglichen Ausführungspfade des Programms und die im Programm enthaltenen Schadfunktionen auf.

Um diese detaillierte Codeanalyse von Programmen zu unterbinden, finden in den letzten Jahren vermehrt sogenannte *obfuscation*-Techniken Anwendung, bei denen der Programmautor oder ein Verbreiter versucht, den Binärcode beziehungsweise die Semantik des Programms vor potentiellen Analysten zu verstecken. Eine einfache und trotzdem vielversprechende Möglichkeit zum Verstecken des Codes bieten sogenannte *Packer*. Hierbei wird das compilierte Programm mit zum Teil eigens entwickelten Packern komprimiert und damit auch verschleiert. Die dazugehörige Entpackroutine muss zwar anschließend vor dem Programm platziert werden – anderenfalls wäre das Schadprogramm nicht mehr ausführbar – aber ein Analyst muss trotzdem zunächst den verwendeten Packer analysieren, bevor er das Programm selbst entpacken und analysieren kann. Weitere Techniken, die zum Erschweren des *Reverse Engineering*

eingesetzt werden können, sind zum Beispiel die Verwendung von polymorphem Code und verschiedene *Anti-Reversing* Techniken [Linn and Debray, 2003, Moser et al., 2007b, Popov et al., 2007].

Der große Vorteil der statischen Analyse liegt darin, dass alle möglichen Programmpfade betrachtet werden können und die Ergebnisse nicht von Umgebungsparametern beeinflusst werden. Neben den bereits erwähnten Störmaßnahmen seitens der Schadprogrammautoren stellt die nur sehr eingeschränkt mögliche Automatisierbarkeit den größten Nachteil der statischen Analyse dar. Die statische Analyse ist damit in der Regel sehr zeitaufwendig.

Dynamische Ansätze

Bei der dynamischen Analyse wird das Schadprogramm in einer instrumentalisierten Umgebung ausgeführt und beobachtet. Dabei unterscheidet man zwischen der Code- und Verhaltensanalyse. Zur Codeanalyse werden *Debugger* eingesetzt, die den Programmfluss zur Laufzeit überwachen. Diese erlauben es dem Analysten, den Ablauf zu verfolgen und zu unterbrechen (*Breakpoint*). Dabei können zusätzlich die Werte verschiedener Register ausgelesen und modifiziert werden. Die Verhaltensanalyse entspricht dagegen eher einer *Black-Box*-Analyse, bei der das Schadprogramm ausgeführt wird und der Analyst nicht den Code des Programms untersucht, sondern lediglich das Verhalten des Programms und seine Auswirkungen auf das System während der Programmausführung betrachtet. Gerade im Zusammenhang mit *obfuscation*-Techniken kommt dabei der größte Vorteil der dynamischen Verhaltensanalyse zum Tragen: Da das Programm sich in der Ausführung befindet und der Code selbst gar nicht betrachtet wird, greifen die erwähnten Verschleierungstechniken hier nicht. Unabhängig davon wie stark das Programm verschleiert oder verschlüsselt sein mag, wird es, um seinen Zweck zu erfüllen, letztendlich auch seinen eigentlichen Code ausführen und damit sein Verhalten offenlegen.

Neben einer Vielzahl von Standardwerkzeugen, wie zum Beispiel dem von MICROSOFT kostenlos zur Verfügung gestellten PROCESS MONITOR [Microsoft, 2006] zur Beobachtung von Programmverhalten und dem Aufzeichnen von Systemänderungen, wurden in den letzten Jahren vermehrt spezielle Werkzeuge und Analyseumgebungen entwickelt. Diese zeichnen das Verhalten eines Programms detailliert auf und sind zusätzlich in der Lage, die Ausführung zu unterbrechen und zu manipulieren [Bayer et al., 2006b, Song et al., 2008, Willems et al., 2007]. Innerhalb der Analyseumgebungen wird das Programm in einer kontrollierten Umgebung ausgeführt. Die kontrollierte, vielfach auch als *Sandbox* bezeichnete, Umgebung erlaubt, den Zugriff des Programms auf externe Ressourcen wie zum Beispiel das Internet, zu reglementieren. Zusätzlich können so alle vom Programm ausgehenden Systemaufrufe protokolliert werden. Als Ergebnis der dynamischen Analyse wird ein detaillierter Bericht geliefert, der sämtliche beobachteten Systemaufrufe und die jeweiligen Argumente enthält.

Im Gegensatz zur statischen Analyse lässt sich die dynamische Analyse sehr gut automatisieren. Analysesysteme können dabei täglich die Ausführung von tausenden neuen Schadprogrammen beobachten und entsprechende Verhaltensreporte erstellen. Auch wenn ein automatisiertes Erkennen der „Schadhaftigkeit“ momentan noch ein großes Problem in der dynamischen Verhaltensanalyse darstellt, können doch sehr

schnell Informationen über die gesammelten Schadprogramme gewonnen werden. Ein Problem bei der dynamischen Analyse sind die die Ergebnisse beeinflussenden Umgebungsparameter. Beispielsweise kann das Ausführen der eigentlichen Schadfunktion von bestimmten Parametern, wie der richtigen Sprachversion oder einer bestimmten Uhrzeit, abhängen. Findet das Schadprogramm diese Bedingungen nicht vor, kommt die Schadfunktion nicht zur Ausführung und kann damit von dem dynamischen Analysesystem auch nicht beobachtet werden. Vielfach setzten Schadprogrammautoren auch Sandbox-Erkennungsmechanismen ein, um Analysesysteme zu detektieren und entsprechend zu reagieren.

Bei der Verhaltensanalyse werden in der Regel keine Operationen unterbunden, das heißt, das Programm kann Schadfunktionen ausführen und sich so beispielsweise auf das Dateisystem kopieren. Daher muss nach jeder Analyse eines Schadprogramms die Analyseumgebung in einen „sauberen“ Zustand zurückversetzt werden, bevor das nächste Programm untersucht werden kann. Neben Hardware basierten Lösungen, wie zum Beispiel PC-Wächter [Dr. Kaiser Systemhaus] und anderen kommerziellen Programmen [Faronics, 2010], bieten virtuelle Maschinen hier eine kostengünstige Variante, um die Analyseumgebung einfach und schnell zurückzusetzen. Die Virtualisierung stellt allerdings einen zusätzlichen Umgebungsparameter dar, der beim Einsatz von entsprechenden Erkennungsmechanismen seitens der Schadprogramme die dynamische Analyse stark verfälschen kann. Im Rahmen dieser Arbeit wurden daher nur native Analysesysteme verwendet. Da die zum Zurücksetzen von nativen Systemen existierenden Hardwarelösungen nicht überall eingesetzt werden können und bei den Softwarelösungen Probleme mit der Überwachung des *Master Boot Record (MBR)* und direkten Schreibzugriffen auf den physischen Datenträger bestehen, wird in dieser Arbeit ein eigens entwickeltes Wiederherstellungssystem eingesetzt [Engelberth et al., 2008].

Zurücksetzen der Analyseumgebung

Bei dem zum Zurücksetzen der Analyseumgebung entwickelten System wird nach jeder Analyse die komplette Systempartition und der MBR des Systems unter Verwendung des Linux-Werkzeugs `dd` [Rubin et al.] neu geschrieben. Zur Steuerung des Wechsels zwischen Analyselauf und dem Wiederherstellen wird ein *Preboot eXecution Environment (PXE)*-Server zwischen den Anaysesystemen und dem Internet platziert. Abbildung 2.5 stellt den Aufbau des Systems schematisch dar.

Der Zyklus einer Analyse besteht aus vier Schritten:

1. Über das *PXE* wird Linux auf dem System gestartet.
2. Die vollständige Systempartition und der MBR werden mittels `dd` neu geschrieben.
3. Über PXE wird Windows gebootet.
4. Die Analyses des Schadprogramms wird durchgeführt. Anschließen erfolgt der Neustart des Systems (Schritt 1).

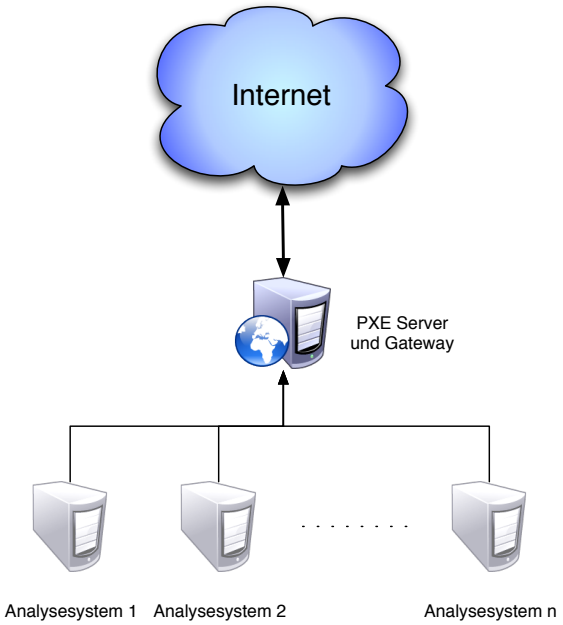


Abbildung 2.5: Schematischer Überblick über den Aufbau des Analysesystems mit Fokus auf dem Wiederherstellungsmechanismus.

Da die Systempartition des Analysesystems relativ groß ist, muss die Sicherungskopie der Partition im selben Rechner vorgehalten werden, um das Zurücksetzen mit vertretbarem Zeitaufwand durchführen zu können. Abbildung 2.6 zeigt den Aufbau der in den Rechnern verwendeten Festplatten.

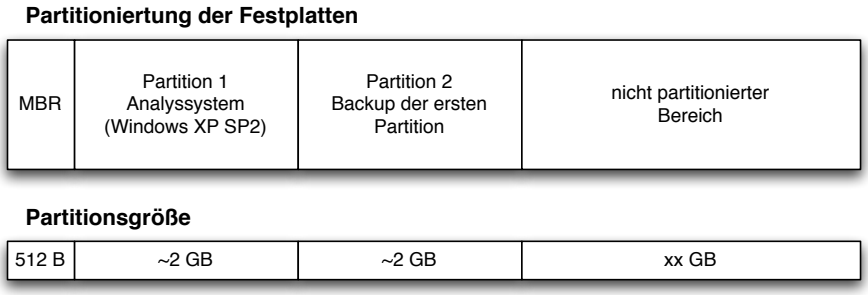


Abbildung 2.6: Aufbau der im Analysesystem verwendeten Festplatten.

In Partition 1 ist das Analysesystem – im vorliegenden Fall eine Windows XP SP2-Installation – enthalten. Nachdem dieses System vollständig konfiguriert ist, wird der komplette Inhalt in Partition 2 kopiert, beispielsweise unter Verwendung einer Linux Live CD und dem Kopierprogramm `dd`. Partition 2 enthält somit ein 1:1 Backup des Analysesystems. Sowohl Partition 1 als auch Partition 2 sind in diesem Fall so klein

wie möglich gehalten, um den Zeitaufwand für den Wiederherstellungsprozess zu minimieren. Der verbleibende freie Speicherplatz auf der Festplatte bleibt ungenutzt.

Ein eigens für das System aufgesetzter PXE-Server entscheidet in Abhängigkeit von dem letzten Zustand des Systems, ob das auf dem Rechner installierte Analysesystem gestartet wird, oder ob das System zurückgesetzt werden muss. Im letzteren Fall wird ein eigens erstelltes Linux-System über das *Trivial File Transfer Protocol* auf den Rechner übertragen und dort gebootet. Dieses kopiert die Sicherung des Systems aus Partition 2 zurück in Partition 1 und schreibt den MBR neu. Anschließend meldet es dem PXE-Server, dass das System wieder sauber ist und startet den Rechner neu.[1]

Verhaltensanalyse mit CWSandbox

Die in Kapitel 3 vorgestellten Verhaltensmuster basieren auf Reporten, die mit Hilfe von CWSandbox erstellt wurden. CWSandbox führt die Schadprogramme in einer instrumentalisierten Umgebung aus, beobachtet und protokolliert alle sicherheitsrelevanten Systemaufrufe und generiert daraus automatisch einen detaillierten Bericht [Willems et al., 2007].

Um das Verhalten des analysierten Programms aufzuzeichnen, verwendet CWSandbox hauptsächlich zwei Techniken: *API-Hooking* und *DLL-Injection*. Unter API-Hooking versteht man das Umleiten von *Windows API-Aufrufen* auf sogenannte *Hookfunktionen*. Dies sind eigene Funktionen, die zum Beispiel den API-Aufruf protokollieren oder die verwendeten Parameter modifizieren können. Meistens wird aus der Hookfunktion heraus auch die ursprüngliche API-Funktion aufgerufen oder deren Funktionalität simuliert. Da der Aufruf der ursprünglichen API-Funktion von der Hookfunktion durchgeführt wird, kann diese auch die Rückgabe der API-Funktion protokollieren und gegebenenfalls manipulieren.

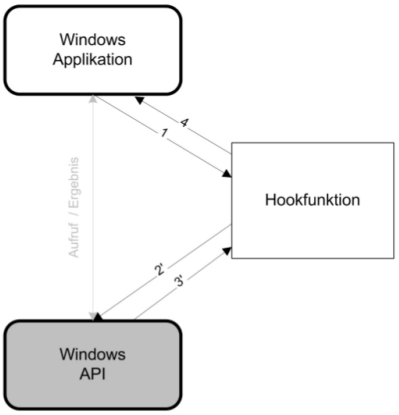


Abbildung 2.7: Ablauf des API-Hooking [Willems, 2007].

Abbildung 2.7 stellt die Umsetzung das API-Hooking schematisch dar. Schritt eins zeigt hier die Umleitung des API-Aufrufs auf die Hookfunktion. Im zweiten Schritt wird die ursprüngliche API-Funktion von der Hookfunktion aufgerufen und in Schritt

[1] Die Umsetzung der beschriebenen Analyseumgebung erfolgte durch Ben Stock.

drei erhält die Hookfunktion die Rückgabe der API-Funktion. Diese wird anschließend (un-)verändert an das aufrufende Programm weitergeleitet. Um innerhalb von beobachteten Programmen die API-Hookfunktionen zu installieren, muss eigener Code in diese eingeführt werden. Dazu verwendet CWSANDBOX die sogenannte DLL-Injection-Technik, bei der eine eigene DLL in einen laufenden oder gerade startenden Prozess eingefügt wird. Die Initialisierungsroutine dieser DLL installiert dann die Hooks für die gewünschten API-Funktionen. Durch diese Funktionalitäten ist es möglich, mit der CWSandbox automatisierte Verhaltensanalysen von Schadprogrammen zu erstellen. Der generierte Bericht enthält detaillierte Informationen über die analysierten Prozesse. Dazu gehören beispielsweise Informationen über Änderungen am Dateisystem und der Windows-Registry. Zudem umfasst der Bericht die Daten, die während der Analysephase über das Netzwerk gesendet wurden.

```
1  <analysis cwsversion="2.1.6" time="08.13.2009 16:23:13"
       file="C:\malware.exe" md5="74824">
2  <calltree>
3  <process_call pid="2556" filename="c:\malware.exe" starttime="00:01.906"
       startreason="AnalysisTarget"/>
4  <process_call pid="944" filename="C:\WINDOWS\system32\svchost.exe"
       starttime="00:04.515" startreason="DCOMService"/>
5  </calltree>
6  <processes>
7  <process pid="2556" filename="c:\malware.exe" username="Administrator"
       starttime="00:01.906" terminationtime="02:01.015" startreason="
       AnalysisTarget" applicationtype="Win32Application">
8  <thread tid="2560">
9  <load_image filename="c:\malware.exe" size="9096"/>
10 <load_dll filename="C:\WINDOWS\system32\ntdll.dll" size="757760"/>
11 ...
12 <create_mutex name="CTF.TimListCache.FMPDefaultS" owned="0"/>
13 ...
14 <connect_socket remote_addr="195.13.105.133" remote_port="80"/>
15 <connect_socket remote_addr="195.13.105.133" remote_port="80"/>
16 ...
17 <create_file filetype="file" srcfile="C:\DOKUME&#x7E;1\ADMINI&#x7E;1\
       LOKALE&#x7E;1\Temp\connect.html" flags="FILE_ATTRIBUTE_NORMAL
       SECURITY_ANONYMOUS"/>
18 ...
19 </process>
20 <process pid="944" filename="C:\WINDOWS\system32\svchost.exe" filesize="
       14336" username="SYSTEM" starttime="00:04.515" terminationtime="
       02:01.187" startreason="DCOMService">
21 <thread tid="2676">
22 <load_image filename="C:\WINDOWS\system32\svchost.exe" size="24576"/>
23 <load_dll filename="C:\WINDOWS\system32\ntdll.dll" size="757760"/>
24 ...
25 </process>
26 </processes>
27 <running_processes>
28 <running_process pid="300" filename="C:\WINDOWS\system32\svchost.exe"/>
29 ...
30 <running_process pid="2556" filename="c:\malware.exe"/>
31 </running_processes>
32 </analysis>
```

Listing 2.2: Der Aufbau eines CWSandbox-Reports.

In Listing 2.2 ist ein CWSandbox Verhaltensreport beispielhaft dargestellt[2]. Neben den Namen der aufgezeichneten API-Aufrufe sind dabei auch immer die übergebenen Argumente mit aufgeführt. Damit ist es möglich, das Verhalten im Detail nachzuvollziehen und beispielsweise die IP-Adressen der kontaktierten Server oder die von dem Schadprogramm verwendeten Mutexe zu identifizieren.

2.2 Spam

Die sich im Zusammenhang mit *Spam* als erstes aufdrängende Frage ist die nach einer Definition des Begriffs. Das Wort selbst liefert dabei keinerlei Anhaltspunkte, da sich Spam von *Spiced Ham* („gewürzter Schinken“) ableitet und keine weiterreichende Bedeutung hat. Zum Synonym für massenhaft über das Internet verschickte Werbenachrichten wurde Spam erst dank einem Sketch von *Monty Python*, in dem das Wort „Spam“ 132 mal genannt wurde.

Die folgenden Definitionen werden vielfach zur Definition von Spam herangezogen:

> „Spam is the term now generally used to refer to unsolicited electronic messages, usually transmitted to a large number of recipients. They usually, but not necessarily, have a commercial focus, promoting or selling products or services“ [National Office for the Information Economy (NOIE), 2003]

> „Spam is generally understood to mean the repeated mass mailing of unsolicited commercial messages by a sender who disguises or forges his identity.“ [Gauthronet and Drouard, 2008]

> „An electronic message is ‘spam’ if (A) the recipient’s personal identity and context are irrelevant because the message is equally applicable to many other potential recipients; AND (B) the recipient has not verifiably granted deliberate, explicit, and still-revocable permission for it to be sent.“ [The Spamhaus Project, 2010]

Auch wenn bis heute keine einheitliche Definition für Spam existiert, lassen sich aus den existierenden Definitionen doch drei Hauptkriterien herausfiltern, die E-Mail-Nachrichten erfüllen müssen, um als Spam klassifiziert zu werden [Schryen, 2007]:

1. Spam sind elektronische Nachrichten *(e-mail)*.
2. Spam sind nicht erbetene Nachrichten *(unsolicited)*.
3. Spam wird massenhaft verschickt *(bulk)*.

Nur wenn alle drei Bedingungen erfüllt sind, wird die Nachricht als Spam bezeichnet. Hat ein Empfänger beispielsweise einen *Newsletter* abonniert, können die in diesem Rahmen massenhaft verschickten (Werbe-) Nachrichten nicht als Spam klassifiziert werden. Ebenso handelt es sich bei einem Kettenbrief nicht um Spam, da dieser trotz langer Empfängerlisten, nicht als massenhaft versendet einzustufen sind.

Entsprechend den drei Kriterien wird Spam auch als *Unsolicited Bulk E-Mail* bezeichnet [The Spamhaus Project, 2010]. Für kommerzielle Spam-Nachrichten, wenn

[2] Um die Übersichtlichkeit zu erhöhen, wurden verschiedene Argumente modifiziert oder ausgelassen.

also der Sender und/oder Auftraggeber Spam-Nachrichten als Werbemittel nutzt, wird in diesem Zusammenhang der Begriff *Unsolicited Commercial E-Mail (UCE)* verwendet.

Spam-Statistiken

Der Anteil an Spam-Nachrichten an den täglich verschickten E-Mail-Nachrichten hat sich in den letzten Jahren bei circa 90 Prozent eingependelt (siehe Abbildung 2.8). Dabei hat auch das Vorgehen einzelner Firmen gegen Spam versendende Botnetze [Stewart, 2011] oder diese beherbergende Service-Provider [Stewart, 2009] jeweils nur zu einem kurzzeitigen Abfall des Spam-Volumens geführt. Die freigewordenen „Ressourcen“ werden meist relativ schnell von anderen Botnetzen übernommen.

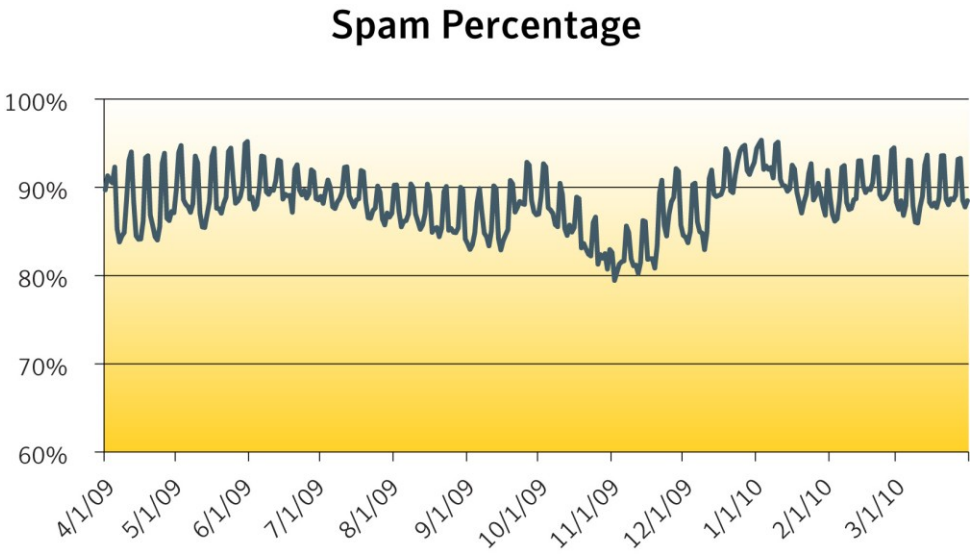


Abbildung 2.8: Anteil an Spam-Nachrichten an allen versendeten E-Mail-Nachrichten [Symantec, 2010a].

Das Spam-Volumen verharrt trotz der Weiterentwicklungen bei technischen Spam-Filtern und dem juristischen Vorgehen gegen Spammer auf einem extrem hohen Niveau. Das lässt sich auf die immer besser getarnten Nachrichten und die starke Verteilung der Spam-Quellen zurückführen. Durch immer besser getarnte Nachrichten wird es immer schwieriger, automatisiert Spam von normalen Nachrichten zu unterschieden. Waren die ersten Nachrichten schon allein an der schlechten Schreibweise zu erkennen, lassen sich aktuelle *Phishing*-Nachrichten nur noch sehr schwer von echten Nachrichten der betroffenen Institute unterscheiden.

Die weltweite Verteilung der Spam-Quellen sorgt dafür, dass die strafrechtliche Verfolgung von Spammern extrem schwierig ist. Dabei stehen die Kosten, beispielsweise zur Ermittlung und Zuordnung einer einzelnen IP-Adresse, in keinem Verhältnis zu dem letztlichen Nutzen. In der Regel gehören diese IP-Adressen zu infizierten Rechnern von „unbeteiligten Dritten“ (Bots). Das Risiko eines Spammers ist somit als sehr

gering einzustufen. Abbildung 2.12 zeigt die globale Verteilung der Spam-Quellen für März 2010.

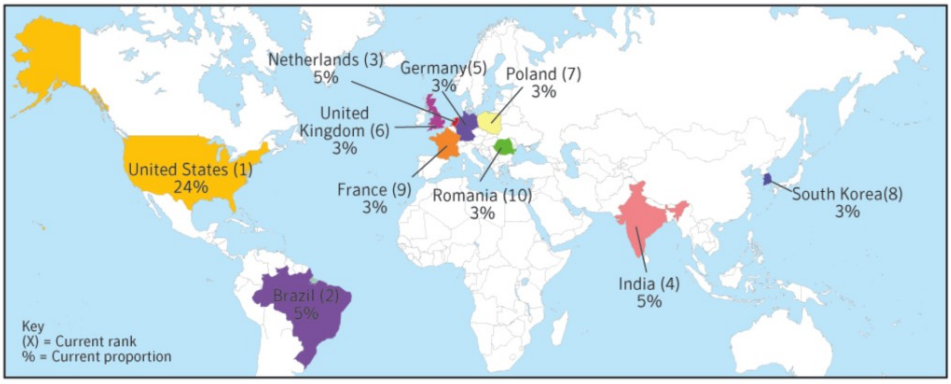


Abbildung 2.9: Weltweite Verteilung der Spam-Quellen [Symantec, 2010a].

Im März 2010 wurde laut Symantec [2010a] fast jede vierte Spam-Nachricht von einem Rechner aus den USA verschickt. Mit großem Abstand folgen Brasilien, die Niederlande und Indien mit jeweils 5 Prozent. Deutschland belegt in der Studie mit 3 Prozent aller verschickten Spam-Nachrichten Platz 5. Diese Studie stellt im Gegensatz zu den Studien zum Spam-Volumen nur eine Momentaufnahme dar und ist großen Schwankungen unterworfen. In der ersten Dezemberwoche 2010 stellte laut M86 Security [2010c] Indien mit 11,4 Prozent die größte Spam-Quelle dar, gefolgt von Russland (10,7 Prozent), Vietnam (7,4 Prozent) und Brasilien (6,4 Prozent). Die USA waren in diesem Zeitraum nur für rund 2,7 Prozent der Spam- Nachrichten „verantwortlich“.

2.2.1 Ausgewählte Spam-Techniken

Klassische Spam-Techniken

Die einfachste Methode Spam-Nachrichten zu verschicken wird als *direct spamming* bezeichnet. Dabei nutzt der Spammer die sich unter seiner Kontrolle befindenden Rechner, um von diesen direkt mit den Mail-Servern der Empfänger zu kommunizieren. Da dabei viele *Simple Mail Transfer Protocol (SMTP)*-Verbindungen aufgebaut werden, ist dieses Spam-Verfahren für *Internet Service Provider (ISPs)* einfach zu erkennen. Durch das Blocken der jeweiligen IP-Adressen oder des Anschlusses lässt sich das direkte Spammen einfach und effektiv unterbinden. Weil der Spammer seine eigenen Systeme verwendet, besteht für ihn zusätzlich die Gefahr, identifiziert und straf- und/oder zivilrechtlich verfolgt zu werden.

Als Reaktion auf die beschriebenen Probleme nutzten Spammer ab circa 1990 *Open Relays*, um die Spam-Nachrichten zu versenden [Jung and Sit, 2004]. Zu diesem Zeitpunkt waren die meisten Mail-Server noch als offene *Mail-Relays* konfiguriert und nahmen auch Nachrichten von und für Dritte entgegen. Spammer nutzten die – aus heutiger Sicht – „Fehlkonfiguration“ von Mail-Servern massiv aus. Heute stellen offene *Relays* kein Problem mehr dar. Die wenigen, noch als solche konfigurierten Mail-

Server lassen sich relativ einfach über *Blacklists* blockieren. Das Verhalten von Spammern, die auch heute noch nach *Open Replays* suchen, um diese zum Versenden von Spam-Nachrichten zu nutzen, lässt sich mit Hilfe sogenannter *Proxy-Pots* beobachten. Diese simulieren *Open Replays* und können Informationen über Spam-Kampagnen liefern, die in die Entwicklung und/oder Anpassung von Spamfiltern einfließen können [Andreolini et al., 2005, Pathak et al., 2008].

Analog zu den offenen *Relays* lassen sich auch offene Proxy-Server zum Versenden von Spam-Nachrichten nutzen. Diese werden von Spammern missbraucht, um die SMTP-Kommandos an die Mail-Server weiterzuleiten. Da nur der Proxy direkt mit dem Mail-Server kommuniziert, bleibt der Spammer selbst unerkannt. Da auch die Anzahl im Internet erreichbarer offener Proxy-Server beschränkt ist, lassen sich diese einfach durch den Einsatz von *Blacklists* filtern. Über offene Proxy-Server verschickte Spam-Nachrichten stellen daher – zumindest im Moment – keine größere Gefahr dar.

SOCKS-Proxy basierte Spam-Verfahren

Um die zur Filterung eingesetzten *Blacklists* zu umgehen, verwenden Spammer kompromittierte Rechner als „private“ Proxy. Über einen zwischen dem kompromittierten Rechner und dem C&C-Server aufgebauten *SOCKS*-Tunnel werden die SMTP-Kommandos, analog zu dem Spammen über offene Proxy-Server, an den Bot übertragen und von diesem an die Mail-Server weitergeleitet. Abbildung 2.10 stellt den Ablauf dieser Kommunikation schematisch dar.

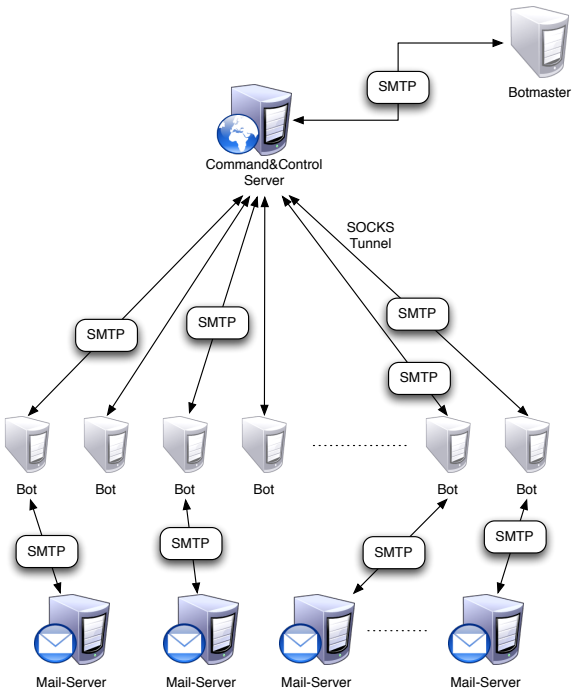


Abbildung 2.10: Schematischer Überblick über das auf einem *SOCKS-Proxy* basierte Spam-Verfahren.

Die Spam-Nachrichten werden dabei von zahlreichen Bots parallel ausgeliefert, was die Verwendung von einfachen *Blacklists* erschwert. Bei vielen infizierten Systemen handelt es sich zudem um über Breitbandanschlüsse an das Internet angebundene Systeme. Diese werden spätestens nach 24 Stunden durch den ISP vom Netz getrennt und bekommen beim erneuten Verbindungsaufbau eine neue IP-Adresse zugewiesen. Dadurch ändern sich die IP-Adressen der Bots relativ häufig was zu hoher Fluktuation auf den *Blacklists* führt.

Um auch hinter *Network-Address-Translation (NAT)-Routern* platzierte Bots bei diesem auf Tunneln basierten Verfahren verwenden zu können, kann die initiale Kommunikation zwischen Bot und C&C-Server und der Aufbau des Tunnels auch von dem Bot ausgehen [Honeynet Project, 2008].

Musterbasierte Spam-Verfahren

Die aktuell in den meisten Botnetzen eingesetzten Spam-Verfahren sind musterbasiert [Holz et al., 2008, Kreibich et al., 2008]. Dabei werden nicht die „fertigen“ SMTP-Kommandos an die Bots übertragen, sondern nur die Vorlagen (Muster), verschiedene *Fülltexte* und die Adresslisten. Abbildung 2.11 zeigt den schematischen Ablauf dieses Verfahrens.

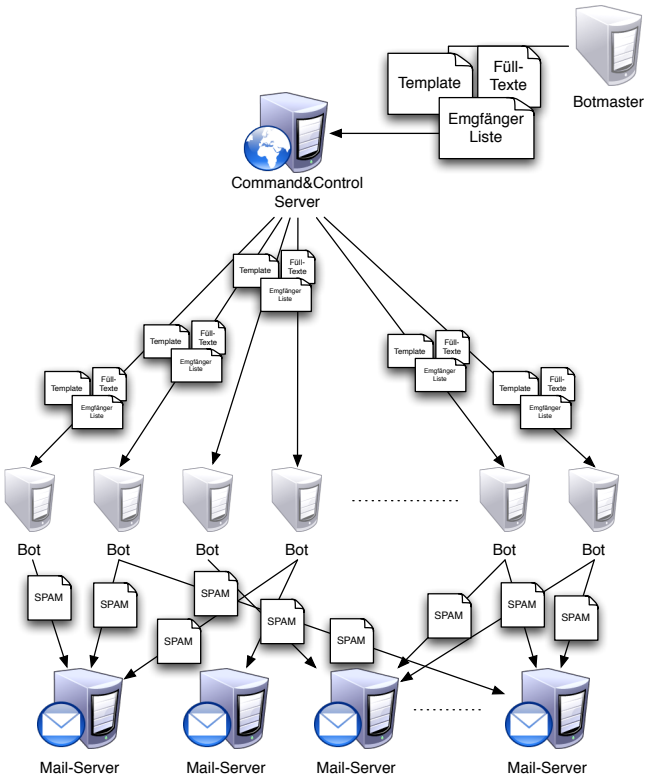


Abbildung 2.11: Schematischer Überblick über musterbasierte Spam-Verfahren.

Die Vorlage bestimmt dabei den Aufbau einer Nachricht und ist mit verschiedenen Markern versehen, an denen Textbausteine von den Bots eingefügt werden müssen. Über *Fülltexte* wird für Varianz zwischen den verschickten Spam-Nachrichten gesorgt. Diese Fülltexte können unter anderem verschiedene Betreffzeilen, Dateianhänge oder in die Nachricht einzubauende URLs umfassen. Die zusammengebauten Spam-Nachrichten werden anschließend von den Bots selbst verschickt, das heißt, neben dem Erstellen der Nachrichten wird auch die SMTP-Kommunikation mit den Mail-Servern von den Bots übernommen.

Aktuell wenden die meisten Botnetze musterbasierte Spam-Verfahren an (siehe Abschnitt 2.2.3). Dabei findet die Kommunikation zwischen dem Bot und dem C&C-Server meist verschlüsselt statt. Damit kann der optimale Spam-Filter – die verwendete Vorlage – für die ausgestoßenen Spam-Nachrichten nicht direkt abgegriffen werden, sondern muss mit den in Kapitel 4 präsentierten Methoden aus den Spam-Nachrichten extrahiert werden.

2.2.2 Technische Spam-Filter

Blacklisting

Wie bereits erwähnt setzen Mail-Server *Blacklists* von IP-Adressen ein, um das Senden von Nachrichten durch bekannte Spammer zu unterbinden. Alle von diesen IP-Adressen verschickten E-Mail-Nachrichten werden von den Mail-Servern abgelehnt und nicht weiter verarbeitet. In der Regel enthalten die Listen die Adressen von *Open Relays* oder offenen Proxy-Servern, allerdings existieren auch *Blacklists*, die anderweitig auffällig gewordene Systeme, beispielsweise einzelne Spambots, enthalten. Die Listen werden von den Mail-Servern über den *Domain Name Service (DNS)* abgefragt und daher auch als *DNS Blacklists (DNSBL)* bezeichnet.

Die Effizienz von *DNSBL*s wurde von Jung and Sit [2004] evaluiert, allerdings ist fraglich, ob die Ergebnisse heute noch Bestand haben. Mit den Botnetzen steht den Spammern heute eine neuer Verbreitungsmechanismus zur Verfügung, der resistent gegenüber *Blacklists* ist.

Im Gegensatz zu den *DNSBL*s können *Uniform Resource Identifier Blacklists* (URIBL) auch zum Filtern von Botnetz-Spam eingesetzt werden. Hierbei werden nicht die IP-Adressen der Sender zum Filtern verwendet, sondern Domainnamen und IP-Adressen, die im E-Mail-Text von bekannten Spam-Nachrichten versendet wurden. Die Filterung erfolgt somit auf dem Inhalt der Nachrichten. Neue E-Mail-Nachrichten müssen deshalb von dem Mail-Server immer erst vollständig entgegengenommen werden, bevor eine Filterung erfolgen kann.

Prüfsummen-Verfahren

Das Filtern von Spam-Nachrichten mittels Prüfsummen ist möglich, da die Spam-Nachrichten per Definition in großer Masse verschickt werden. Der Nachrichtentext bleibt dabei fast identisch und kann über unscharfe Prüfsummen-Verfahren (*fuzzy hashing*) gut abgebildet werden. Zum Filtern werden die Prüfsummen neu empfangener E-Mail-Nachrichten an einen zentralen Server übertragen, der die Spam-

Wahrscheinlichkeit über die Häufigkeiten der Prüfsummen bestimmt und zurückmeldet.

Bei Prüfsummen-basierten Verfahren wird der gesamte Inhalt der Nachrichten zum Berechnen der Prüfsumme herangezogen. Dabei lassen sich nur kleine Unterschiede, wie beispielsweise eine andere Anrede oder unterschiedlich viele Leerzeichen, durch unscharfe Prüfsummen-Verfahren abfangen. Existieren zwischen den Nachrichten einer Spam-Kampagne große Unterschiede, kann keine eindeutige Prüfsumme berechnet werden. Musterbasierte Spam-Kampagnen und insbesondere mit zufälligen Zeichenfolgen verwässerte Kampagnen – meist werden diese Zeichenfolgen in HTML-Kommentaren eingefügt und sind für den Empfänger nicht sichtbar – lassen sich daher nicht mit Prüfsummen-Verfahren filtern.

Heuristische und statistische Filtersysteme

Bei heuristischen Filtersystemen werden verschiedene Unterscheidungsmerkmale aus einem Datensatz klassifizierter Nachrichten extrahiert. Aus diesen Merkmalen werden Regeln erstellt, mit deren Hilfe neue Nachrichten gefiltert werden können. Die teilweise sehr „präzisen“ Regeln lassen sich dabei von Spammern auswerten und anschließend durch das Anpassen der Spam-Nachrichten umgehen. So änderten die Spammer zum Beispiel die Schreibweise des Wortes *Viagra* in *V1agra* oder *Vi@gra*, um von den regelbasierten Systemen nicht erkannt zu werden.

Die statistischen Filter basieren größtenteils auf dem Theorem von Bayes [Androutsopoulos et al., 2000, Sahami et al., 1998] oder anderen Techniken des maschinellen Lernens [Drucker et al., 1999]. Auch diese müssen auf vorab klassifizierten Datensätzen trainiert werden, um eine Abgrenzung von Spam- und normalen E-Mail-Nachrichten zu lernen. Beim naiven Bayes-Filter werden die in den Nachrichten des Trainingsdatensatzes gefunden Worte mit Wahrscheinlichkeiten belegt, die als Spam-Indikatoren dienen. Basierend auf diesen Wahrscheinlichkeiten lässt sich anschließend die Spam-Wahrscheinlichkeit für eine verdächtige Nachricht berechnen. Das Umgehen der Filter seitens der Spammer erfolgt auch hier durch das Umschreiben oder das Einfügen von Wörtern. Allerdings müssen diese einen negativen Spam-Indikator besitzen. Zufällige Zeichenfolgen sind in diesem Fall also nicht sinnvoll.

Greylisting

Der *Greylisting*-Ansatz ist orthogonal zu den übrigen Ansätzen, stellt aber einen relativ aggressive Filter dar. Entsprechend konfigurierte Mail-Server lehnen jede Nachricht von unbekannten Sendern ab und antworten stattdessen mit einer „temporarily reject“-Nachricht. Gleichzeitig wird das Tripel aus IP-Adresse und Sender- und Empfängeradresse in einer Liste abgespeichert. Wird nach einer Zeitspanne von typischerweise 30 bis 60 Minuten eine Nachricht mit demselben Tripel empfangen, stellt der Mail-Server diese zu und löscht den Eintrag in der *Greylist*.

Während legitime Mail-Server (Sender) das wiederholte Senden von nicht angenommenen Nachrichten unterstützen, ignorieren die meisten Botnetze die Antworten des Empfängers und führen keinen zweiten Versuch durch. Damit stellt das *Grayli-*

sting momentan einen sehr zuverlässigen Schutz gegen Botnetz-Spam dar, wobei eine Anpassung der Botnetze relativ einfach möglich ist und vermutlich bald erfolgen wird.

2.2.3 Aktuelle Spambots

Der überwiegende Anteil an Spam-Nachrichten wird aktuell über Botnetze verteilt. Diese bieten den Spammern ausreichend Ressourcen, um massenhaft Nachrichten zu verschicken, und gewährleisten gleichzeitig eine gesicherte Anonymität. Der Spammer beziehungsweise Botmaster nimmt nie selbst Kontakt zu den Mail-Servern auf, sondern versteckt sich, oder vielmehr seine IP-Adresse, hinter den Bots. Zusätzlich stellt diese auf tausende IP-Adressen verteilte Kommunikation die verschiedenen Filtertechniken vor große Probleme. Einfache Ansätze wie IP-Blacklists oder Schranken, die die Anzahl erlaubter Nachrichten pro IP-Adresse beschränken, laufen hier ins Leere. Die vielfach dynamisch vergebenen IP-Adressen für *Dial-in*-Anschlüsse, verstärken die Problematik zusätzlich.

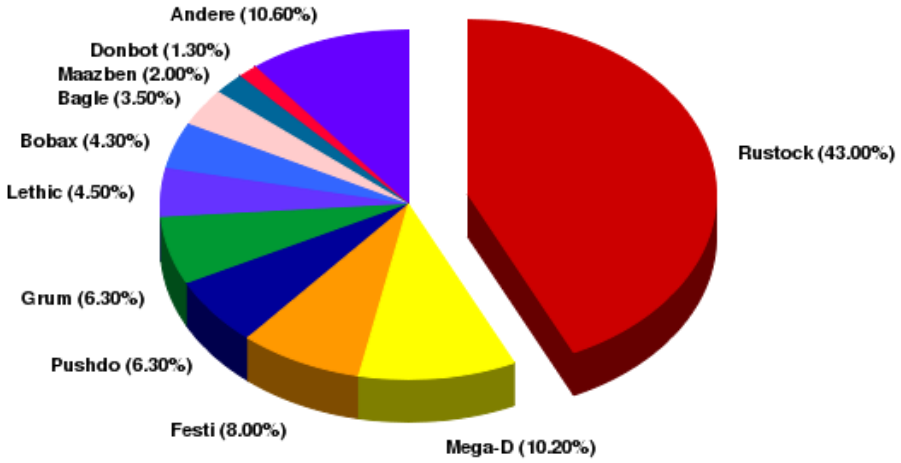


Abbildung 2.12: Die Spam-Anteile der zehn aktivsten Botnetze [M86 Security, 2010b].

Abbildung 2.12 zeigt die zehn Botnetze, die in der ersten Jahreshälfte 2010 für fast 90 Prozent aller von Botnetzen versendeten Spam-Nachrichten verantwortlich waren. Mit 43 Prozent stieß das RUSTOCK-Botnetz in diesem Zeitraum noch den mit Abstand meisten Spam aus. Obwohl die Infrastruktur des RUSTOCK-Botnetzes immer noch voll funktionsfähig scheint, wurden ohne ersichtlichen Grund seit September 2010 keine Spam-Nachrichten mehr von den Bots verschickt [Hay, 2010, Mushtaq, 2010]. Abbildung 2.13 zeigt den abrupten Abfall des von M86-SECURITY für RUSTOCK gemessenen Nachrichten-Ausstoßes[3].

[3] Seit Januar 2011 ist das RUSTOCK-Botnetz wieder aktiv.

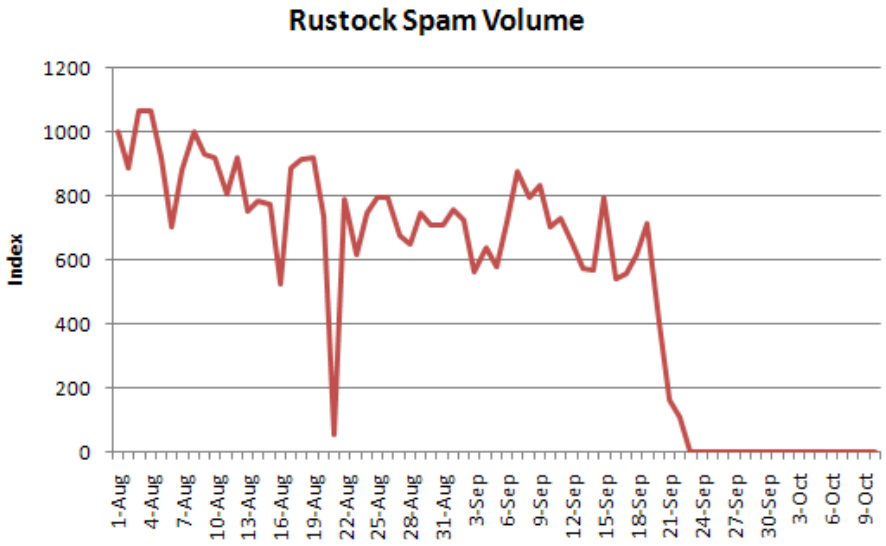


Abbildung 2.13: Um den 20. September 2010 beobachteter Abfall der aus dem RUSTOCK-Botnetz verschickten Spam-Nachrichten [Hay, 2010].

Tabelle 2.1 zeigt die Basisinformationen zu den zehn im ersten Quartal 2010 aktivsten Botnetzen [M86 Security, 2010a, Stewart, 2008, 2009]. Die meisten Botnetze sind zentralisiert, das heißt, die Kommunikation läuft über C&C-Server. Dabei setzen nur das LETIC- und das BOBAX-Botnetz eigene Protokolle ein. Die übrigen Botnetze wenden ein musterbasiertes Spamverfahren an, siehe Abschnitt 2.2.1.

Botnetz	Spam-Art	Spam pro Bot/Stunde	Botnetz-Größe	seit
RUSTOCK	Template	25.000	150.000 - 2.400.000	2005
PUSHDO	Template	4.500	1.500.000 - 2.000.000	2007
LETHIC	SOCKS	12.000 - 60.000	210.000 - 310.000	2008
GRUM	Template	4.000	600.000	2007
BOBAX	Template	7.200	100.000	2004
BAGLE	SOCKS	variabel	150.000 - 230.000	2004
FESTI	Template	nicht bekannt	nicht bekannt	2009
MEGA-D	Template	105.000	50.000	2007

Tabelle 2.1: Zusammenfassung der zentralen Eigenschaften der zehn aktuell aktivsten Spambots [Aridogan, 2010, Meinecke, 2011, Yigit, 2011].

2.3 Maschinelles Lernen

Im Bereich des Maschinellen Lernens werden Algorithmen entwickelt, die bei der Bearbeitung komplexer Aufgaben das in früheren Läufen gewonnene Wissen mit einfließen lassen. Das „erlernte Wissen“ kann somit zur Verbesserung der Algorithmen und damit auch deren Ergebnisse eingesetzt werden [Hoernlimann and Meyer, 2005].

Dabei wird auch das auf den zuvor gewonnenen Informationen basierende Anpassen von einzelnen Parametern als Lernen angesehen. Alpaydin [2004] und Mitchell [1997] definieren das Maschinelle Lernen wie folgt:

> *„Machine Learning is programming computers to optimize a performance criterion using example data or past experience.“* [Alpaydin]

> *„A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.“* [Mitchell]

Beide Definitionen fordern keine Verhaltensänderung des Systems selbst. Schon die verbesserte Performanz aufgrund von Erfahrungswerten genügt, um von einem Lerneffekt sprechen zu können.

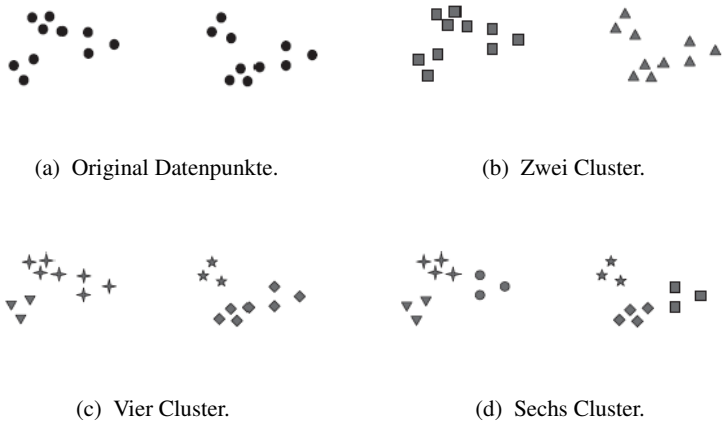
Zu den klassischen Bereichen des Maschinellen Lernens gehören die Clusteranalyse und die Klassifikation von Daten. Sowohl für die Clusteranalyse als auch für die Klassifikation ist es zwingend notwendig, Erfahrungswerte aus vorhergegangenen Analysen in den Algorithmus oder in die zentralen Parameter einfließen zu lassen.

2.3.1 Clusteranalyse

Das Ziel der Clusteranalyse ist, Daten automatisch so in Klassen oder Gruppen – sogenannte *Cluster* – einzuteilen, dass die Objekte eines Clusters möglichst ähnlich, Objekte aus verschiedenen Clustern dagegen möglichst unähnlich zueinander sind. Die Einteilung, das sogenannte *Clustering*, basiert auf Informationen, die aus den Daten selbst gewonnen werden. Für eine sinnvolle Clusteranalyse ist daher eine genaue Definition des Ähnlichkeitsmaßes zwischen den zu verarbeitenden Objekten erforderlich. Abbildung 2.14 soll diese Problematik bei der Clusteranalyse verdeutlichen. Die dargestellten Datenpunkte (a) lassen sich auf drei verschiedene Weisen in Cluster unterteilen (b)-(c). Die Clusterzugehörigkeit wird hierbei jeweils über die Form der Datenpunkte kodiert. Die Datenpunkte lassen sich in zwei, vier oder sechs Cluster unterteilen, wobei jede der Aufteilungen nachvollziehbar wirkt. Dies verdeutlicht das Grundproblem der Clusteranalyse: Es existiert keine Definition die beschreibt wie ein Cluster aufgebaut ist und/oder wie die Zugehörigkeit von Datenpunkten bestimmt werden muss. Diese Entscheidung wird einzig durch die Daten selbst und das Ziel der Clusteranalyse bestimmt. Ohne genauere Informationen darüber, wie die Anordnung der Datenpunkte zustande kommt und nach welchen Eigenschaften die Clusterzugehörigkeit berechnet werden soll, lässt sich keine Aussage über die Güte der einzelnen Aufteilungen machen.

Ähnlichkeit zwischen Datenobjekten

Die Ähnlichkeit zwischen Datenobjekten wird in der Regel durch eine *Distanzfunktion* modelliert. Diese ist für Objektpaare definiert und spiegelt den Abstand zwischen den beiden Objekten wieder [Ester and Sander, 2000]. Dabei werden die Objekte bei kleinen Abständen als ähnlich und bei großen Abständen als unähnlich interpretiert. Unter Verwendung einer *Ähnlichkeitsfunktion* ist diese Interpretation umgekehrt.



(a) Original Datenpunkte. (b) Zwei Cluster.

(c) Vier Cluster. (d) Sechs Cluster.

Abbildung 2.14: Verschiedene Möglichkeiten eine Menge von Datenpunkten in Cluster einzuteilen [Tan et al., 2005].

Die Wahl der passenden Distanzfunktion (oder Ähnlichkeitsfunktion) und der zu vergleichenden Eigenschaften hängt bei der Clusteranlayse sowohl von den Daten als auch von dem Ziel der Analyse ab. Dieser Zusammenhang lässt sich relativ einfach an der Clusteranalyse von Textdokumenten verdeutlichen. Das Ziel der Analyse bestimmt hier, welche Eigenschaften herangezogen werden müssen. Bei einer inhaltlichen Analyse der Dokumente, bei der Texte mit ähnlichem Inhalt in Cluster zusammenzufassen sind, werden die Häufigkeiten der in den Texten enthaltenen Wörter berechnet. Diese Häufigkeiten bilden einen sogenannten *Featurevektor* für das jeweilige Dokument. Über die Vektoren lassen sich die Dokumente anschließend anhand von *Ähnlichkeitsmatrizen* gruppieren. Dabei werden Dokumente, die eine Vielzahl von Wörtern miteinander teilen, als inhaltlich verwandt eingestuft. Der inhaltliche Zusammenhang der Wörter zueinander lässt sich analog analysieren, indem die *Featurevektoren* nicht für die Dokumente, sondern für die einzelnen Wörter aufgestellt werden. In der Ähnlichkeitsmatrix würden dann die Wörter, die in vielen Dokumenten gemeinsam auftreten, als inhaltlich verwandt interpretiert.

Das vorliegende Beispiel ist bewusst sehr einfach gehalten. Trotzdem verdeutlicht es den engen Zusammenhang von zu interpretierenden Eigenschaften und der verwendeten Distanz- oder Ähnlichkeitsfunktion mit dem Analyseziel. Umgekehrt hängt wiederum die Güte der Clusteranalyse stark von der Adäquatheit der Distanz- oder Ähnlichkeitsfunktion und den interpretierten Eigenschaften ab.

Clustering-Typen

Das Ergebnis einer Clusteranalyse ist eine Einteilung der Datenobjekte in verschiedene Cluster. Man unterschiedet hierbei die im Folgenden vorgestellten Verfahren [Tan et al., 2005]. Da in Kapitel 3 ein hierarchisches Clustering eingesetzt wird, wird dieser Typ genauer betrachtet.

Partitionierende, dichtebasierte und hierarchische Verfahren. Bei partitionierenden Verfahren wird die Menge der Datenobjekte in k disjunkte Cluster zerlegt, wobei jeder Cluster mindestens ein Datenobjekt enthalten muss und jedes Objekt genau zu einem Cluster gehört. Das heißt, die Anzahl der Cluster wird vorab festgelegt und eine initiale Lösung wird durch das Verschieben von Datenobjekten zwischen den Clustern weiter optimiert. Dichtebasierte Verfahren hingegen versuchen, „selbstständig“ Gebiete im Raum zu identifizieren, in denen die Datenobjekte dicht zusammen liegen, und die sich gleichzeitig klar von anderen Datenobjekten (oder Objekthäufungen) abgrenzen lassen. Ein dichtebasierter Cluster ist somit ein Gebiet, in dem die lokale *Punktdichte* der Datenobjekte – die lokale Punktdichte eines Datenobjekts entspricht der Anzahl der Objekte in einer festgelegten Umgebung um das Datenobjekt – eine vorab definierte Schranke überschreitet. Die Menge der Datenobjekte ist aufgrund der hohen Punktdichte räumlich zusammenhängend und kann dabei durchaus verschiedene Formen annehmen, siehe Abbildung 2.15.

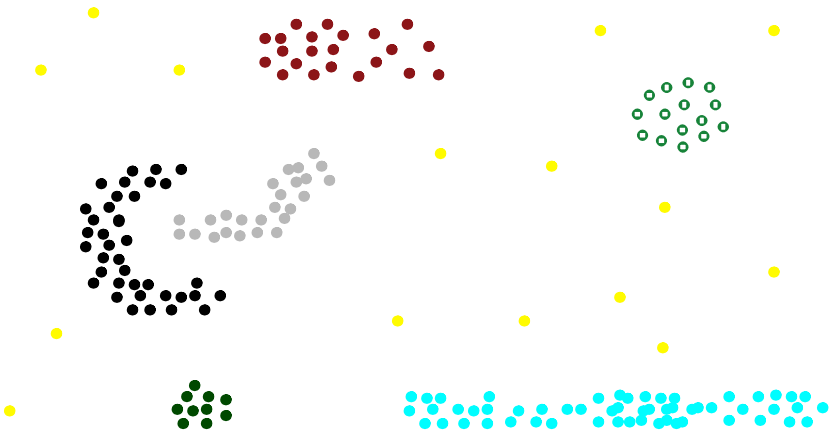


Abbildung 2.15: Beispiele für dichtebasierte Cluster.

Hierarchische Verfahren benötigen keine Vorabinformationen über die gesuchte Clusteranzahl. Stattdessen werden Cluster entweder im *bottom-up*-Verfahren miteinander verschmolzen, oder im *top-down*-Verfahren aufgesplittet. Es wird somit keine eindeutige Zerlegung der Datenmenge in mehrere Cluster berechnet, sondern eine hierarchische Repräsentation der Daten vorgenommen. Aus dieser können dann verschiedene Clusterings abgeleitet werden. Die hierarchische Repräsentation wird in der Regel als *Dendrogramm* dargestellt, siehe Abbildung 2.16. Dabei handelt es sich um einen Baum, der die hierarchische Zerlegung der Datenmenge in Teilmengen visualisiert. Die Blätter des Baumes entsprechen den einzelnen Datenobjekten und jeder Knoten (Cluster) ist die Vereinigung aller seiner Nachfolger (Subcluster). Die Wurzel des Baumes entspricht schließlich dem Cluster, der alle Datenobjekte enthält.

Ein bekanntes und einfaches hierarchisches Clusterverfahren ist die sogenannte *Single-Link-Methode*. Diese Methode benötigt neben den paarweisen Distanzen zwischen den einzelnen Datenobjekten eine Distanzfunktion für den Abstand zwischen Mengen von Objekten. Um den Abstand zwischen zwei Mengen X und Y zu bestimmen, betrachtet man in der Regel die Abstände zwischen den Objekten aus X und Y.

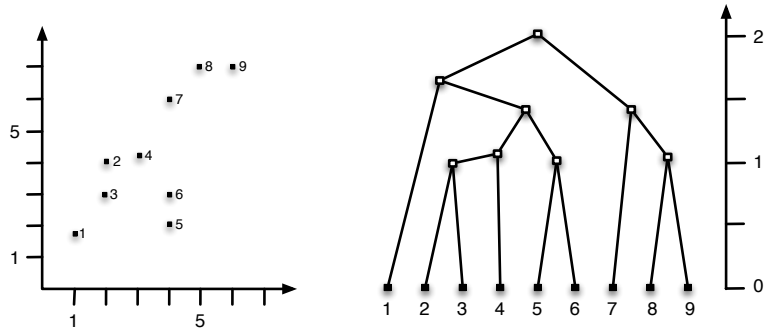


Abbildung 2.16: Single-Link-Dendrogramm. [Ester and Sander, 2000].

Bei der Single-Link-Methode wird dabei der minimale Abstand zwischen den Objekten aus den beiden Mengen herangezogen:

$$d_{single\_link}(X,Y) = \min_{x \in X,\ y \in Y} dist(x,y). \tag{2.1}$$

Der Wert $dist(x,y)$ entspricht dabei der Distanzfunktion für einzelne Datenobjekte. Wird statt der minimalen Distanz zwischen den Objekten zweier Mengen der maximale oder durchschnittliche Abstand als Distanzfunktion zwischen den Mengen verwendet, spricht man vom *Complete-Link* respektive *Average-Link* Clustering. Die Distanzfunktionen dieser beiden Methoden lauten wie folgt:

$$d_{complete\_link}(X,Y) = \max_{x \in X,\ y \in Y} dist(x,y). \tag{2.2}$$

$$d_{average\_link}(X,Y) = \frac{1}{|X| \cdot |Y|} \cdot \sum_{x \in X,\ y \in Y} dist(x,y). \tag{2.3}$$

Exklusive, überlappende und unscharfe Verfahren. Wird jedes Datenobjekt höchstens einem Cluster zugeordnet, spricht man von einem exklusiven Clustering. Bei den in den Abbildungen 2.14 und 2.15 dargestellten Zerlegungen handelt es sich ausschließlich um exklusive Clusterings. Ein überlappendes Clustering entsteht, wenn zumindest einzelne Datenobjekte mehreren Clustern zugeordnet werden. Überlappende Clusterings entstehen immer dann, wenn einzelne Objekte gleichzeitig mehreren Gruppen angehören können. Beispielsweise lassen sich Schuhe in die überlappenden Cluster Lederschuhe, Stoffschuhe, Winterschuhe, Turnschuhe, Stiefel etc. einordnen.

Während ein Objekt beim überlappenden Clustering „eindeutig“ Element der verschiedenen Cluster ist, werden sie beim *unscharfen* Clustering nur mit bestimmten Gewichten den Clustern zugeordnet. Initial ist dabei jedes Datenobjekt auch Element jedes Clusters. Das vergebene Gewicht zwischen 0 – „nicht Element des Clusters“ – und 1 – „eindeutig Element des Clusters“ – zeigt an, wie stark die Zugehörigkeit ist. Die Gewichte jedes Objekts müssen sich über die einzelnen Cluster zu 1 aufsummieren. Im Gegensatz zu dem überlappenden Clustering lässt sich das unscharfe Clustering relativ einfach in ein exklusives Clustering überführen. Dazu wird jedes Datenobjekt

ausschließlich dem Cluster zugeordnet in welchem es das größte Gewicht besitzt. Diese „Vereinfachung“ ist beim überlappenden Clustering in der Regel nicht möglich, da das Objekt zu 100 Prozent mehreren Clustern angehört.

Vollständige und partielle Verfahren. Beim vollständigen Clustering wird jedes Datenobjekt mindestens einem Cluster zugeordnet. Diese vollständige Zuordnung macht unter Umständen einelementige Cluster für sogenannte *Ausreißer* – Datenobjekte, die keinem anderen Datenobjekt zugeordnet werden können – notwendig. Der Informationsgehalt solcher Cluster ist im Allgemeinen relativ gering. In vielen Fällen bietet es sich daher an, auf diese kleinen Cluster zu verzichten und stattdessen die nicht geclusterten Datenobjekte zurückzuhalten und beispielsweise in einem iterativen Clusterverfahren, zusammen mit zusätzlichen Datenobjekten, nachträglich erneut zu clustern. Verfahren bei denen nicht zwingend alle Datenobjekte einem Cluster zugeordnet werden müssen, bezeichnet man als partiell.

2.3.2 Klassifikation

Während bei der Clusteranalyse die gesuchten Cluster vorab unbekannt sind und damit neues Wissen aus den Daten selbst gewonnen wird, wird die Klassifikation eingesetzt, um unbekannte Datenobjekte bereits bekannten Klassen mit ebenfalls bekannten Eigenschaften zuzuordnen. Bei der Klassifikation wird somit weniger neues Wissen aus den zu klassifizierenden Datenobjekten extrahiert, als vielmehr bereits bekanntes Wissen – in Form eines *Klassifikators* – auf diese angewandt. Um einen Klassifikator erstellen zu können, müssen vorab die Eigenschaften von Klassen erlernt werden. Dazu wird ein Trainigsdatensatz von bereits klassifizierten Datenobjekten herangezogen und ausgewertet. Im Idealfall identifiziert der Klassifikator alle für die jeweiligen Klassen charakteristischen und/oder diskriminativen Merkmale und verwendet diese zur Zuordnung neuer Datenobjekte. Grundsätzlich besitzt die Klassifikation damit zwei Aufgaben [Ester and Sander, 2000]:

1. Generierung von Klassifikationswissen, sowie
2. Klassifizieren von unbekannten Datenobjekten in bekannte Klassen.

Bewertung von Klassifikatoren

Aus einer Menge von Trainingsdaten können in Abhängigkeit von den verwendeten Verfahren und den zugrunde gelegten Parametern meist mehrere verschiedene Klassifikatoren gelernt werden. Umso wichtiger ist es, ein Maß zur Bewertung von Klassifikatoren zu definieren, mit dem sich die Leistungsfähigkeit der verschiedenen Klassifikatoren bestimmen und vergleichen lässt. Zur Leistungsbestimmung von Klassifikatoren wird neben den Trainingsdaten immer auch ein zweiter, markierter Datensatz benötigt, auf dem der Klassifikator getestet wird. Die Datenobjekte dieser Testmenge werden dem Klassifikator in der Lernphase vorenthalten und ihm erst im Anschluss daran zur Klassifikation vorgelegt. Die Anzahl der dabei falsch klassifizierten Objekte bestimmt den *Klassifikationsfehler* eines Klassifikators und kann als *Gütemaß* für diesen verwendet werden.

Konfusionsmatrix. Die Güte eines Klassifikators soll ausdrücken, wie gut dieser die Klassen für Datenobjekte vorhersagen kann. Dazu werden die folgenden vier Häufigkeiten in Abhängigkeit für die jeweils untersuchte Klasse berechnet und in einer sogenannten *Konfusionsmatrix* zusammenfasst:

- **richtig positiv**: Das Datenobjekt ist Element der untersuchten Klasse und wurde von dem Klassifikator auch entsprechend zugeordnet.
- **falsch positiv**: Das Datenobjekt gehört nicht zu der untersuchten Klasse, wurde dieser von dem Klassifikator aber fälschlicherweise zugeordnet.
- **falsch negativ**: Das Datenobjekt gehört nicht zur untersuchten Klasse und wurde vom Klassifikator auch nicht fälschlich zugeordnet.
- **richtig negativ**: Das Datenobjekt ist Element der untersuchten Klasse, wurde von dem Klassifikator aber nicht als solches erkannt.

Die Zusammenhänge der vier Häufigkeiten werden im Folgenden an der in Abbildung 2.17(a) dargestellten Konfusionsmatrix eines einfachen Drei-Klassen-Problems erläutert.

		vorhergesagte Klassen		
		A	**B**	**C**
bekannte Klassen	**A**	25	5	2
	B	3	32	4
	C	1	0	16

(a)

		vorhergesagte Klassen		
		A	**B**	**C**
bekannte Klassen	**A**	tp_A	e_{AB}	e_{AC}
	B	e_{BA}	tp_B	e_{BC}
	C	e_{CA}	e_{CB}	tp_C

(b)

Abbildung 2.17: Die Konfusionsmatrix für ein Drei-Klassen-Problem (a) und die Bedeutung der einzelnen Einträge (b).

Die Zeilen der Matrix entsprechen der korrekten Klassifizierung, und in den Spalten sind die jeweiligen Vorhersagen des Klassifikators aufgetragen. Die auf der Diagonalen eingetragenen Werte entsprechen somit der Anzahl der korrekt klassifizierten Datenobjekte pro Klasse. Alle übrigen Werte zeigen Fehlklassifizierungen an. Im vorliegenden Fall wurden zum Beispiel 25 Datenobjekte der Klasse A korrekt klassifiziert. Weiter 5 Objekte wurden fälschlich der Klasse B zugeordnet und zwei Objekte fälschlich der Klasse C. In Abbildung 2.17(b) sind die einzelnen Felder der Konfusionsmatrix nochmals abstrakt beschrieben. Die *richtig positiv* Einträge für die einzelnen Klassen sind darin mit tp_{class} gekennzeichnet. Die $e_{class1class2}$ Einträge geben die Anzahl der Fehlklassifikationen im Bezug auf $class1$ an.

Grundsätzlich existieren verschiedene *Gütemaße*, mit denen sich Klassifikatoren basierend auf den Daten der Konfusionsmartix beurteilen lassen. Im Folgenden werden mit *Recall* und *Precision* zwei Maße vorgestellt, die im Bereich des *Information Retrieval* vielfach zur Beurteilung der Güte von *Information Retrieval* Systemen herangezogen werden [Womser-Hacker, 1989] und sich sehr gut zur Beurteilung von Klassifikatoren eignen. Recall und Precision werden auch zur Evaluierung der in Kapitel 3 eingesetzten Klassifizierung verwendet. Das anschließend vorgestellte $F1$-Maß stellt lediglich eine Kombination der beiden vorherigen Maße dar [van Rijsbergen, 1979]. Zum besseren Verständnis werden die drei Maße auch jeweils an dem in Abbildung 2.17 dargestellten Beispiel nachvollzogen.

Recall. Recall – auf Deutsch: *Trefferquote* oder *Sensitivität* – stellt ein Maß für die Vollständigkeit der Klassifizierung, bezogen auf eine konkrete Klasse, dar. Es ist als das Verhältnis zwischen der Anzahl richtig klassifizierter Datenobjekte (richtig positiv) und der Anzahl aller zu dieser Klasse gehörenden Datenobjekte des Datensatzes (richtig positiv + falsch negativ) definiert.

$$R = \frac{|\,\textit{true positive}\,|}{|\,\textit{true positive}\,| \;+\; |\,\textit{false negative}\,|} \tag{2.4}$$

Für die in Abbildung 2.17 dargestellte Konfusionsmatrix berechnet sich das Recall-Maß für die Klasse A somit wie folgt:

$$R_A = \frac{tp_A}{tp_A \;+\; e_{AB} \;+\; e_{AC}} = \frac{25}{25 \;+\; 5 \;+\; 2} \approx 0.78 \tag{2.5}$$

Der Wertebereich des Recall-Maßes liegt zwischen 0 und 1, wobei 0 für das schlechteste und 1 für das bestmögliche Ergebnis vergeben wird.

Precision. Precision – auf Deutsch: *Genauigkeit* oder *Relevanz* – liefert die Genauigkeit, mit der eine bestimmte Klasse durch den Klassifikator erkannt wird. Dazu wird die Anzahl der richtig klassifizierten Datenobjekte (richtig positiv) mit der Anzahl aller der Klasse zugeordneten Datenobjekte (richtig positiv + falsch positiv) ins Verhältnis gesetzt.

$$P = \frac{|\,\textit{true positive}\,|}{|\,\textit{true positive}\,| \;+\; |\,\textit{false positive}\,|} \tag{2.6}$$

Für obiges Drei-Klassen-Problem ergibt sich damit das folgende Precision-Maß für die Klasse A:

$$P_A = \frac{tp_A}{tp_A \;+\; e_{BA} \;+\; e_{CA}} = \frac{25}{25 \;+\; 3 \;+\; 1} \approx 0.86 \tag{2.7}$$

Auch der Wertebereich des Precision-Maßes liegt zwischen 0 und 1. Das schlechteste Ergebnis wird auch hier mit 0 und das bestmögliche mit 1 bewertet.

$F1$-Maß. Das $F1$-Maß stellt eine Kombination von Recall und Precision dar. Da die beiden Maße miteinander konkurrieren, ist in der Praxis nur die gemeinsame Betrachtung beider Maße sinnvoll. Einzeln betrachtet lassen sich beide Maße einfach maximieren, indem entweder alle oder nur wenige Datenobjekte der untersuchten Klasse zugeordnet werden. Die erste Variante führt zu einem Recall-Wert von 1 und die zweite maximiert den Precision-Wert. Dabei wird jedoch das jeweils andere Maß minimiert.

Im $F1$-Maß werden das Recall- und das Precision-Maß wie folgt kombiniert:

$$F = \frac{2 \cdot P \cdot R}{P + R} \tag{2.8}$$

Eine perfekte Klassifizierung liefert dabei einen Wert von 1, während niedrige Recall- oder Precision-Werte auch zu einem niedrigen $F1$-Maß führen.

2.4 Formale Sprachen

Die in diesem Abschnitt vorgestellten Zusammenhänge beruhen auf den Arbeiten von Schöning [2009] und Der [2001]. Insbesondere die dargestellten Definitionen wurden von Schöning [2009] übernommen.

Sprachen und Grammatiken. Die in der Informatik zur Beschreibung von Daten, Informationen und Programmen verwendeten Sprachen werden als *künstliche*, oder auch *formale* Sprachen bezeichnet. Diese lassen sich einfacher verarbeiten als natürliche Sprachen, da die zulässigen Ausdrücke und deren Bedeutung exakt definiert sind. Beispielsweise erlauben *Programmiersprachen* und *Maschinensprachen* nur einen bestimmten Satz von Zeichen und Symbolen. Jedes Zeichen besitzt dabei eine eindeutige Bedeutung, die dem „Computer“ bekannt ist. Dadurch ist es diesem möglich, das Programm zu „verstehen“ und die gewünschte Funktionalität auszuführen.

Eine (formale) Sprachen L beschreibt somit eine beliebige Teilmenge von Wörtern über einem Alphabet Σ. Die formale Definition einer Sprache lautet wie folgt:

Definition 2.4.1 (Formale Sprache)**.** *Sei Σ ein Alphabet und Σ^* die Menge aller Wörter, die sich durch die Konkatenation von Symbolen aus Σ bilden lässt. Eine (formale) Sprache über Σ ist jede beliebige Teilmenge von Σ^*.*

Da der Konkatenation von Symbolen formal – noch – keine Beschränkungen auferlegt sind, können Sprachen durchaus unendliche Objekte darstellen. Um trotzdem mit diesen arbeiten zu können, existieren mit *Automaten* und *Grammatiken* zwei Beschreibungsformen, mit denen sich auch unendliche Sprachen endlich beschreiben lassen. Auf die Automaten zur Beschreibung von Sprachen wird an dieser Stelle verzichtet, da im Rahmen dieser Arbeit ausschließlich Grammatiken verwendet werden.

Die in Abbildung 2.18 dargestellte Grammatik definiert eine sehr stark eingeschränkte natürliche Sprache. Die formal als *Variablen* bezeichneten Platzhalter für die Wörter sind darin durch spitze Klammern kenntlich gemacht. Die Konstruktion eines Satzes geht immer von dem sogenannten *Startsymbol* $\langle Satz \rangle$ aus und die sogenannte *Terminalsymbole*, also die eigentlichen Wörter aus denen sich Sätze erstellen

$$
\begin{array}{rcl}
\langle Satz \rangle & \rightarrow & \langle Subjekt \rangle \ \langle Prädikat \rangle \ \langle Objekt \rangle \\
\langle Subjekt \rangle & \rightarrow & \langle Artikel \rangle \ \langle Attribut \rangle \ \langle Substantiv \rangle \\
\langle Artikel \rangle & \rightarrow & \epsilon \\
\langle Artikel \rangle & \rightarrow & \text{der} \\
\langle Artikel \rangle & \rightarrow & \text{die} \\
\langle Artikel \rangle & \rightarrow & \text{das} \\
\langle Attribut \rangle & \rightarrow & \epsilon \\
\langle Attribut \rangle & \rightarrow & \langle Adjektiv \rangle \\
\langle Attribut \rangle & \rightarrow & \langle Adjektiv \rangle \ \langle Attribut \rangle \\
\langle Adjektiv \rangle & \rightarrow & \text{klein} \\
\langle Adjektiv \rangle & \rightarrow & \text{bissig} \\
\langle Adjektiv \rangle & \rightarrow & \text{große} \\
\langle Substantiv \rangle & \rightarrow & \text{Hund} \\
\langle Substantiv \rangle & \rightarrow & \text{Katze} \\
\langle Prädikat \rangle & \rightarrow & \text{jagt} \\
\langle Objekt \rangle & \rightarrow & \langle Artikel \rangle \ \langle Attribut \rangle \ \langle Substantive \rangle
\end{array}
$$

Abbildung 2.18: Die Grammatik einer einfachen natürlichen Sprache.

lassen, sind *der, die, das, kleine, bissige, große, Hund* und *Katze*. Über die dargestellten Regeln der Grammatik lässt sich beispielsweise der Satz

der kleine bissige Hund jagt die große Katze

ableiten. Zum einfacheren Verständnis kann die Ableitung eines Satze auch graphisch, durch einen sogenannten *Syntaxbaum* oder auch *Ableitungsbaum*, darstellen werden (Abbildung 2.19). Die Wurzel des Baums ist dabei mit dem Startsymbol markiert und die Blätter des Baumes stellen die Terminalsymbole dar. Jeder innere, mit einer Variablen markierte, Knoten entspricht der linken Seite einer Regel und seine Kinder sind die Objekte, die auf der rechten Seite der Regel stehen.

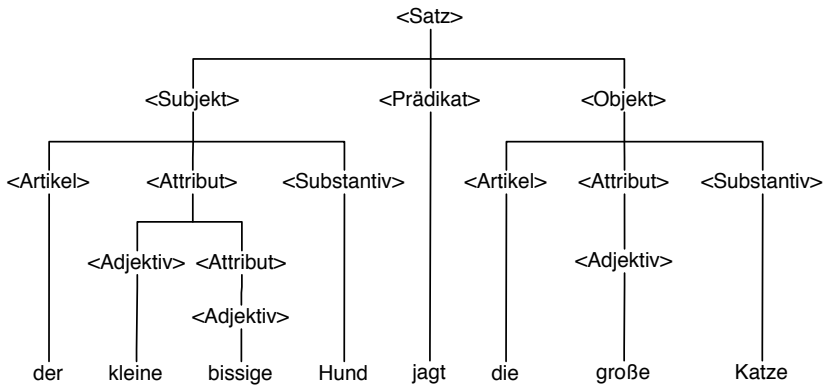


Abbildung 2.19: Syntaxbaum zur Ableitung eines Satzes [Schöning, 2009].

Aus der dargestellten Grammatik lassen sich die für eine allgemeine Definition einer Grammatik notwendigen Informationen leicht ablesen. Neben den Terminalsymbolen und Variablen sind dies die verschiedenen Ableitungsregeln und eine ausgewiesene Startvariable. Formal wird eine Grammatik wie folgt definiert:

Definition 2.4.2 (Grammatik)**.** *Eine Grammatik ist ein 4-Tupel $G = (V, \Sigma, P, S)$, das folgende Bedingungen erfüllt:*

- *V ist eine endliche Menge, die Menge der Variablen.*
- *Σ ist eine endliche Menge, das Terminalalphabet. Es muss gelten: $V \cap \Sigma = \emptyset$.*
- *P ist die endliche Menge der Regeln und Produktionen. Formal ist P eine endliche Teilmenge von $(V \cup \Sigma)^+ \times (V \cap \Sigma)^*$.*
- *$S \in V$ ist die Startvariable.*

Seien $u, v \in (V \cup \Sigma)^*$. Wir definieren die Relation $u \Rightarrow_G v$ (in Worten: u geht unter G unmittelbar über in v), falls u und v die Form haben

$$u = xyz$$
$$v = xy'z \text{ mit } x, z \in (V \cup \Sigma)^*$$

und $y \to y'$ eine Regeln in P ist.
Die von G dargestellte (erzeugte, definierte) $Sprache$ ist

$$L(G) = \{w \in \Sigma^* | S \Rightarrow_G^* w\}$$

Hierbei ist $\Rightarrow_G^*$ die reflexive und transitive Hülle von $\Rightarrow_G$.
Eine Folge von Wörtern $(w_0, w_1, ..., w_n)$ mit $w_0 = S, w_n \in \Sigma^*$ und $w_0, \Rightarrow w_1 \Rightarrow ... \Rightarrow w_n$ heißt Ableitung von w_n. Ein Wort $w \in (V \cup \Sigma)^*$, das also noch Variablen enthält – wie es typischerweise im Verlauf einer Ableitung auftritt – heißt auch Satzform.

Chomsky-Hierarchie. Die von Chomsky [1956] vorgenommene Einteilung von Grammatiken kennt vier Typen (0-3). Jeder Typ erweitert dabei die Anforderungen, die die Grammatikregeln erfüllen müssen. Das heißt, die niedrigen Grammatiken stellen jeweils eine Obermenge der höheren Grammatiken dar. Abbildung 2.20 veranschaulicht diesen Zusammenhang nochmals.

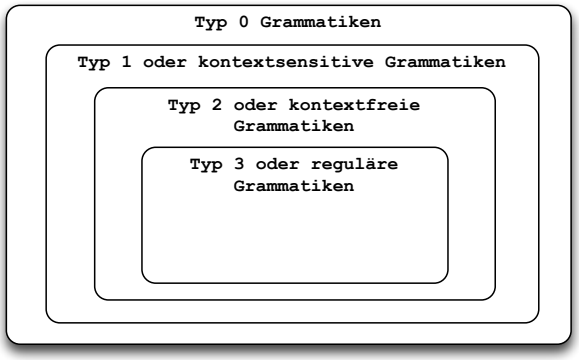


Abbildung 2.20: Grammatik-Typen nach Chromsky.

Formal sind die vier Typen wie folgt definiert.

Definition 2.4.3 (Chomsky-Hierarchie)**.** *Jede Grammatik ist zunächst automatisch vom Typ 0. Das heißt, bei Typ 0 sind den Regeln keinerlei Einschränkungen auferlegt.*

Eine Grammatik ist vom Typ 1 oder kontextsensitiv, falls für alle Regeln $w_1 \to w_2$ in P gilt: $|w_1| \leq |w_2|$.

Eine Typ 1-Grammatik ist vom Typ 2 oder kontextfrei, falls für alle Regeln $w_1 \to w_2$ in P gilt, dass w_1 eine einzelne Variable ist, das heißt $w_1 \in V$.

Eine Typ 2-Grammatik ist vom Typ 3 oder regulär, falls zusätzlich gilt: $w_2 \in \Sigma \cup \Sigma V$, das heiß, die rechten Seiten von Regeln sind entweder einzelne Terminalsymbole oder ein Terminalsymbol gefolgt von einer Variablen.

Eine Sprache $L \subseteq \Sigma^*$ heißt vom Typ 0 (Typ 1, Typ 2, Typ 3), falls es eine Typ 0 (Typ 1, Typ 2, Typ 3)-Grammatik G gibt mit $L(G) = L$.

Die für die Typ 1- und Typ 2-Grammatiken verwendeten Bezeichnungen *kontextsensitiv* beziehungsweise *kontextfrei* lassen sich dabei wie folgt herleiten: Die kontextfreie Regel $A \to x$ ermöglicht das Ersetzen der Variablen A durch x. Der Kontext, in dem A steht wird dabei nicht beachtet. Eine kontextsensitive Grammatik erlaubt Regeln, die explizit den Kontext von Variablen berücksichtigen. Beispielsweise kann die Regel $uAv \to uxv$ nur angewendet werden, wenn die Variable A zwischen den Terminalsymbolen u und v steht. Mit Hilfe der kontextsensitiven Grammatiken lassen sich damit weit komplexere Sprachen beschreiben, als dies mit kontextfreien, oder regulären Grammatiken möglich wäre.

Das Wortproblem. Für die Anwendung von Grammatiken, beispielsweise zum Filtern von Schadprogrammen (Kapitel 3) oder Spam-Nachrichten (Kapitel 4), ist es wichtig, dass in endlicher Zeit entschieden werden kann, ob ein Wort Element der von der Grammatik definierten Sprache ist, oder nicht. Dieses Problem wird als *Wortproblem* bezeichnet und ist für Typ 1-Grammatiken (und damit auch für Typ 2, Typ 3-Sprachen) entscheidbar.

Satz 2.4.1 (Wortproblem)**.** *Es gibt einen Algorithmus, der bei Eingabe einer kontextsensitiven Grammatik $G = (V, \Sigma, P, S)$ und eines Wortes $x \in \Sigma^*$ in endlicher Zeit entscheidet, ob $x \in L(G)$ oder $x \notin L(G)$.*

Intuitiv lässt sich die Entscheidbarkeit des Wortproblems schon aus der Definition für kontextsensitive Grammatiken folgern. Diese verlangt, dass die linke Seite einer Regel höchstens so lang sein darf wie die rechte Seite der Regel. Folglich können beim Herleiten eines beliebigen Wortes die „Zwischenergebnisse“ (die Satzformen) nur länger werden. Gleichzeitig sind sie aber durch die Länge n des herzuleitenden Wortes beschränkt. Da nur endlich viele Wörter über $(V \cup \Sigma)^*$ der Länge $\leq n$ existieren, lässt sich durch systematisches Durchprobieren in endlicher Zeit entscheiden, ob ein gegebenes Wort in $L(G)$ liegt [Schöning, 2009].

Backus-Naur-Formen. Die in den folgenden Kapiteln vorgestellten Grammatiken werden in der *Backus-Naur-Form* formuliert [Backus et al., 1963]. Diese Schreibweise

erlaubt eine kurze und übersichtliche Darstellung kontexfreier Grammatiken. Dabei werden mehrere Regeln, die dieselbe linke Seite haben, in einer einzelnen „Metaregel“ zusammengefasst.

⟨*Satz*⟩	:	⟨*Subjekt*⟩ ⟨*Prädikat*⟩ ⟨*Objekt*⟩
⟨*Subjekt*⟩	:	⟨*Artikel*⟩ ⟨*Attribut*⟩ ⟨*Substantiv*⟩
⟨*Artikel*⟩	:	der
	|	die
	|	das
	|	ϵ
⟨*Attribut*⟩	:	⟨*Adjektiv*⟩ ⟨*Attribut*⟩
	|	⟨*Adjektiv*⟩
	|	ϵ
⟨*Adjektiv*⟩	:	kleine
	|	bissig
	|	große
⟨*Substantiv*⟩	:	Hund
	|	Katze
⟨*Prädikat*⟩	:	jagt
⟨*Objekt*⟩	:	⟨*Artikel*⟩ ⟨*Attribut*⟩ ⟨*Substantive*⟩

Abbildung 2.21: Die BNF einer einfachen natürlichen Sprache.

Die BNF der obigen Grammatik ist in Abbildung 2.21 dargestellt. Die Variablen sind in dieser Notation durch spitze Klammern ⟨ ... ⟩ gekennzeichnet und die durch eine Regel definierte Ableitung wird durch einen Doppelpunkt dargestellt[4]. Alternativen beim Ableiten werden durch einen senkrechten Stich abgetrennt.

Der in Abschnitt 2.5 vorgestellte Parsergenerator PLY arbeitet auf einer BNF-Notation für kontextfreie Grammatiken. Aus dieser wird ein *endlicher Automat* konstruiert, der die Grammatik erkennt.

2.5 Lex und Yacc

Mit Hilfe der beiden Programme LEX und YACC lassen sich *Parser* für kontextfreie Grammatiken erstellen. Diese werden beispielsweise bei der Entwicklung von *Compilern* für Programmiersprachen eingesetzt. Die dabei notwendige lexikalische und syntaktische Analyse des Quellcodes wird von einem *Lexer* beziehungsweise *Parser* übernommen. Im Rahmen dieser Arbeit wird die PYTHON-Implementierung PLY der beiden Werkzeuge LEX und YACC eingesetzt. Die Arbeitsweise der Werkzeuge und deren Verwendung wird im Folgenden an dem bereits bei der Einführung von Grammatiken verwendeten Beispiel dargestellt. Dabei werden nur die zum Verständnis dieser Arbeit notwendigen Grundlagen vorgestellt. Für eine detaillierte Beschreibung von PLY wird auf Beazley [2009] verwiesen.

[4] Backus und Naur verwenden ursprünglich keinen einfachen Doppelpunkt sondern die Zeichenfolge „::==“.

2.5.1 Lexikalische Analyse

Bei der lexikalischen Analyse wird der übergebene Eingabestrom, in der Regel ist dies der Quellcode eines zu untersuchenden Programms, in einzelne Symbole (*Tokens*) zerlegt. Dabei unterscheidet der *Lexer* zwischen den in der Syntax der Sprache definierten Termialsymbolen und Variablen.

Die Grammatik der Beispiel-Sprache kennt mit *der, die, das, klein, bissig, große, Hund, Katze, jagt* neun Terminalsymbole, die sich auf vier Variablen verteilen. In LEX werden diese Variablen als Token bezeichnet und jeweils über reguläre Ausdrücke definiert. Damit ergeben sich die in Listing 2.3 dargestellten vier Token und die dazugehörigen Token-Definitionen.

```
# Tokenliste
tokens = ('ARTIKEL',
          'ADJEKTIV',
          'SUBSTANTIV',
          'PRAEDIKAT',
          )

# Regulaere Ausdruecke zur Definiton der Token
t_ARTIKEL    = r'der|die|das'
t_ADJEKTIV   = r'kleine|bissige|grosse'
t_SUBSTANTIV = r'Hund|Katze'
t_PRAEDIKAT  = r'jagt'
```

Listing 2.3: Definition der Tokenliste und der einzelnen Token in PLY.

Der im letzten Absatz abgeleitete Satz

der kleine bissige Hund jagt die große Katze

wird von dem Lexer in die folgenden Token zerlegt:

```
(ARTIKEL, 'der'), (ADJEKTIV, 'kleine'), (ADJEKTIV, 'bissige'),
(SUBSTANTIV, 'Hund'), (PRAEDIKAT, 'jagt'), (ARTIKEL, 'die'),
(ADJEKTIV, 'Katze')
```

Damit ist die lexikalische Analyse abgeschlossen. Sämtliche Wörter sind bekannt und können zu Token abgeleitet werden.

2.5.2 Syntaktische Analyse

Die syntaktische Analyse einer Eingabe erfolgt durch den Parser. Dieser arbeitet nicht direkt auf dem Eingabestrom, sondern auf der durch den Lexer erstellten Token-Folge. Dabei wird untersucht, ob diese Folge syntaktisch richtig ist, das heißt, ob sich die Token-Folge, ausgehend von der Startvariable, aus den Grammatikregeln herleiten lässt.

Die Grammatikregeln werden in YACC – die Bezeichnung ist ein Akronym für *Yet Another Compiler Compiler* – in einer BNF-ähnlichen Notation angegeben. In der PYTHON-Implementierung PLY wird dabei jede Regel in einer eigenen Funktion definiert.

```
# Funktionen fuer die Grammatikregeln
def p_satz(p):
    '''satz : subjekt PRAEDIKAT objekt'''
    p[0] = p[1] + " " + p[1] + " " + p[3]
    pass

def p_subjekt(p):
    '''subjekt : ARTIKEL attribut SUBSTANTIV
               | attribut SUBSTANTIV'''
    if len(p) == 4:
        p[0] = p[1] + " " + p[2] + " " + p[3]
    else:
        p[0] = p[1] + " " + p[2]
    pass

def p_attribut(p):
    '''attribut : ADJEKTIV attribut
                | ADJEKTIV
                | '''
    if len(p) == 2:
        p[0] = p[1]
    elif len(p) == 3:
        p[0] = p[1] + " " + p[2]
    pass

def p_objekt(p):
    '''objekt : ARTIKEL attribut SUBSTANTIV
              | attribut SUBSTANTIV'''

    if len(p) == 4:
        p[0] = p[1] + " " + p[2] + " " + p[3]
    else:
        p[0] = p[1] + " " + p[2]
    pass
```

Listing 2.4: Definition von Grammatikregeln in PLY.

Für die definierenden Grammatikregeln ergeben sich die in Listing 2.4 dargestellten Funktionen. Die einzelnen Regeln der Grammatik sind in den jeweiligen Funktionen in einer BNF-verwandten Notation angegeben. Daran schließt sich jeweils die gewünschte Verarbeitung der Token-Werte an. Diese wird auch als *action code* bezeichnet. Im vorliegenden Beispiel werden die Werte der Token beim Ableiten einfach konkateniert. Das Ergebnis einer erfolgreichen Ableitung eines Satzes entspricht somit dem Satz selbst.

2.5.3 Ableiten mit PLY

Beim Ableiten eines Satzes verwendet YACC das sogenannte *LR-Parsing*, oder auch *shift-reduce Parsing*. Diese *bottom-up*-Technik versucht in dem Eingabestrom rechte Seiten von Grammatikregeln zu identifizieren, leitet diese jeweils zu der linken Seite ab und führt die entsprechende Aktion aus.

Stack	→ *Aktion*
. LexToken(ARTIKEL, 'der')	→ *Shift ARTIKEL*
ARTIKEL . LexToken(ADJEKTIV, 'kleine')	→ *Shift ADJEKTIV*
ARTIKEL ADJEKTIV . LexToken(ADJEKTIV, 'bissige')	→ *Shift ADJEKTIV*
ARTIKEL ADJEKTIV ADJEKTIV . LexToken(SUBSTANTIV, 'Hund')	→ *Reduce [attribut : ADJEKTIV]*
ARTIKEL ADJEKTIV attribut . LexToken(SUBSTANTIV, 'Hund')	→ *Reduce [attribut : ADJEKTIV attribut]*
ARTIKEL attribut . LexToken(SUBSTANTIV, 'Hund')	→ *Shift SUBSTANTIV*
ARTIKEL attribut SUBSTANTIV . LexToken(PRAEDIKAT, 'jagt')	→ *Reduce [subjekt : ARTIKEL attribut SUBSTANTIV]*
subjekt . LexToken(PRAEDIKAT, 'jagt')	→ *Shift PRAEDIKAT*
subjekt PRAEDIKAT . LexToken(ARTIKEL, 'die')	→ *Shift ARTIKEL*
subjekt PRAEDIKAT ARTIKEL . LexToken(ADJEKTIV, 'grosse')	→ *Shift ADJEKTIV*
subjekt PRAEDIKAT ARTIKEL ADJEKTIV . LexToken(SUBSTANTIV, 'Katze')	→ *Reduce [attribut : ADJEKTIV]*
subjekt PRAEDIKAT ARTIKEL attribut . LexToken(SUBSTANTIV, 'Katze')	→ *Shift SUBSTANTIV*
subjekt PRAEDIKAT ARTIKEL attribut SUBSTANTIV . $end	→ *Reduce [objekt : ARTIKEL attribut SUBSTANTIV]*
subjekt PRAEDIKAT objekt . $end	→ *Reduce [satz : subjekt PRAEDIKAT objekt]*
satz . $end	→ *Fertig! Ausgabe: 'der kleine bissige Hund jagt die grosse Katze'*

Tabelle 2.2: Ableitung eines Satzes mit PLY.

Zum Ableiten wird die Eingabe nach und nach auf einen *Stack* geschoben (*shift*). Auf diesem werden die Wörter in Token übersetzt und – falls möglich – wird eine Grammatikregel angewendet (*reduce*). Tabelle 2.2 zeigt, wie die einzelnen Token beim Parsen des Satzes „*der kleine bissige Hund jagt die grosse Katze*“ auf den Stack gelegt und anschließend über die Regeln abgeleitet werden.

Der Inhalt des Stack ist jeweils linksbündig und der vom Parser durchgeführte Schritt rechtsbündig dargestellt. Der Parser schiebt solange neue Token auf den Stack, bis die rechte Seite einer Grammatikregel auf dem Stack liegt. Ist dies der Fall, werden die Token der rechten Seite durch die linke Seite der entsprechenden Regel ersetzt. Anschließend werden – falls notwendig – die nächsten Token auf den Stack geschoben, bis sich wieder eine Regel anwenden lässt.

Der Quellcode für die in dem Beispiel verwendete Grammatik ist in Anhang A dargestellt. Neben dem Quellcode der Grammatik kann auch die vollständige Ausgabe des Parsers, die beim Ableiten des Satzes generiert wurde, dort nachgeschlagen werden.

2.6 Zusammenfassung

Dieses Kapitel liefert die zum Verständnis der Arbeit notwendigen Grundlagen. Nach einer Vorstellung der verschiedenen Schadprogramm-Klassen und den von diesen ausgehenden Gefahren, wurden mit der statischen und der dynamischen Analyse die beiden Grundkonzepte der Programmanalyse eingeführt. Für beide Konzepte wurden dabei auch die jeweiligen Vor- und Nachteile erläutert. Auf die dynamische Analyse von Schadprogrammen in sogenannten Sandboxen wurde im Detail eingegangen, da das in Kapitel 3 vorgestellte Analysesystem auf einer Sandbox-gestützten Verhaltensanalyse aufsetzt. Neben dem zur Überwachung eingesetzten API-Hooking, wurde hier auch der Aufbau der Analyseumgebung betrachtet. Da die Analysesysteme während der Sandboxanalyse infiziert werden, muss sichergestellt sein, dass sie nach jeder Analyse wieder in einen sauberen Zustand zurückgesetzt werden.

Da Spam einen weiteren Schwerpunkt dieser Arbeit darstellt, wurden der Begriff selbst und die von *Spammern* am häufigsten verwendeten Methoden zum Versenden der Nachrichten vorgestellt. Anschließend wurden die Arbeitsweise und Probleme klassischer Spam-Filter beschrieben. Die Auflistung und Beschreibung der zehn momentan am aktivsten spammenden Botnetze rundet den Grundlagenabschnitt zu Spam ab.

Mit der Clusteranalyse und der Klassifikation wurden zwei für diese Arbeit grundlegende Methoden des maschinellen Lernens beschrieben und voneinander abgegrenzt. Zum besseren Verständnis der Verfahren und der Unterschiede zwischen diesen, wurden beide Verfahren zusätzlich an konkreten Beispielen nachvollzogen.

Die in dieser Arbeit präsentierten Filter basieren auf regulären Ausdrücken und kontextfreien Grammatiken. Die Einführung in die formalen Sprachen liefert die theoretische Basis zum Verständnis der Filter und enthält neben den notwendigen Definitionen auch ein leicht verständliches Beispiel einer einfachen Sprache. Diese Sprache wird auch bei der Vorstellung von LEX und YACC herangezogen, um die Arbeitsweise der beiden zur lexikalischen und syntaktischen Analyse verwendeten Programme vorzustellen.

KAPITEL III

Musterbasierte Filter für Schadprogramme

Schadprogramme stellen aktuell eine der größten Herausforderungen für die Computersicherheit dar. So sehen sich Sicherheitsexperten und die Hersteller von Antiviren-Programmen täglich mit einer immensen Anzahl neuer Schadprogramme konfrontiert, für die geeignete Signaturen entwickelt werden müssen [Microsoft, 2009, Symantec, 2009]. Um mit dieser Entwicklung Schritt zu halten, werden Werkzeuge benötigt, die bekannte und im Idealfall auch unbekannte Schadprogramme mit einem Minimum an Benutzerinteraktion analysieren und klassifizieren.

Kontext. Bei der Analyse von (Schad-) Programmen unterscheidet man zwischen *statischen* und *dynamischen* Ansätzen, siehe hierzu auch Abschnitt 2.1.2. Bei der rein statischen Analyse wird versucht, aus dem Binärcode und – falls vorhanden – dem Quellcode Informationen über das Programm zu extrahieren. Das Programm selbst kommt dabei nie zur Ausführung. Die dynamische Analyse hingegen untersucht das Verhalten des Programms zur Laufzeit. Dazu wird das Programm in einer kontrollierten Umgebung ausgeführt und es werden sämtliche Operationen protokolliert.

Beide Analyseformen besitzen verschiedene Vor- und Nachteile, die in Abschnitt 2.1.2 diskutiert werden. Zu dem in diesem Kapitel vorgestellten musterbasierten Filteransatz führten dabei die folgenden zwei grundlegenden Eigenschaften der dynamischen Programmanalyse:

1. Die dynamische Programmanalyse lässt sich unter Verwendung von sogenannten Sandbox Systemen vollständig automatisieren.
2. Die bei der Sandbox-Analyse generierten Verhaltensreporte liefern keine definitive Aussage darüber, ob das analysierte Programm ein Schadprogramm ist.

Dank der sehr guten Automatisierbarkeit der dynamischen Analyse lassen sich praktisch alle Schadprogramme detailliert analysieren. Damit stellt dieser Ansatz die einzige Möglichkeit dar, mit der stetig steigenden Anzahl an neuen Schadprogrammen Schritt halten zu können. Allerdings liefert die Analyse kein zufriedenstellendes Ergebnis, und letztlich muss jeder mit einem Sandbox-System erstellte Verhaltensreport von einem menschlichen Analysten gesichtet werden, um eine Interpretation des beobachteten Verhaltens zu erhalten.

Problemstellung. Um die von einem Analysten zu sichtenden Verhaltensreporte bei der ständig wachsenden Anzahl neuer Schadprogramme zu reduzieren, kann die dynamische Analyse mit Methoden des maschinellen Lernens kombiniert werden. Eine auf dem beobachteten Verhalten basierte Clusteranalyse von Schadprogrammen kann zum Beispiel dabei helfen, automatisiert neue Familien von Schadprogrammen zu erkennen. Das frühzeitige Identifizieren und Markieren neuer Familien ist essentiell, um Schadprogramme erfolgreich bekämpfen zu können. Eine Klassifizierung der Schadprogramme liefert hingegen keine Informationen über neue Familien. Hier werden stattdessen unbekannte Schadprogramme bereits bekannten Familien zugeordnet. Die Klassifikation kann somit als Filter für unbekannte Schadprogramme eingesetzt werden und reduziert die Kosten der manuellen Analyse erheblich. In Kombination mit Techniken des maschinellen Lernens kann die dynamische Verhaltensanalyse deshalb zu einer „vollwertigen“ Analysemethode ausgebaut werden. Offen bleibt, wie die verschiedenen Techniken kombiniert werden müssen und wie die Schnittstellen zwischen den beiden Feldern Schadprogrammanalyse und Maschinelles Lernen konzipiert werden sollten.

Ansatz. Für eine effektive und gleichzeitig effiziente Analyse müssen die zu verarbeiteten Daten und die verwendeten Algorithmen so aufeinander abgestimmt sein, dass diskriminative Muster für die angewandten Lernmethoden erkennbar werden. Dazu wurde eine neue Verhaltensrepräsentation – das *Malware Instruction Set* (MIST) – entwickelt, die ausschließlich eine Optimierung der Datengrundlage zur Analyse mit Techniken des *Data Mining* und maschinellen Lernens verfolgt. Die Implementierung erlaubt dabei sowohl das automatische Generieren der MIST-Reporte während der Analyse, als auch eine nachträgliche Konvertierung bereits existierender Verhaltensreporte. Da die Repräsentation nicht an einen festen Aufbau der Verhaltensreporte gebunden ist, kann MIST auch als Metasprache für Verhaltensreporte verschiedenster Analysewerkzeuge verwendet werden.

Die auf MIST basierenden Filter werden über kontextfreie Grammatiken realisiert und basieren ausschließlich auf den bei der Clusteranalyse gewonnen familienspezifischen Informationen. Damit lassen sich neue Schadprogramme nach dem Abschluss der dynamischen Analyse sofort bekannten Schadprogrammfamilien zuordnen oder als neuartig klassifizieren.

Ergebnisse. Durch den Einsatz des *Malware Instruction Set* lassen sich Verhaltensreporte optimal auf ihre Analyse mit Techniken des maschinellen Lernens vorbereiten. Die mit der Übersetzung einhergehende Datenreduktion und -konzentration beeinflussen dabei sowohl die Genauigkeit als auch die Laufzeit der Clusteranalyse und der Klassifikation. Die Genauigkeit, mit der bekannte Schadprogrammfamilien korrekt klassifiziert werden, konnte unter Verwendung von MIST im Schnitt um 21 Prozent gesteigert werden. Für unbekannte Schadprogrammfamilien stieg die Genauigkeit sogar um fast 79 Prozent. Bei der Clusteranalyse liefert der auf MIST-codierten Daten arbeitende Algorithmus rund 7 Prozent bessere Ergebnisse als das zum Zeitpunkt der wissenschaftlichen Veröffentlichung genaueste, wissenschaftliche Analysesystem.

Ausblick auf das Kapitel. In Abschnitt 3.1 werden verwandte Arbeiten vorgestellt. Das *Malware Instruction Set* wird in Abschnitt 3.2 eingeführt und anhand mehrerer Beispiele vorgestellt. Zusätzlich erfolgt eine empirische Bewertung des Ansatzes. Ein konkreter Einsatz von MIST bei der Analyse von Schadprogrammen wird in Abschnitt 3.3 präsentiert. Die aus diesen Ergebnissen resultierenden Verhaltensfilter werden in Abschnitt 3.4 vorgestellt und anhand eines Beispieldatensatzes evaluiert. Das Kapitel schließt mit einer Zusammenfassung und einem Ausblick auf mögliche Weiterentwicklungen in Abschnitt 3.5.

3.1 Verwandte Arbeiten

Schadprogramme stellen seit Jahren das größte Risiko im Internet dar. Bei nahezu jedem Angriff auf Systeme sind Schadprogramme beteiligt und mit Würmern und Bots existieren Programme, die vollkommen autonom Systeme im Internet angreifen und kompromittieren. Der Schadprogrammanalyse und -bekämpfung wird daher viel Aufmerksamkeit seitens der Wissenschaft gewidmet, wobei viele verschiedene Ansätze und Methoden entwickelt wurden, die auch Überschneidungen mit dem in diesem Kapitel vorgestellten Ansatz aufweisen.

Statische Analyse von Schadprogrammen. Einer der ersten Ansätze zum Analysieren und Erkennen von Schadprogrammen wurde von Lo et al. [1995] vorgestellt. Die Schadprogramme wurden hierbei manuell analysiert, um sogenannte *telltale signs* zu extrahieren, die als Indikator für bösartiges Verhalten galten und zum Erkennen von anderen Schadprogrammen verwendet werden konnten. Dieser Ansatz wurde von Christodorescu and Jha [2003] weiterentwickelt. Sie schlugen eine gegen bekannte Verschleierungstechniken resistente Architektur vor, die bestimmte bösartige Muster suchte. Schließlich stellten Christodorescu et al. [2005] eine semantische Analyse von Schadprogrammen vor, die von Preda et al. [2007, 2008] erweitert wurde.

Trotz allem bleibt die statische Analyse von Schadprogrammen sehr anfällig für Verschleierungstechniken, die von vielen Angreifern eingesetzt werden, um die Analyse zu erschweren [Ferrie, 2008, 2009, Linn and Debray, 2003, Popov et al., 2007, Szor, 2005]. Auch wenn einfache Verschleierungstechniken bis zu einem gewissen Grad kompensiert werden können, bleibt das allgemeine Problem der Analyse von verschleiertem Code NP-vollständig [Moser et al., 2007a]. Aus diesem Grund wird in dieser Arbeit mit der dynamischen Analyse gearbeitet. Unter der Voraussetzung, dass das Programm seinen Zweck erfüllen soll, lässt sich sein Verhalten weit weniger einfach kaschieren als sein Code.

Dynamisches Entpacken von Schadprogrammen. Um die statische Analyse von Schadprogrammen zu erschweren, setzten aktuelle Schadprogramme mehrheitlich Packer oder Verschlüsselungsalgorithmen ein. Um den eigentlichen Code aus dieser gepackten Variante zu extrahieren und eine anschließende statische Analyse wieder zu ermöglichen, wurden in den letzten Jahren generische *Unpacker* entwickelt [Böhne, 2010, Dinaburg et al., 2008, Martignoni et al., 2007, Royal et al., 2006, Sharif

et al., 2009]. Vereinfacht dargestellt, arbeiten diese Entpacker nach folgendem Prinzip: Das Schadprogramm wird ausgeführt und während der Laufzeit überwacht. Der Entpacker muss dabei erkennen, wann die Entpackroutine beendet ist und das Schadprogramm vollständig und unverschlüsselt im Hauptspeicher liegt. Anschließend werden die entsprechenden Seiten des Speichers ausgelesen und das Schadprogramm aus diesen Daten neu zusammengesetzt. Im Idealfall kann das Programm anschließend statisch analysiert werden. Auf diesem Prinzip beruhende Ansätze können gewissermaßen als Kombination der dynamischen und statischen Analyse betrachtet werden und einige Probleme der rein statischen Analyse beheben. Trotzdem bleiben auch hier mögliche Verschleierungstechniken ein großes Problem. Das Schadprogramm muss theoretisch zu keinem Zeitpunkt vollständig entpackt im Speicher liegen. Auch können gänzlich andere Verschleierungstechniken angewendet werden, die es unmöglich machen, das Programm in seiner „Reinform“ zu extrahieren. Der in dieser Arbeit verfolgte Ansatz der Schadprogrammanalyse basiert deshalb auf der rein dynamischen Analyse, die unabhängig von Verschleierungstechiken und genutzten Packern ist.

Dynamische Analyse von Schadprogrammen. Die mit der statischen Analyse einhergehenden Probleme machen eine rein dynamische Analyse von Schadprogrammen sehr attraktiv. In Analysesystemen wie CWSANDBOX [Willems et al., 2007], Anubis [Bayer et al., 2006a,b], BitBlaze [Song et al., 2008], Norman Sandbox [Norman, 2003] oder ThreatExpert [ThreadExpert, 2010] werden Schadprogramme in einer kontrollierten Umgebung ausgeführt. Die von dem Schadprogramm ausgeführten Operationen und damit die am System vorgenommenen Veränderungen werden dabei aufgezeichnet und am Ende der Analyse in einem detaillierten Verhaltensreport zusammengefasst. In dieser Arbeit wurde das Analysewerkzeug CWSANDBOX eingesetzt, um das Verhalten der Schadprogramme zu protokollieren. Prinzipiell ließe sich aber auch auf eines der anderen Analysesysteme zurückgreifen.

Wie in Abschnitt 2.1.2 dargestellt, ist die dynamische Verhaltensanalyse weit weniger anfällig für Verschleierungstechniken, als die statische Code-Analyse. Trotzdem existieren auch für die Verhaltensanalyse verschiedene Ansätze, um eine detaillierte Analyse erfolgreich zu unterbinden [Sharif et al., 2008]. Der größte Nachteil bei der dynamischen Analyse besteht darin, dass in der Regel nur ein Ausführungspfad aufgezeichnet wird. Das kann zu einem unvollständigen Bild des Verhaltens der Schadprogramme führen. Kommen Teile des Programms, die nur unter bestimmten Bedingungen ausgeführt werden, nicht zur Ausführung, kann ein vollkommen falsches Bild eines Programms entstehen. Um diese Beschränkung der Verhaltensanalyse aufzuheben, haben Moser et al. [2007b] die Technik der *multi-path execution* vorgeschlagen. Der resultierende Verhaltensreport besitzt dann keine einfache sequentielle Struktur mehr, sondern muss als Baum oder Graph dargestellt werden. Da die in dieser Arbeit verwendete Analyseumgebung CWSANDBOX diese Technik nicht unterstützt, wird in der vorliegenden Arbeit auf die Betrachtung mehrerer Ausführungspfade verzichtet. Die Ergebnisse lassen sich aber unabhängig davon auch auf ein System, das die Technik der *multi-path execution* unterstützt, übertragen.

Schadprogrammanalyse mit Methoden des maschinellen Lernens. Der Einsatz von Methoden des maschinellen Lernens zur Klassifikation von Schadpro-

grammen wurde zuerst von Schultz et al. [2001] und Kolter and Maloof [2006] untersucht. Beide Arbeiten verwenden einfache, aus dem compilierten Schadprogramm gewonnene und als Zeichenketten codierte Merkmale, um Lernalgorithmen zu trainieren und bösartiges von normalem Verhalten zu unterscheiden. Der Ansatz wurde von Perdisci et al. [2008] nochmals erweitert. Auch der Vorschlag von Stolfo et al. [2007], der den Inhalt von Dateien nach darin eingebetteten Schadprogrammen durchsucht, arbeitet mit einer ähnlichen Technik. Die Klassifikation des Verhaltens von Schadprogrammen wird erst von Lee and Mody [2006] und Rieck et al. [2008] eingeführt. Die hier vorgestellte Arbeit baut auf diesen Arbeiten auf und entwickelt das Konzept konsequent weiter.

Während die Klassifikation genutzt werden kann, um bekannte Schadprogramme einer bekannten Klasse zuzuweisen, können mit der Clusteranalyse neue Verhaltensmuster erkannt werden, was den Aufwand der manuellen Analyse stark reduziert. Das erste System, das die Clusteranalyse auf aufgezeichnetem Verhalten von Schadprogrammen einsetzt, wurde von Bailey et al. [2007] eingeführt und später von Bayer et al. [2009] optimiert. Das System von Bayer et al. [2009] besitzt eine exzellente Laufzeit auf realen Daten, muss aber genau wie das System von Bailey et al. [2007] immer sämtliche zu untersuchenden Schadprogramme in einem Lauf analysieren. Das in Abschnitt 3.3 vorgestellte System erweitert die bisherigen Arbeiten um eine inkrementelle Analyse. Durch die Kombination der Clusteranalyse mit der Klassifikation wird hier erstmals eine iterative Analyse von Verhaltensreports ermöglicht, die den praktischen Einsatz des Systems für beliebig große Datensätze ermöglicht.

Signaturgenerierung zum Filtern von Schadprogrammen. Bei der verhaltensbasierten Clusteranalyse von Schadprogrammen werden immer auch die charakteristischen Verhaltensmuster für die einzelnen Cluster berechnet. Diese Muster bündelt zum Beispiel Systemaufrufe, die von allen Schadprogrammen des Clusters ausgeführt wurden, oder auch Sequenzen solcher Systemaufrufe. Die Idee, diese Informationen als Signatur für Schadprogrammfamilien zu verwenden, wird auch an der Universität Dortmund seit circa 2 Jahren verfolgt. Die dynamische Analyse der Schadprogramme erfolgt hier ebenfalls mit CWSANDBOX. Nach der Clusteranalyse der Verhaltensreporte extrahieren Apel et al. [2010] die charakteristischen Verhaltenssequenzen und gleichen diese mit den Verhaltenssequenzen gutartiger Programme ab. Dadurch soll die *false-positives* Rate beim Filtern minimiert werden. Erste Informationen zu der Vorverarbeitung der Verhaltensreporte wurden zwar von Apel et al. [2009] erwähnt, aber Details zu den generierten Signaturen und den eingesetzten Filtern wurden noch nicht in wissenschaftlichen Beiträgen veröffentlicht.

Das von Kolbitsch et al. [2009] vorgestellte System verwendet sogenannte Verhaltsgraphen, um das sicherheitskritische Verhalten von Schadprogrammen abzubilden. Dazu werden die Schadprogramme in ANUBIS ausgeführt und die Verhaltensreporte werden in gerichtete azyklische Graphen umgewandelt, die anschließend zum Filtern eingesetzt werden können. Jeder Graph stellt dabei das Verhalten eines einzelnen Schadprogramms dar, das heißt, die Graphen sind keine Signaturen für ganze Familien. Diese „Einschränkung“ spiegelt sich auch in den Erkennungsquoten der Filter wider, die für einzelne Familien bei nur 10 Prozent liegen.

3.2 Das *Malware Instruction Set* - MIST

Die meisten Analyseumgebungen zur verhaltensbasierten (Schad-) Programmanalyse, wie beispielsweise Anubis [Bayer et al., 2006b] und CWSANDBOX [Willems et al., 2007], verwenden textuelle- oder *Extensible Markup Language* (XML)-Codierungen, um das beobachtete Verhalten zu protokollieren. Diese Codierungen erlauben es zwar für menschliche Analysten leicht verständliche Reporte zu generieren, stellen aber für die maschinelle Weiterverarbeitung der Reporte ein Problem dar. Die strukturierten und häufig mehrfach geschachtelten Reporte enthalten in vielen Fällen zu viele Informationen, die das beobachtete Verhalten sehr genau spezifizieren. Dadurch wird das Auffinden von generischen Verhaltensmustern nahezu unmöglich, was speziell bei der Anwendung von Methoden des maschinellen Lernens für schlechte Ergebnisse verantwortlich ist. Die textuellen Verhaltensreporte hingegen sind in der Regel zu ungenau. Hier wird das beobachtete Verhalten sehr stark aggregiert und vereinfacht, um es in einer von Menschen lesbaren Form zu präsentieren. Dadurch sind die sich dahinter befindenden Verhaltensmuster jedoch nicht mehr erkennbar. Neben diesen Problemen vergrößert die Komplexität beider Codierungen die Reporte zusätzlich. Die Größe der Reporte beeinflusst – gerade bei der Analyse großer Datensätze – die Laufzeit und den Speicherbedarf der Analysen erheblich.

Um die beschriebenen Probleme zu lösen und die Weiterverarbeitung der Reporte zu optimieren, wurde eine spezielle Verhaltensrepräsentation entwickelt [Trinius, 2008, Trinius et al., 2009b, 2010]. Diese wird in Anlehnung an die Instruktionssätze in der Prozessorarchitektur mit *Malware Instruction Set* (MIST) bezeichnet. Im Gegensatz zu den existierenden textuellen und XML-codierten Reporten, wird das beobachtete Verhalten von (Schad-) Programmen in der MIST-Codierung als Sequenz von MIST-Instruktionen beschrieben. Die einzelnen Ausführungen der Prozesse und Threads werden dabei in einem einzelnen, sequenziell geordneten Report zusammengefasst. Jede MIST- Instruktion codiert darin genau einen aufgezeichneten Systemaufruf und dessen Argumente. Zur Codierung der Systemaufrufe werden kurze numerische *Bezeichner* verwendet. So wird der Systemaufruf `move_file` beispielsweise durch `03 05` dargestellt. Um verschiedene Detailstufen bei der Verhaltensrepräsentation zu unterstützen, werden die Argumente der Systemaufrufe „intelligent“ sortiert und auf unterschiedliche Ebenen verteilt. Diese Ebenen werden im Folgenden als MIST-Level – oder kurz *Level* – bezeichnet. Des Weiteren werden Argumente mit variabler Länge, beispielsweise Datei- oder Mutexname, durch einen Hashwert repräsentiert. Diese Hashwerte dienen auch als Index für den entsprechenden Eintrag in der Zuordnungstabelle.

Abbildung 3.1 zeigt die Grundstruktur einer MIST-Instruktion. Das erste Level der MIST-Instruktionen korrespondiert mit der Kategorie und dem Namen des aufgezeichneten Systemaufrufs. Beispielsweise codiert `03 05` die Kategorie `filesystem` (`03`) und den Systemaufruf `move_file` (`05`). Die sich anschließenden Level der MIST-Instruktion enthalten unterschiedliche Blöcke von Argumenten, wobei die Argumente von links nach rechts immer spezifischere Informationen über den Systemaufruf enthalten. Die Grundidee für diese Neuordnung der Argumente besteht darin, sehr variable Bestandteile, wie zum Beispiel temporäre Dateinamen, ans Ende der MIST-Instruktion zu schieben, während *stabile* und diskriminative Teile, wie verwen-

dete Verzeichnisse und Mutexnamen, im vorderen Teil der MIST-Instruktion einsortiert werden. Die Feinheit der Verhaltensreporte lässt sich somit einfach justieren, indem die MIST-Instruktion nur bis zu einem bestimmten MIST-Level gelesen und bei der Analyse des Reports berücksichtigt werden. Dadurch ist es möglich, Schadprogramme mit ähnlichem Verhalten selbst dann einander zuzuordnen, wenn sich einzelne Teile der Instruktion beispielsweise aufgrund der Verwendung zufälliger Dateinamen unterscheiden.

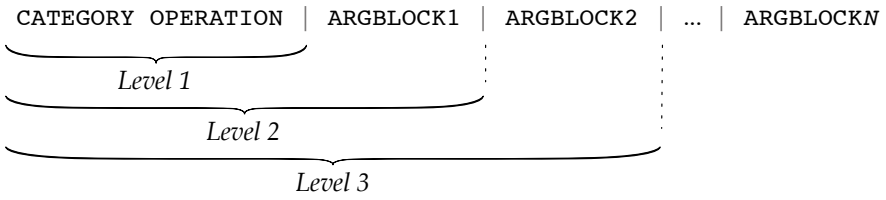


Abbildung 3.1: Schematischer Aufbau einer MIST-Instruktion. Das Feld `CATEGORY` codiert die Kategorie des Systemaufrufs und das Feld `OPERATION` die eigentliche Operation. Die Argumente sind in einen oder mehrere `ARGBLOCK`s unterteilt.

Nachdem das Grundkonzept von MIST eingeführt ist, werden im Folgenden die Repräsentationen verschiedener Systemaufrufe beschrieben. Da MIST insgesamt 120 einzelne Systemaufrufe mit den entsprechenden Attributen unterstützt, können an dieser Stelle nicht alle Systemaufrufe beschrieben werden. Stattdessen werden einige wenige repräsentative Beispiele vorgestellt, die das Konzept der MIST-Instruktionen weiter erläutern und vertiefen sollen. Tabelle 3.1 gibt zusätzlich einen Überblick über die Verteilung der einzelnen Systemaufrufe auf die verschiedenen Kategorien von Systemaufrufen.

3.2.1 Verhaltensrepräsentation

Jeder MIST-Report besteht aus einer Vielzahl von MIST-Instruktionen, die einzelne, während der Analyse aufgezeichnete Systemaufrufe codieren. Ein MIST-Report könnte prinzipiell auch sofort während der Verhaltensanalyse erstellt werden. Da aktuell aber noch keine Analyseumgebung über ein entsprechendes Plugin verfügt, werden im Folgenden Übersetzungen von XML-codierten Reporten, die mit der Analyseumgebung CWSANDBOX [Willems et al., 2007] erstellt wurden, in MIST betrachtet. Die Konvertierung behält dabei die Ordnung der aufgezeichneten Systemaufrufe bei, das heißt, alle enthaltenen MIST-Instruktionen kommen in derselben Reihenfolge vor, in der sie von der CWSANDBOX aufgezeichnet und protokolliert wurden.

Wie bereits erwähnt, bestehen MIST-Instruktionen aus unterschiedlichen Feldern: ein `CATEGORY` Feld, ein `OPERATION` Feld und ein oder mehrere `ARGBLOCK` Felder. Das Feld `CATEGORY` codiert die globale Klasse der MIST-Instruktion. Wie in Tabelle 3.1 dargestellt, wird dabei zwischen 20 Kategorien unterschieden. Jede Kategorie kapselt wiederum eine oder mehrere verwandte Operationen. So enthält zum Beispiel die `winsock_op` Kategorie die Operationen `create_ socket`, `connect_socket` und `send_socket` sowie 12 weitere Operationen. Diese Ope-

rationen stellen zusammen alle sicherheitsrelevanten Systemaufrufe dar, die notwendig sind, um eine *Winsock*-basierte Netzwerkkommunikation durchzuführen.

Codierung	Kategorie	Anzahl der Systemaufrufe
01	Windows COM	4
02	DLL Handling	3
03	Filesystem	14
04	ICMP	1
05	Inifile	5
06	Internet Helper	5
07	Mutex	2
08	Network	6
09	Registry	9
0A	Process	7
0B	Windows Services	11
0C	System	2
0D	Systeminfo	7
0E	Thread	3
0F	User	8
10	Virtual Memory	5
11	Window	5
12	Winsock	13
13	Protected Storage	9
14	Windows Hooks	1

Tabelle 3.1: MIST-Kategorien und deren Codierung sowie die Anzahl der in ihnen enthaltenen Operationen

Die Anzahl und die Art der `ARGBLOCK` Felder für jede MIST-Instruktion hängt von dem jeweiligen Systemaufruf ab. In Abschnitt 3.2.2 werden hierzu einige Beispiele präsentiert.

Um das Konzept der MIST-*Level* umzusetzen, wurde für jede MIST- Instruktion eine optimale Anordnung der Argumente erarbeitet. Anschließend wurden die Blöcke von Argumenten in mehrere Ebenen (Level) unterteilt, wobei die hohen Level diejenigen Argumente mit hoher *Variabilität* und die niedrigen Level die eher konstanten Argumente enthalten. Als sehr variabel werden dabei diejenigen Argumente bezeichnet, die beim wiederholten Ausführen desselben Schadprogramms, unterschiedliche Werte besitzen. So ist es zum Beispiel sehr wahrscheinlich, dass ein Programm, das während der ersten Analyse eine Datei im Windows-Verzeichnis `temp` anlegt, dies auch während der folgenden Analyseläufe macht. Gleichzeitig wird diese Datei aber mit einer ebenfalls hohen Wahrscheinlichkeit jedes Mal einen anderen Namen besitzen, da für temporäre Dateien in der Regel zufällige Dateinamen verwendet werden. Folglich wird in einer MIST-Instruktion das Erstellen der Datei im ersten Level codiert, der Pfad zu der angelegten Datei aber im zweiten Level und der Dateiname selbst im dritten Level.

Dem angedeuteten Vorgehen liegt die Annahme zugrunde, dass beim Vergleich zweier Ausführungen eines Programms auf einem niedrigen Level (Level 1) das iden-

tische Verhalten beobachtet wird, während auf höheren Levels feine Unterschiede, wie beispielsweise Dateinamen, zwischen den einzelnen Läufen erkannt werden können.

Die angedeutete *Aufsplittung* der Dateinamen wird in MIST konsequent umgesetzt: Jeder Dateiname wird in die Komponenten `Dateityp`, `Dateipfad`, `Dateiname` und `Parameter` aufgeteilt. Da Typ und Pfad einer Datei robuster sind als ihr eigentlicher Name und mögliche Parameter, werden der `Dateityp` und der `Dateipfad` in niedrigeren Levels gespeichert als die übrigen Bestandteile.

Neben der Neuordung der Argumente und dem Aufsplitten der Dateinamen werden bei der Transformation auch verschiedene Argumentwerte ersetzt. So wird im ersten Schritt der Dateiname des analysierten Programms, zum Beispiel `D:\sandbox\malware.exe`, durch `C:\analysetarget.exe` ersetzt. Dieses Ersetzen erfolgt in allen Argumenten, die das Schadprogramm über seinen Dateinamen referenzieren. Neben der ersten Instruktion – diese lädt das zu analysierende Programm – können dies beispielsweise das Dateisystem betreffende Instruktionen sein, oder auch solche, die das Programm als Dienst in die Windows-*Registry* eintragen.

Eine weitere Neuerung von MIST besteht darin, die von dem Betriebssystem verwendeten Prozess- und Thread-Identifikationsnummern (IDs) durch „genauere“ Informationen zu ersetzen. Sowohl die Prozess- als auch die Thread-IDs werden von dem Betriebssystem „zufällig“ vergeben. Die konkreten Werte besitzen daher keine nützlichen Informationen für die Verhaltensanalyse, werden aber für die Zuordnung von zahlreichen Instruktionen, zum Beispiel `create_process` oder `kill_process`, benötigt. In der MIST-Darstellung werden daher alle Prozess-IDs durch den Dateinamen des Prozesses ersetzt, der anschließend, wie alle übrigen Dateinamen, ebenfalls aufgesplittet wird. Um den Dateinamen zu ermitteln, wird entweder auf die Liste laufender Prozesse zurückgegriffen, oder das den Prozess erstellende Kommando wird ermittelt und geparst. Die in diesem Kommando eventuell enthaltenen Aufrufparameter werden dabei im Argument `Parameter` abgespeichert. Für die Threads existiert leider kein aussagekräftiger Bezeichner. Trotzdem werden die vom Betriebssystem vergebenen Identifikationsnummern durch neue ersetzt. Die Threads eines jeden Prozesses werden entsprechend ihrem Auftreten aufsteigend durchnummeriert, das heißt, der erste Thread innerhalb eines Prozesses erhält immer auch die Thread-ID 1. Auch wenn diese Nummerierung den Thread nicht genauer beschreibt als die ursprüngliche, kann damit zumindest eine erste Ordnung in die Threads jedes Prozesses gebracht werden, wodurch sich die Unterschiede zwischen den Reporten eines Schadprogramms, oder mehrerer Schadprogramme einer Familie, weiter reduzieren.

3.2.2 Beispiele

Für jeden von MIST unterstützen Systemaufruf existiert eine feste MIST-Codierung. Zusätzlich ist die Anordnung der Argumente für alle MIST-Instruktionen fest definiert. Zum Zeitpunkt der Übersetzung eines protokollierten Systemaufrufs ist damit bekannt, welche Argumente in welcher Codierung an welcher Stellen in der letztlich erstellten MIST-Instruktion vorkommen. Sollte eines der erwarteten Argumente in dem aufgezeichneten Systemaufruf nicht belegt sein, wird es in der MIST-Instruktion mit dem Wert 0 (Null) belegt.

Der `load_dll` Systemaufruf

Das Subsystem in Windows Betriebssystemen ist über Bibliotheken – *dynamic-link libraries* (DLLs) – implementiert. Diese Bibliotheken lassen sich in Programme einbinden und stellen die Schnittstellen zum eigentlichen Betriebssystem bereit. Der Systemaufruf `load_dll` wird folglich von jedem Programm sowohl während der Prozessinitialisierung als auch zur Laufzeit mehrfach aufgerufen.

```
<load_dll filename="C:\WINDOWS\system32\kernel32.dll" successful="1"
 address="#7C800000" end_address="#7C908000" size="1081344"
 filename_hash="c88d57cc99f75cd928b47b6e444231f26670138f"/>
```

(a) CWSANDBOX-Repräsentation

```
                    "c:\windows\system32\"                  parameter              successful
   02 02   | 00006b2c 0c7d3f9c | 00108000 0c94b872 | 00000000 7C800000 7C908000 10
load_dll      "dll"                 size    "kernel32"          address  end_address
```

(b) MIST-Repräsentation

Abbildung 3.2: Repräsentation des Systemaufrufs `load_dll`. Im CWSANDBOX-Format wird der Systemaufruf als XML-Element dargestellt, während er in der MIST-Darstellung als strukturierte Zeichenkette codiert wird.

Abbildung 3.2 zeigt für einen mit der CWSANDBOX protokollierten `load_dll` Systemaufruf das XML-Element in CWSANDBOX-Repräsentation und die dazugehörige MIST-Instruktion. Bis auf das Attribut `filename_hash` werden alle Attribute des XML-Elements in die MIST-Repräsentation übernommen. Attribute die für die spätere Analyse keinerlei Nutzen beinhalten, wie hier das Attribut `filename_hash`, finden bei der Übersetzung in MIST keine Berücksichtigung. Die Werte der übrigen Attribute werden wie folgt in ihre MIST-Repräsentation überführt: Attribute mit numerischem Wert (`address`, `end_address` und `size`) werden in hexadezimaler Darstellung mit fester Länge gespeichert. Attribute, die nur vorab bekannte Werte annehmen können – wie etwa das Attribut `successful` – werden über eine feste Zuordnungstabelle übersetzt.[1] Die Werte von nicht numerischen, frei wählbaren Attributen, wie das Attribut `filename`, werden in mehreren Schritten in die MIST-Darstellung überführt: Im ersten Schritt wird der zu verarbeitende Wert in Kleinbuchstaben konvertiert. Dadurch soll verhindert werden, dass unterschiedliche Schreibweisen von Dateinamen und anderen Attributen den Vergleich von MIST-Reporten negativ beeinflussen. Handelt es sich bei dem Attribut zudem um einen Dateinamen, wird dieser zur weiteren Verarbeitung in seine Bestandteile (`Dateipfad`, `Dateiname`,

[1] Obwohl es sich bei `successful` um ein binäres Attribut handelt, wird dieses über einen zweistelligen Bitvektor codiert. Dies ist notwendig, um neben `True` (10) und `False` (01) auch den Wert „undefiniert“ (00) darstellen zu können.

Dateityp und Parameter) zerlegt. Die Teile werden anschließend einzeln weiterverarbeitet. Jeder Wert wird mit Hilfe einer schnellen Hash-Funktion – beispielsweise dem Standard ELF – in einen numerischen Wert übersetzt, der hexadezimal codiert in die MIST-Instruktion mit aufgenommen wird. Falls das Protokollieren der Zuordnung von Hash- und verarbeitetem Wert für nicht numerische Argumente aktiviert ist, wird diese Zuordnung zusätzlich in der entsprechenden Zuordnungstabelle abgespeichert. Die MIST-Codierung des Wertes dient dabei als Index für den jeweiligen Tabelleneintrag.

Innerhalb der load_dll MIST-Instruktion werden die Argumente wie folgt geordnet: Die beiden ersten Argumente sind der Dateityp und der Dateipfad der Bibliothek. Diese Informationen sind relativ konstant und daher im zweiten MIST-Level enthalten. Wie bereits erwähnt, wird in MIST versucht, die Argumente entsprechend ihrer *Variabilität* anzuordnen. Im dritten Level werden die Größe und der Dateiname der Bibliothek abgespeichert. Beide Werte können bei der Analyse von zwei unterschiedlichen Varianten eines Programms abweichen und sollten daher auf einem höheren, detaillierteren MIST-Level gespeichert werden. Die variabelsten Argumente der load_dll Instruktion (Parameter, address, end_address und successful) schließen die MIST-Instruktion auf Level 4 ab.

Dieses Beispiel zeigt anschaulich, dass MIST-Instruktionen eine bedeutend bessere Datenbasis für Techniken des Data-Mining und des maschinellen Lernens darstellen als die traditionelle XML-Repräsentation. Die kompakte Darstellung macht einen schnellen Vergleich aller load_dll Instruktionen möglich und das intelligente Anordnen der Argumente erlaubt die Einführung von MIST-Levels, die schlussendlich die Qualität der Analyse verbessern.

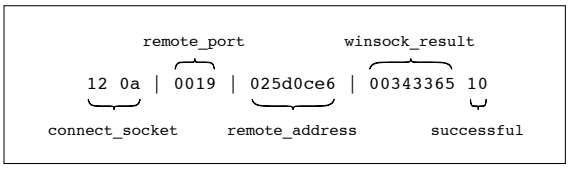
Der connect_socket Systemaufruf

Wenn Schadprogramme eine Netzwerkverbindung über die *Winsock*-Bibliothek aufbauen wollen, müssen sie den Systemaufruf connect_socket ausführen. Abbildung 3.3 zeigt die CWSANDBOX- und MIST-Repräsentation eines aufgezeichneten connect_socket Systemaufrufs. Das von der Analyseumgebung protokolliert XML-Element besitzt fünf Attribute: Die Nummer des verwendeten Socket (socket), die IP-Adresse (remote_addr) und den Port (remote_port) des kontaktierten Servers, das Winsock Ergebnis (winsock_result) und einen booleschen Wert (successful) der anzeigt, ob die Verbindung erfolgreich aufgebaut werden konnte.

Mit Ausnahme der *Socketnummer* – dabei handelt es sich um einen dynamischen Wert, der von dem Betriebssystem bestimmt wird und dem damit keine weitere Bedeutung zukommt – werden alle Attribute des XML-Elements in die MIST-Repräsentation übernommen. Die Ordnung der Argumente kann der Abbildung 3.3(b) entnommen werden. Da die IP-Adresse des zu kontaktierenden Servers und das *Winsock*-Ergebnis sehr stark variieren können, werden diese Informationen in MIST-Level 4 verschoben. Einzig der Port, auf dem die Kommunikation stattfinden soll, verbleibt für den connect_socket Systemaufruf in Level 2.

```
<connect_socket socket="1500" remote_addr="192.168.1.163"
 remote_port="25" successful="1" winsock_result="10035"/>
```

(a) CWSANDBOX Repräsentation



(b) MIST Repräsentation

Abbildung 3.3: Repräsentation des Systemaufrufs `connect_socket`. Im CWSANDBOX-Format wird der Systemaufruf als XML-Element dargestellt, während er in der MIST-Darstellung als strukturierte Zeichenkette codiert wird.

Der `move_file` Systemaufruf

Das dritte Beispiel stellt eine sehr komplexe Transformation vor, da hierbei mehrere Dateinamen in Argumenten eines Systemaufruf enthalten sind. Dateinamen in Argumenten der Systemaufrufe werden bei der Übersetzung in MIST in ihre Bestandteile (Pfad, Dateiname, Dateityp und Parameter) zerlegt. Lediglich der Dateityp und der Pfad werden anschließend im zweiten MIST-Level berücksichtigt. Der eigentliche Dateiname und mögliche Parameter hingegen werden in höhere Level verschoben. Bei dem Systemaufruf `move_file` existieren nun mit der Quelle (`srcfile`) und dem Ziel (`dstfile`) der Operation zwei Argumente, die Dateinamen spezifizieren. Folglich müssen hier zwei Dateinamen gesplittet und neu geordnet werden. Abbildung 3.4 zeigt einen aufgezeichneten `move_file` Systemaufruf in CWSANDBOX- und MIST-Notation.

Auch bei diesem Systemaufruf werden mit `srcfile_hash` und `dstfile_hash` die Hashwerte der betroffenen Dateien bei der Konvertierung in MIST nicht beachtet. Die übrigen Argumente fließen, mit Ausnahme der Dateinamen und Parameter, alle in das erste MIST-Level ein (siehe Abbildung 3.4 (b)). Im Gegensatz zu den Dateinamen werden die Werte der Argumente `filetype`, `desiredaccess` und `flags` während der Übersetzung in MIST nicht gehasht, sondern über eine *Zuordnungstabelle* direkt in MIST übersetzt. Diese Argumente können nur eine begrenzte Anzahl bekannter Werte annehmen, weshalb auf eine feste Zuordnung zwischen Argumentwerten und MIST-Werten zurückgegriffen wird. Tabelle 3.2 zeigt einen Auszug aus der Zuordnungstabelle für das `flags` Argument. Die einzelnen Werte werden dabei als *Bitvektoren* definiert. Insgesamt sind hier 26 vordefinierte Werte möglich, die frei kombiniert werden können.

```
<move_file filetype="file" srcfile="C:\DOKU...\Temp\&#x7E;5.tmp.exe"
 srcfile_hash="hash_error"
 dstfile="C:\WINDOWS\system32\nxzhpxgdmwuod.exe"
 dstfile_hash="hash_error" desiredaccess="FILE_ANY_ACCESS"
 flags="MOVEFILE_REPLACE_EXISTING"/>
```

(a) CWSANDBOX Repräsentation

```
                flags      "exe"            "exe"          "&#x7E;5.tmp"     "nxzhpxgdmwuod"
  03 04  | 00 400000 01 00006ce5 0d7ac98c 00006ce5 0c7d3f9c | 08d0f2f0 00000000 0bb41dc4 00000000
move_file    :      filetype   "c:\dok...\temp\" "c:\windows\system32\"      parameter          parameter
   creationdistribution
```

(b) MIST Repräsentation

Abbildung 3.4: Repräsentation des Systemaufrufs `move_file`. Im CWSANDBOX Format wird der Systemaufruf als XML-Element dargestellt, während er in der MIST-Darstellung als strukturierte Zeichenkette codiert wird.

Für Argumente, die gleichzeitig mehrere Werte annehmen können – dies trifft sowohl auf das Argument `flags` als auch auf das Argument `desiredaccess` zu – werden die Bitvektoren der einzelnen Werte aufaddiert und anschließend als binär codierte numerische Zahl interpretiert. In die MIST-Instruktion wird dieser Wert in hexadezimaler Codierung mit fester Länge aufgenommen. Dieser Ansatz ist zum einen robust gegen Permutationen in der Anordnung der einzelnen Werte und erlaubt zum anderen eine Rückübersetzung des MIST-Werts in die Ausgangsnotation.

Wert	**MIST Bitvektoren**
FILE_ATTRIBUTE_ARCHIVE	00000000000000000000000001
FILE_ATTRIBUTE_COMPRESSED	00000000000000000000000010
FILE_ATTRIBUTE_HIDDEN	00000000000000000000000100
FILE_ATTRIBUTE_NORMAL	00000000000000000000001000
FILE_ATTRIBUTE_OFFLINE	00000000000000000000010000
...	...
MOVEFILE_WRITE_THROUGH	00100000000000000000000000
MOVEFILE_CREATE_HARDLINK	01000000000000000000000000
MOVEFILE_FAIL_IF_NOT_TRACKABLE	10000000000000000000000000

Tabelle 3.2: Zuordnung zwischen der CWSANDBOX- und der MIST-Repräsentation für verschiedene Werte des Arguments `flags`.

Der MIST-Report

Listing 3.1 zeigt einen Auszug aus einem auf das zweite Level beschnittenen MIST-Report eines Schadprogramms der Familie ADULTBROWSER[2]. Der vollständige Report umfasst 723 MIST-Instruktionen, die sich auf zwei Prozesse verteilen. Der initiale Prozess, also der von der Analyseumgebung gestartete, besitzt fünf Threads während der zweite, von dem Schadprogramm selbst erzeugte Prozess, lediglich aus einem Thread besteht.

```
1  # process 00000000 0000066a 022c82f4 00000000 thread 0001 #
2  02 01 | 00000000 0000066a 0002e000
3  02 02 | 00006b2c 047c8042 000b9000
4  02 02 | 00006b2c 047c8042 00108000
5  02 02 | 00006b2c 047c8042 000aa000
6  ...
7  03 0a | 000004 004000 01 00000000 0da3b4d3
8  03 03 | 03 000004 000002 004000 01 00006a22 0bb7ba3e
9  03 03 | 03 000004 000006 004000 02 00623700 00000000
10 0d 03 |
11 0e 01 |
12 02 02 | 00006b2c 047c8042 0013d000
13 0e 01 |
14 01 01 | 00000000 00000000 051cb26d
15 09 02 | 02fe81cd
16 09 05 | 02fe81cd 0376c925
17 0d 01 |
18 ...
19 09 02 | 0a1f802c
20 09 05 | 0d043cdd 06502568
21
22 # process 00000000 0000066a 022c82f4 00000000 thread 0002 #
23 0c 01 | 0000ea60
24
25 # process 00000000 0000066a 022c82f4 00000000 thread 0003 #
26 02 02 | 00006b2c 047c8042 00374000
27 09 02 | 0a288e6c
28 09 02 | 0d3c6adc
29 09 02 | 0a1f802c
30 09 02 | 0e9a697c
31 09 02 | 062d69ee
32 09 02 | 0ab6c38e
33 03 07 | 000004 004000 01 00006df6 0d004290
34 0e 01 |
35 03 07 | 000004 004000 01 00006df6 0d004290
36 ...
37 # process 00006ce5 047c8042 0ac9f6d4 00000000 thread 0001 #
38 02 01 | 00006ce5 047c8042 00006000
39 02 02 | 00006b2c 047c8042 000b9000
40 02 02 | 00006b2c 047c8042 00108000
41 02 02 | 00006b2c 047c8042 000aa000
42 ...
```

Listing 3.1: Auszüge aus dem Level 2 MIST-Report eines Schadprogramms.

[2]Das analysierte Binary ist dem Referenzdatensatz entnommen und besitzt den folgenden SHA1 Hash: `0be249c7a4b903dacc02cd01fcf7ed7e81c1b76c`

Innerhalb eines MIST-Reports werden die Prozesse und Threads durch eine *Info*-Zeile voneinander getrennt. Der Aufbau der Zeile ist in Abbildung 3.5 dargestellt. Die Zeile ist am Anfang und am Ende von dem Symbol # begrenzt und dadurch für Werkzeuge, die auf dem MIST-Report arbeiten, leicht zu erkennen. Die Codierung des Prozessbezeichners erfolgt analog zu den Dateinamen in MIST-Instruktionen. An erster Stelle wird der `Dateityp` codiert. Auf diesen folgen der `Dateipfad` und der `Dateiname`. Am Ende steht dann die Codierung möglicher `Parameter`. Die von MIST zugewiesene `Thread-ID` wird hexadezimal codiert dargestellt.

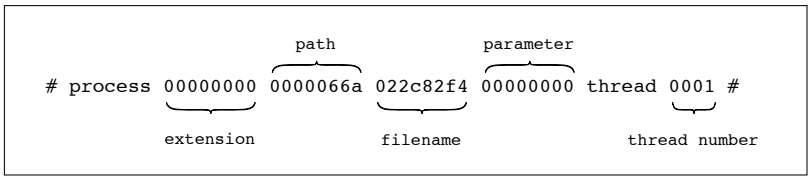


Abbildung 3.5: Aufbau der *Info*-Zeile zur Trennung der einzelnen Prozesse und Threads innerhalb eines MIST-Reports.

Der Report selbst soll hier nur eine Vorstellung von der MIST-Notation geben. Da bei der Entwicklung von MIST das Hauptaugenmerk auf der weiteren Verarbeitung der Reporte mit Hilfe von automatisierten Werkzeugen liegt, tritt die menschliche Lesbarkeit klar in den Hintergrund. Ohne das Wissen um die Codierung der einzelnen Instruktionen und die Anordnung aller Argumente, lassen sich für menschliche Analysten nur wenige bis gar keine Informationen aus einem MIST-Report ziehen.

3.2.3 Empirische Bewertung von MIST

Um den Nutzen von MIST im Zusammenhang mir der dynamischen Verhaltensanalyse von Schadprogrammen messen zu können, wurde MIST mit zwei existierenden Notationen zur Verhaltensrepräsentation verglichen. Als Datenbasis dienten 500 Schadprogramme, die aus dem Datenbestand des MWANALYSIS Dienstes[3] entnommen wurden. MWANALYSIS verfügt über circa eine Million Schadprogramme, die über einen Zeitraum von drei Jahren aus verschiedensten Quellen gesammelt wurden. Neben *Honeypots* und *Spamtraps* trugen auch Hersteller von Antivirus Produkten und verschiedene Forscherteams zum Aufbau dieser Datenbasis bei. Aus diesen Daten wurde eine kleine Teilmenge entnommen und mit Hilfe von sechs unabhängigen Virenscannern klassifiziert. Der Aufbau des so entstandenen Referenzdatensatzes wird in Abschnitt 3.3.2 eingehend beschrieben. Dem Referenzdatensatz wurden 500 Schadprogramme aus 5 verschiedenen Familien entnommen, die anschließend in der Analyseumgebung CWSANDBOX ausgeführt wurden.

[3] MWANALYSIS, erreichbar unter *https://mwanlaysis.org*, bietet eine kostenlose dynamische Verhaltensanalyse mit CWSANDBOX an.

Für jeden der so gewonnenen 500 Verhaltensreports wurden die folgenden vier Repräsentationen herangezogen:

1. Der XML-codierte Report, der von CWSANDBOX geliefert wird.
2. Die von Rieck et al. [2008] vorgeschlagene Erweiterung der XML-codierten Reporte.
3. Die MIST-Level 1 Repräsentation für Verhaltensreporte.
4. Die MIST-Level 2 Repräsentation für Verhaltensreporte.

Verhaltens-Repräsentation

Die Eignung von MIST zur Repräsentation von Programmverhalten wurde experimentell überprüft. Dabei fand der von Rieck et al. [2008] eingeführte Ansatz zur Einbettung von textuellen Daten in einen Vektorraum Anwendung. Bei diesem Ansatz wird jeder Report durch einen Vektor repräsentiert. Dadurch lässt sich die Frage der Ähnlichkeit zwischen einzelnen Reporten auf geometrische Distanzmaße reduzieren.

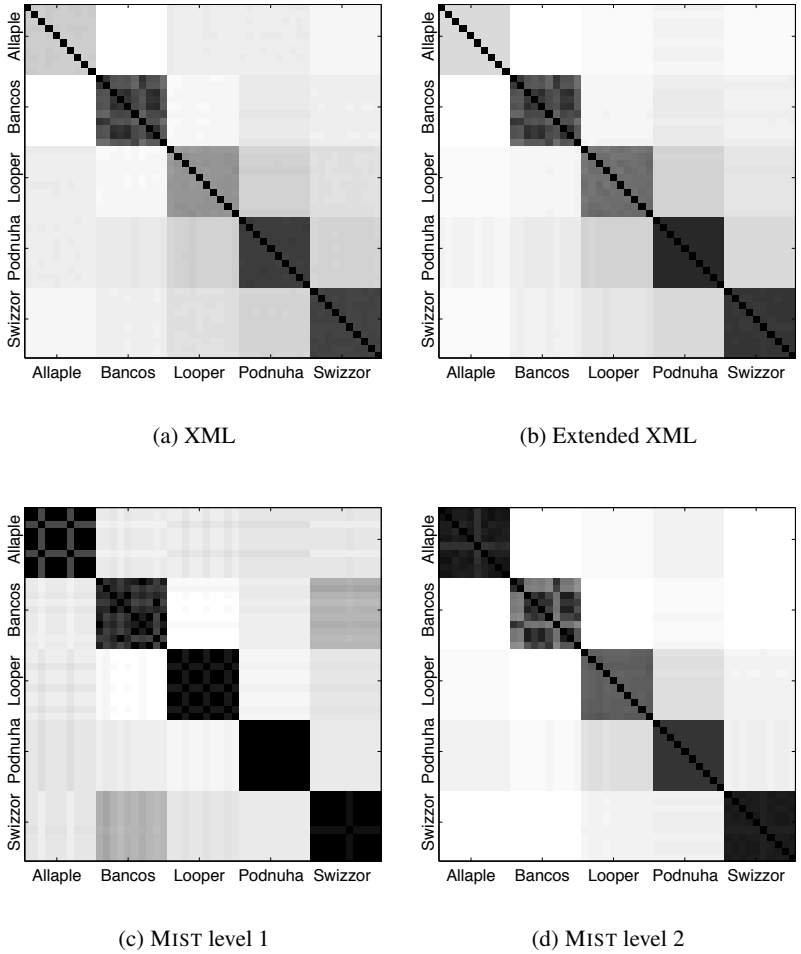
Abbildung 3.6 zeigt die Ergebnisse der empirischen Untersuchung. Innerhalb der Distanzmatrizen deuten dunkle Flächen einen kleinen Abstand zwischen den Reporten und helle Flächen hohe Distanzen an. Die Ergebnisse der original CWSANDBOX Repräsentation zeigen, dass mit dieser XML-codierten Darstellung des beobachteten Verhaltens keine klare Abgrenzung zwischen den untersuchten Familien möglich ist. Innerhalb der beiden Schadprogrammfamilien ALLAPLE und LOOPER zeigt die Distanzmatrix sehr helle Bereiche an. Dies bedeutet, dass auch innerhalb der Familie die Abstände zwischen den Reporten in dieser Notation sehr hoch sind und bei einer Clusteranalyse oder einer Klassifikation, basierend auf den XML-Dokumenten, keine brauchbaren Ergebnisse zu erwarten sind. Um alle Reporte von ALLAPLE Schadprogrammen korrekt zu klassifizieren, müsste eine Schranke gewählt werden, die für die anderen Familien zu niedrig ist. Dadurch würden zum Beispiel Reporte der Familien PODNUHA und SWIZZOR ebenfalls als ALLAPLE klassifiziert.

Die Ergebnisse für die erweiterte XML-Notation unterscheiden sich nur marginal von denen für die XML-Notation der CWSANDBOX. Auch hier kann keine sinnvolle Schranke gewählt werden, die alle Familien sauber voneinander trennt.

Die Distanzmatrizen für die MIST-codierten Daten weisen entlang der Diagonalen bedeutend dunkler eingefärbte Bereiche auf. Die Mitglieder der einzelnen Schadprogrammfamilien liegen hier viel näher zusammen. Dies kann an der für die MIST-Level 1 Repräsentation fast schwarzen Einfärbung bei vier der fünf Familien abgelesen werden. Insgesamt zeigt die Distanzmatrix für die MIST-Level 1 Notation innerhalb der Familien die sauberste Zuordung. Allerdings existieren hier auch Überschneidungen zwischen unterschiedlichen Familien, beispielsweise zwischen BANCOS und SWIZZOR. Durch die MIST-Level 2 Repräsentation reduzieren sich diese Überschneidungen zwischen den Familien. Allerdings verringert sich im Gegenzug die Genauigkeit innerhalb der Familien. Dies ist auf den höheren Detailgrad des beobachteten Verhaltens in Level 2 zurückzuführen.

Das beschriebene Experiment zeigt, dass sich Programmverhalten durch MIST sehr gut darstellen lässt. Auch wenn zur Evaluierung nur fünf Schadprogrammfamilien herangezogen wurden, kann geschlussfolgert werden, dass die MIST-Notation bedeutend

besser für die Verhaltensanalyse geeignet ist, als die gängigen XML-Notationen. In der MIST-Codierung liegen die Verhaltensreporte von Schadprogrammen einer Familie, nach ihrer Vektorisierung, näher zusammen und besitzen gleichzeitig einen ausreichend großen Abstand zu Reporten anderer Schadprogrammfamilien.



(a) XML

(b) Extended XML

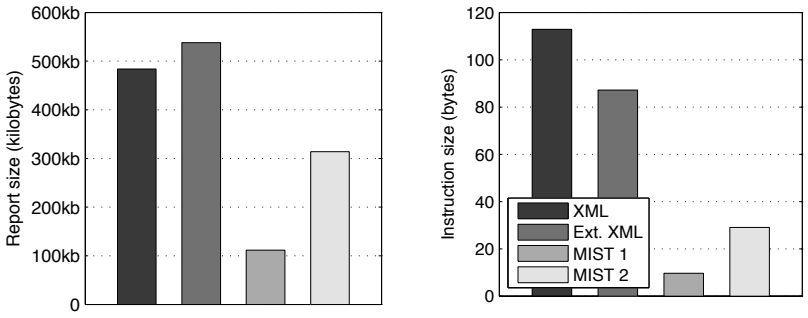
(c) MIST level 1

(d) MIST level 2

Abbildung 3.6: Vergleich ausgewählter Notationen zur Verhaltensrepräsentation. Distanzmatrizen für das Verhalten von fünf Schadprogrammfamilien, wobei jeder Familie durch zehn Reporte repräsentiert wird. Dunkle Flächen zeigen kleine und helle Flächen große Abstände zwischen den Vektoren an.

Datenreduktion

Im zweiten Experiment wurde die Größe der Instruktionen und der Reporte in den jeweiligen Notationen verglichen. Abbildung 3.7 zeigt die Ergebnisse für die vier betrachteten Formate, namentlich die original XML-Notation der CWSANDBOX, die von Rieck et al. [2008] verwendete erweiterte Repräsentation, und die MIST-Level 1 und Level 2 Notationen.



(a) Durchschnittliche Größe der Verhaltensreporte

(b) Durchschnittliche Größe einer Instruktion

Abbildung 3.7: Vergleich der Größe von Verhaltensinstrukionen und -reporte.

Durch die Verwendung der MIST-Repräsentation reduziert sich sowohl die Länge der Reporte, als auch die durchschnittliche Instruktionslänge erheblich. Benötigen Verhaltensreporte in den XML-codierten Formaten im Schnitt noch 450 Kilobyte, reduziert sich der Speicherbedarf für MIST-Level 1 auf 100 Kilobyte und für MIST-Level 2 auf durchschnittlich 300 Kilobyte. Analog verhält es sich mit den einzelnen Instruktionen. Im Schnitt werden pro Instruktion in der MIST-Level 2 Notation 30 Byte benötigt. Die beiden XML-Repräsentationen benötigen mit über 80 Byte durchschnittlich 2,5 mal soviel Speicher pro Instruktion. Insbesondere im Hinblick auf das *Data-Mining* und Algorithmen des maschinellen Lernens ist diese Datenreduktion sehr wichtig, da sich mit ihr auch die Laufzeit und der Speicherbedarf von Analysen auf sehr großen Datensätzen reduzieren.

3.2.4 Zusammenfassung

In Summe zeigen die beiden Experimente die großen Vorteile der MIST- Repräsentation für die dynamische Verhaltensanalyse: Das Verhalten wird durch MIST so dargestellt, dass eine viel genauere Abgrenzung zwischen den verschiedenen Familien von Schadprogrammen möglich wird, bei gleichzeitiger drastischer Reduktion des Ressourcenverbrauchs. Damit bietet MIST die optimale Grundlage, um aufgezeichnetes Verhalten von (Schad-) Programmen effizient und effektiv zu analysieren.

3.3 Automatische Analyse von Schadprogrammen mit MALHEUR und MIST

Das *Malware Instruction Set* wurde mit der Zielsetzung entwickelt, die automatische Analyse von Verhaltensreporten zu optimieren. Um eine optimale Anbindung von MIST an mögliche Analysewerkzeuge zu erreichen, wurde parallel zu MIST und in enger Abstimmung mit MIST das Analyseprogramm MALHEUR [Rieck, 2010] ent-

wickelt. Dieses arbeitet auf in MIST-Notation codierten Verhaltensreporten und kann sowohl zur Clusteranalyse unbekannter Schadprogramme als auch zur Klassifikation von Schadprogrammen bekannter Familien verwendet werden. Im Folgenden wird der Aufbau eines auf MIST und MALHEUR basierenden Analysesystems vorgestellt und die Analyseergebnisse des Systems werden mit den Ergebnissen konkurrierender Systeme verglichen [Rieck et al., 2009, 2011]. Der zum Vergleich der verschiedenen Systeme verwendete Referenzdatensatz wird in Abschnitt 3.3.2 ausführlich beschrieben. Dieser wurde auch zur Evaluierung der in Abschnitt 3.4 vorgestellten MIST-Filter verwendet.

3.3.1 Systementwurf

Schadprogramme zeichnen sich durch sehr unterschiedliches und zum Teil komplexes Verhalten aus. Dieses reicht von einfachen Modifikationen im Dateisystem, über (unbekannte) Netzwerkaktivitäten bis hin zu komplexen Eingriffen in das Betriebssystem. Anhand der beobachteten Verhaltensmuster, wie zum Beispiel der Verwendung eines bestimmten Mutex oder der Veränderung einer speziellen Systemressource, lassen sich Schadprogramme bestimmten Familien zuordnen. Die Gemeinsamkeiten und die damit einhergehenden Abgrenzungen zu anderen Familien können bei einer automatischen Verhaltensanalyse von Schadprogrammen ausgenutzt werden, um mit Hilfe des maschinellen Lernens große Datensätze von Schadprogrammen zu klassifizieren und/oder zu *clustern*. Abbildung 3.8 zeigt einen schematischen Überblick über das vorgeschlagene Analysesystem zur Verhaltensanalyse von Schadprogrammen.

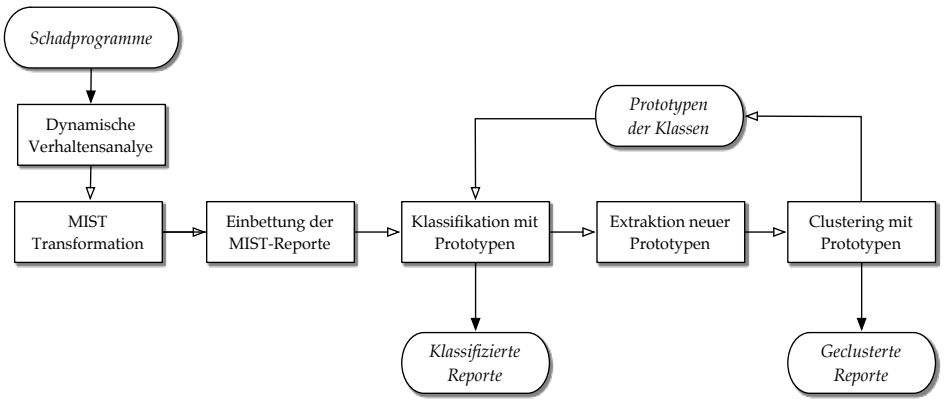


Abbildung 3.8: Schematischer Überblick über das Analysesystem. Neue MIST-codierte Verhaltensreporte werden bekannten Schadprogrammfamilien zugeordnet, oder in neue Cluster eingruppiert. Die während der Clusteranalyse extrahierten Prototypen können anschließend zur Klassifizierung neuer Schadprogramme eingesetzt werden.

Aufzeichnen des Verhaltens als MIST-Report. Im ersten Schritt müssen die Schadprogramme in einer dynamischen Analyseumgebung ausgeführt und beobachtet werden. Basierend auf den dabei aufgezeichneten Instruktionen – in Form von Systemaufrufen – wird ein sequentieller Verhaltensreport für jedes Schadprogramm erstellt.

Dieser Report wird anschließend in die MIST-Repräsentation überführt und dient als Input für die nachfolgenden Schritte.

Einbetten der MIST-Reporte in einen Vektorraum. Die MIST-Reporte werden in einen hoch-dimensionalen Vektorraum eingebettet, in dem jede Dimension genau einem Verhaltensmuster aus zwei aufeinander folgenden MIST-Instruktionen entspricht. In dieser Vektorrepräsentation lässt sich die Gleichheit des Verhaltens auf einen geometrischen Abstand reduzieren, was die Entwicklung und Anwendung von intuitiven und mächtigen Cluster- und Klassifikationsmethoden ermöglicht.

Clustern und Klassifizieren. Auf die in den Vektorraum eingebetteten MIST-Reporte werden Techniken des maschinellen Lernens zum Clustern und Klassifizieren angewandt, um sowohl neue (Clusteranalyse) als auch bekannte (Klassifikation) Verhaltensmuster und damit Schadprogrammfamilien zu identifizieren. Für eine effiziente Berechnung beider Analysetechniken werden Prototyp-Vektoren eingesetzt, die eine große Gruppe von Vektoren (Reporte) mit gleichem Verhalten repräsentieren und damit eine effektive Approximation der exakten Analysen liefern.

Inkrementelle Analyse. Durch das wechselseitige Durchführen jeweils eines Cluster- und Klassifikationsschritts, lässt sich das eingebettete Verhalten von Schadprogrammen inkrementell – beispielsweise täglich – berechnen. Zuerst wird mit Hilfe der in vorhergehenden Clusterläufen extrahierten Prototyp-Vektoren versucht, die neuen Schadprogramme bekannten Clustern (Familien) zuzuordnen. Anschließend werden die übrigen Reporte – diese beschreiben unbekanntes Verhalten – geclustert, um neue Verhaltensmuster (Familien) zu erlernen.

Im Folgenden wird der zur Kalibrierung des Systems verwendete Referenzdatensatz vorgestellt und es werden die Ergebnisse des Systemvergleichs mit konkurrierenden Ansätzen präsentiert.

3.3.2 Referenzdatensatz

Der Referenzdatensatz besteht aus 3.131 Schadprogrammen, die der Datenbasis des MWANALAYSIS-Projekts entnommen worden sind. Das MWANALAYSIS-Projekts sammelt seit über drei Jahren Schadprogramme aus unterschiedlichsten Quellen und analysiert diese in CWSANDBOX. Neben automatisiert sammelnden Werkzeugen, wie HONEYPOTS und SPAMTRAPS, nutzen auch Hersteller von Virenscannern, Forscherteams und viele Privatpersonen den freien Analyseservice. Diese Vielzahl an Quellen garantiert eine breite Abdeckung und große Vielfalt an Schadprogrammen. Alle in der Datenbank enthaltenen Schadprogramme werden sowohl durch CWSANDBOX analysiert, als auch an den freien Onlineservice VIRUS TOTAL [Sistemas] weitergeleitet. VIRUS TOTAL analysiert alle eingehenden Dateien mit derzeit 39 unterschiedlichen Virenscannern. Aus diesen Virenscannern wurden die in Tabelle 3.3 aufgelisteten sechs Virenscanner zum klassifizieren des Refernzdatensatzes ausgewählt.

	Anti-Virus Lösung	ER auf kompl. Datensatz (in %)	ER auf Referenz-datensatz (in %)
1	WEBWASHER-GATEWAY	90,10	94,57
2	IKARUS	88,74	98,59
3	ANTIVIR	85,65	90,48
4	GDATA	78,67	84,60
5	BITDEFENDER	77,24	85,31
6	KASPERSKY	72,16	81,02

Tabelle 3.3: Zur Bestimmung des Referenzdatensatzes herangezogene Virenscanner. ER bezeichnet dabei die Erkennungsrate der Virenscanner.

Die Auswahl basiert auf der durchschnittlichen Erkennungsrate der Virenscanner auf einem Datensatz von über 70.000 Schadprogrammen. Die sechs Virenscanner gehören auf dem Datensatz zu den 10 besten Scannern. Neben der Erkennungsrate wurde die von den Virenscannern gelieferte Klassifizierung als Auswahlkriterium herangezogen. Die Virenscanner sollten den Datensatz nach Möglichkeit in äquivalente Familien – nicht notwendigerweise mit der gleichen Bezeichnung – unterteilen. In Tabelle 3.3 sind für die sechs ausgewählten Virenscanner sowohl die Erkennungsraten auf dem 70.000 Schadprogramme umfassenden Datensatz, als auch die Erkennungsraten auf dem daraus extrahierten Referenzdatensatz aufgelistet. Die Ergebnisse aller 34 von VIRUS TOTAL verwendeten Virenscanner werden in Anhang B dargestellt.

	Schadprogrammklassen	#		Schadprogrammklassen	#
1	ADULTBROWSER	262	*13*	PORNDIALER	97
2	ALLAPLE*	300	*14*	RBOT	101
3	BANCOS	48	*15*	ROTATOR*	300
4	CASINO	140	*16*	SALITY	84
5	DORFDO	65	*17*	SPYGAMES	139
6	EJIK	168	*18*	SWIZZOR	78
7	FLYSTUDIO	33	*19*	VAPSUP	45
8	LDPINCH	43	*20*	VIKINGDLL	158
9	LOOPER	209	*21*	VIKINGDZ	68
10	MAGICCASINO	174	*22*	VIRUT	202
11	PODNUHA*	300	*23*	WOIKOINER	50
12	POSION	26	*24*	ZHELATIN	41

Tabelle 3.4: Zusammensetzung des Referenzdatensatzes. Der Datensatz enthält 3.131 Verhaltensreporte aus 24 Schadprogrammfamilien. Häufig auftretende Familien wurden auf 300 Reporte begrenzt (jeweils angedeutet durch einen Stern*).

Von den gesammelten Schadprogrammen wurden diejenigen als Kandidaten für den Referenzdatensatz herangezogen, die von der Mehrheit der sechs Virenscanner gleich *gelabelt* wurden. Obwohl die von Antivirus Produkten verwendeten Label erwiesenermaßen Inkonsistenzen aufweisen [Bailey et al., 2007], kann der Datensatz durch die Verwendung von sechs unabhängigen Produkten als konsistent und genau genug angesehen werden. Um die ungleich gewichtete Verteilung der Familien zu kompen-

sieren, wurden alle Familien mit weniger als 20 Mitgliedern ausgeschlossen und die restlichen Familien auf maximal 300 Mitglieder beschränkt. Die 3.131 resultierenden Schadprogramme, siehe Tabelle 3.4, wurden erneut in CWSANDBOX analysiert und anschließend in MIST übersetzt.

3.3.3 Vergleich mit konkurrierenden Systemen

Der in Abschnitt 3.3.2 beschriebene Referenzdatensatz wurde sowohl zum Kalibrieren der einzelnen Komponenten des hier vorgestellten Analysesystems, als auch bei einem Vergleich des Gesamtsystems mit konkurrierenden Systemen zur Analyse von Schadprogrammverhalten verwendet. Die Ergebnisse der Clusteranalyse auf den Verhaltensreporten wurden dabei mit der von Bayer et al. [2009] vorgeschlagenen Methode verglichen. Bayer et al. [2009] verwenden ANUBIS SANDBOX zur Analyse der Schadprogramme und extrahieren aus diesen Reporten Verhaltensmerkmale. Auf diesen Merkmalen wird anschließend unter Zuhilfenahme des *Locality Sensitive Hashing (LSH)* [Gionis et al., 1999] ein Clustering berechnet. Als Vergleichssystem für die Klassifikation wurde die von Rieck et al. [2008] vorgestellte Methode herangezogen. Diese lernt mit Hilfe von *Support Vector Maschinen* (SVM) und XML-codierten Verhaltensmerkmalen eine Abgrenzung. Der Vergleich erfolgte jeweils auf dem Referenzdatensatz, wobei für alle Ansätze die jeweils optimalen Einstellungen verwendet wurden.

Clusteringmethode	$F1$-Maß
Clustering mit Prototypen und MIST Level 1	0,950
Clustering mit Prototypen und MIST Level 2	0,936
Clustering mit LSH und Anubis Merkmalen	0,881

Klassifikationsmethoden	F_b	F_u
Klassifikation mit Prototypen und MIST Level 1	0,981	0,997
Klassifikation mit Prototypen und MIST Level 2	0,972	0,964
Klassifikation mit SVM und XML Merkmalen	0,807	0,548

Tabelle 3.5: Vergleich von Analysemethoden auf dem Referenzdatensatz. Die Güte des Clusterings wird über das kombinierte $F1$-Maß angegeben. Für die Klassifikation zeigen die beiden Maße F_b und F_u jeweils einzeln die Güte für die Klassen bekannter und unbekannter Schadprogramme an.

Die Ergebnisse der Vergleiche werden in Tabelle 3.5 dargestellt, wobei das $F1$-Maß als Gütemaß herangezogen wurde. In beiden Fällen liefert das auf MIST basierende Verfahren bessere Ergebnisse als die konkurrierenden Ansätze. Die auf Prototypen basierende Clusteranalyse erreicht ein $F1$-Maß von 0,95 für MIST-Level 1 und 0,936 für MIST-Level 2. Die Methode von Bayer et al. [2009] wird lediglich mit 0,881 bewertet. Ähnlich verhält es sich mit den Klassifikationsergebnissen: Hier liegen die F-Maße für MIST-Level 1 und 2 mindestens bei 0,964 während die Methode von Rieck et al. [2008] für bekannte Schadprogramme nur 0,807 und auf unbekannten Schadprogrammen sogar nur 0,55 erreicht. Die hervorragenden Analyseleistungen des hier vorgestellten Ansatzes basieren auf der geometrischen Formulierung der Clusteranaly-

se und der Klassifikation. Diese nutzt die von der sequentiellen MIST-Repräsentation gelieferten Verhaltensmuster perfekt aus.

An dieser Stelle muss darauf hingewiesen werden, dass die Clusteranalyse von Bayer et al. [2009] unabhängig von den übrigen Analysen an der Technischen Universität Wien durchgeführt wurde. Die Verhaltensanalysen der Schadprogramme erfolgten dabei zu einem späteren Zeitpunkt als die Verhaltensanalysen, auf denen die MIST-Reporte basieren. Das beobachtete Verhalten ist damit nicht exakt dasselbe, was auch ein Grund für die unterschiedliche Leistung der Systeme sein kann. Da das auf MIST basierende Clustering allerdings für mehrere Schadprogrammfamilien ein besseres F-Maß besitzt, ist es eher unwahrscheinlich, dass allein die unterschiedlichen Analysezeiträume für die Ergebnisse verantwortlich sein könnten.

Klasse	# Reports	# Prototypen	# AV Label
C000-0001	126	2	CASINO
C000-0002	298	1	ALLAPLE
C000-0003	48	2	BANCOS
C000-0004	34	2	SALITY
C000-0005	36	3	LDPINCH (25), POISION (11)
C000-0007	50	1	WOIKOINER
C000-0009	40	2	VIKING_DZ
C000-0010	137	1	VIKING_DLL
C000-0011	209	63	LOOPER
C000-0012	21	3	LDPINCH (17), VIRUT (4)
C000-0015	299	2	PODNUHA
C000-0017	45	24	VAPSUP
C000-0018	291	2	ROTATOR
C000-0019	76	2	SWIZZOR
C000-0020	28	2	VIKING_DZ
C000-0022	11	3	POISION
C000-0023	93	2	RBOT (91), SPYGAMES (2)
C000-0031	168	1	EJIK
C000-0034	12	3	SALITY
C000-0039	94	1	PORNDIALER
C000-0050	344	8	VIRUT (198), SPYGAMES (137), RBOT (9)
C000-0058	68	3	DORFDO (65), ZHELATIN (3)
C000-0085	12	2	SALITY
C000-0088	263	2	ADULTBROWSER (262), FLYSTUDIO (1)
C000-0090	21	1	VIKING_DLL
C000-0094	13	8	FLYSTUDIO
C000-0095	174	1	MAGICCASINO

Tabelle 3.6: Details zu dem Cluster- und Klassifikationsergebnis von MALHEUR auf den MIST-codierten Referenzdaten.

Tabelle 3.6 zeigt das detaillierte Ergebnis der Clusteranalyse mit MALHEUR auf MIST-Level 2 Verhaltensreporten. Insgesamt konnten 96,17% der Verhaltensreporte erfolgreich zugeordnet werden. Die übrigen 120 Reporte wurden als *rejected* markiert und würden bei dem vorgestellten inkrementellen Ansatz aus Clusteranalyse und Klassifikation zusammen mit neuen Verhaltensreporten erneut klassifiziert oder *geclustert*. Der überwiegende Anteil aller von dem System erkannten Cluster enthält ausschließlich Reporte zu Schadprogrammen genau einer Familie – entsprechend den Virenscanner Labels. Allerdings verteilen sich verschiedene Familien, wie zum Beispiel VIKING_DZ und SALITY, auf mehrere Cluster. Eine manuelle Analyse der Verhaltensreporte zeigt, dass die entsprechenden Schadprogramme zur Laufzeit unterschiedliches Verhalten zeigten – teilweise aufgrund von unterschiedlichen Umgebungsparametern wie zum Beispiel nicht erreichbare Server oder dem Nachladen unterschiedlicher Schadcodes. Die Aufsplittung der Familien durch MALHEUR ist damit korrekt, da dem Clustering ausschließlich das tatsächliche Verhalten der Schadprogramme zugrunde liegt.

3.3.4 Zusammenfassung

Um der stetig steigenden Anzahl an neuen Schadprogrammen begegnen zu können, sind automatisierte Analysewerkzeuge notwendig, die – möglichst autonom – bekannte Schadprogramme klassifizieren und unbekannte Schadprogramme und Familien erkennen können. Das in diesem Abschnitt vorgestellte, auf MIST und MALHEUR basierende, Analysesystem erfüllt genau diese Anforderungen und liefert dabei bedeutend bessere Cluster- und Klassifikationsergebnisse als konkurrierende Ansätze. Die MIST codierte Verhaltensreporte werden dazu in einen Vektorraum eingebettet und innerhalb dieses geometrischen Raums klassifiziert und/oder geclustert. Die Extraktion von Prototypen ermöglicht dabei zum einen eine inkrementelle Verarbeitung beliebig vieler Verhaltensreporte – ohne das die sonst üblichen Laufzeit- oder Speicherprobleme auftreten – und liefert gleichzeitig Anhaltspunkte für eine detaillierte manuelle Analyse der Familien. Der menschliche Analyst kann sich hierbei auf die die Klasse beschreibenden Prototypen konzentrieren, was den Aufwand erheblich reduziert.

Der hier vorgestellte Ansatz wird seit einem Jahr im Rahmen des MWANALAYSIS-Projekts am Lehrstuhl für Praktische Informatik 1 der Universität Mannheim eingesetzt. Die *394.176* in diesem Zeitraum analysierten Schadprogramme konnten zu *98,83%* in nur *1.032* Klassen geclustert und klassifiziert werden. Mit einer durchschnittlichen Klassengröße von *377* Reporten und *7* Prototypen pro Klasse konnte die Anzahl der von Hand zu untersuchenden Reporte erheblich – auf circa 1,83% aller Reporte – gesenkt werden.

3.4 Musterbasiertes Filtern von Schadprogrammen mit MIST

Basierend auf den Cluster- und Klassifiktionsergebnissen, die mit MALHEUR auf den MIST-codierten Daten gewonnen werden können, ist es möglich, Filter zu konstruieren, mit denen Schadprogramme bekannten Familien zugewiesen werden können (Klassifikation). Wie in Abschnitt 3.3.1 beschrieben, unterteilt MALHEUR die MIST-Verhaltensreporte in Verhaltensmuster aus aufeinander folgenden Instruktionen, kurz n-Gramme. Diese n-Gramme werden von MALHEUR in einen hochdimensionalen Vektorraum abgebildet. In den das Verhalten beschreibenden Vektor fließen damit ausschließlich die in den n-Grammen eines Reports enthaltenen Informationen ein.

Die Idee für die in diesem Abschnitt vorgestellten MIST-Filter ist, die von allen Reporten einer Familie geteilten n-Gramme – also Instruktionsfolgen, die von jedem Schadprogramm der Familie ausgeführt wurden – als Filterkriterien zu verwenden. Werden anschließend in einem unbekannten Report dieselben Instruktionsfolgen gefunden, wird das Schadprogramm der entsprechenden Familie zugeordnet. Gleichzeitig kann man die MIST-Filter als Ausschlussfilter verwenden: Zeigt ein Verhaltensreport keine oder nur zu wenige der charakteristischen Instruktionsfolgen einer Familie, kann das Schadprogramm als „nicht zugehörig“ eingestuft werden. Dieses *Filtern nach Ausschluss* ermöglicht es, die Anzahl möglicher Familien bei der Klassifikation von Schadprogrammen, die Verhaltensmuster unterschiedlicher Familien aufweisen, zu begrenzen.

Der folgende Abschnitt motiviert die Verwendung einer kontextfreien Grammatik zum Filtern von MIST-Reporten und führt die für MIST entworfene Grammatik ein. Anschließend wird der Aufbau der beiden Klassen *MISTLexer* und *MISTParser* zum Parsen von MIST-Reporten beschrieben und es werden die Vorlagen, aus denen diese Klassen und die *MISTFilter* generiert werden, vorgestellt (Abschnitt 3.4.2). In Abschnitt 3.4.3 erfolgt die Evaluierung des Ansatzes auf dem Referenzdatensatz. Darauf folgt eine Zusammenfassung des Konzepts und ein Ausblick auf Erweiterungsmöglichkeiten.

3.4.1 Eine kontextfreie Grammatik für MIST

Die MIST-Filter basieren ausschließlich auf den bei der Clusteranalyse für die einzelnen Familien berechneten charakteristischen n-Grammen. Die optimalen Werte für den Parameter n und das MIST-Level wurden bereits im Rahmen der MALHEUR Entwicklung [Rieck et al., 2011] bestimmt und mit $n = 2$ und dem zweiten MIST-Level belegt. Um die von allen Mitgliedern einer Familie geteilten n-Gramme zu bestimmen, werden die 2-Gramme aller Reporte extrahiert und in ein *Data-Warehouse* eingetragen. Auf den Aufbau des Data-Warehouse wird an dieser Stelle nicht näher eingegangen. Das Datenbank-Schema ist in Abbildung C.1 dargestellt und wird an dieser Stelle erläutert. Die für den Aufbau des Data-Warehouse notwendige Rechenzeit wird durch den Performanzgewinn bei der Bestimmung der von allen Reports geteilten 2-Gramme überkompensiert. Besonders beim Vergleich verschiedener Clusterergebnisse – hier müssen für jeden Clusterlauf die geteilten 2-Gramme jeweils neu bestimmt werden – stellt ein Data-Warehouse eine optimale Datenstruktur dar. Da auf alle 2-Gramme

der MIST-Reporte direkt zugegriffen werden kann, ist es zusätzlich möglich, die nur von einem gewissen Prozentsatz aller Reporte geteilten 2-Gramme zu bestimmen, um beispielsweise einen weicheren Filter zu konstruieren.[4]

Mithilfe des Data-Warehouse lassen sich die von den MIST-Reporten der einzelnen Familien geteilten 2-Gramme sehr einfach bestimmen. Listing 3.2 zeigt die von den als VIRUT bezeichneten Schadprogrammen geteilten Instruktionsfolgen. Die Familie VIRUT wird von lediglich fünf 2-Grammen beschrieben. Zusätzlich handelt es sich dabei ausschließlich um Instruktionsfolgen, bei denen Bibliotheken geladen werden. Folglich ist der auf diesen wenigen Informationen konstruierte Filter erwartungsgemäß sehr unpräzise, siehe hierzu Abschnitt 3.4.3.

```
02 02 | 00006b2c 047c8042 00091000 + 02 02 | 00006b2c 047c8042 00049000
02 02 | 00006b2c 047c8042 00092000 + 02 02 | 00006b2c 047c8042 00011000
02 02 | 00006b2c 047c8042 000b9000 + 02 02 | 00006b2c 047c8042 00108000
02 02 | 00006b2c 047c8042 000aa000 + 02 02 | 00006b2c 047c8042 00092000
02 02 | 00006b2c 047c8042 0000d000 + 02 02 | 00006b2c 047c8042 00011000
```

Listing 3.2: Charakteristische 2-Gramme der VIRUT-Familie.

Auch wenn die Schadprogrammfamilie VIRUT ein schlechtes Beispiel für einen auf 2-Grammen basierenden Filter ist, lässt sich die im Folgenden verwendete kontextfreie Grammatik zum Filtern gut an ihr motivieren. Schon ein regulärer Ausdruck, der überprüft, ob die fünf 2-Gramme in einem MIST-Report enthalten sind, lässt sich nur sehr aufwendig konstruieren. Da mit dem regulären Ausdruck das Vorkommen von fünf nicht zusammenhängenden und nicht untereinander geordneten Instruktionsfolgen überprüft werden soll, muss entweder ein *Dictionary* mit sämtlichen Permutationen – für n-Gramme ergeben sich $n!$ Permutationen – aufgebaut werden oder es muss mit sogenannten *Look-ahead Pattern* gearbeitet werden. Zusätzlich existieren, selbst in diesem Beispiel, schon Überlappungen zwischen den 2-Grammen. Die erste Instruktion des zweiten 2-Gramms entspricht der zweiten Instruktion des vierten 2-Gramms. Folglich muss auch der Fall, dass die beiden 2-Gramme nur überlappend in dem Report vorkommen, berücksichtigt werden. Dies führt im vorliegenden Beispiel zu dem in Listing 3.3 gezeigten regulären Ausdruck mit *144 Teil-Mustern*, oder dem in Listing 3.4 dargestellten regulären Ausdruck mit *Look-ahead Pattern*. Um die Beispiele übersichtlich zu halten, wurden statt der MIST-Instruktionen die Platzhalter (`a-h`) verwendet.

Neben der Größe spricht auch die Unübersichtlichkeit der so konstruierten regulären Ausdrücke gegen den Einsatz einer regulären Sprache. Insbesondere für andere Schadprogrammfamilien mit über 2.000 zu überprüfenden 2-Grammen und hunderten Überlappungen sind reguläre Ausdrücke also nur bedingt geeignet.

[4] In der Zwischenzeit wurde MALHEUR um die hier über das Data-Warehouse umgesetzte Funktionalität erweitert. Nach dem Clustern kann nun eine Liste der von den Reporten geteilten 2-Gramme ausgegeben werden. Auch das Einstellen eines Prozentsatzes zur Konstruktion weicher Filter ist inzwischen möglich.

```
r'.*((ab.*cd.*ef.*gc.*hd)|(cd.*ab.*ef.*gc.*hd)|(cd.*ef.*ab.*gc.*hd)
     |(cd.*ef.*gc.*ab.*hd)|(cd.*ef.*gc.*hd.*ab)|(ab.*ef.*cd.*gc.*hd)
     |(ef.*ab.*cd.*gc.*hd)|(ef.*cd.*ab.*gc.*hd)|(ef.*cd.*gc.*ab.*hd)
     |(ef.*cd.*gc.*hd.*ab)|(ab.*ef.*gc.*cd.*hd)|(ef.*ab.*gc.*cd.*hd)
     |(ef.*gc.*ab.*cd.*hd)|(ef.*gc.*cd.*ab.*hd)|(ef.*gc.*cd.*hd.*ab)
     |(ab.*ef.*gc.*hd.*cd)|(ef.*ab.*gc.*hd.*cd)|(ef.*gc.*ab.*hd.*cd)
     |(ef.*gc.*hd.*ab.*cd)|(ef.*gc.*hd.*cd.*ab)|(ab.*cd.*gc.*ef.*hd)
     |(cd.*ab.*gc.*ef.*hd)|(cd.*gc.*ab.*ef.*hd)|(cd.*gc.*ef.*ab.*hd)
     |(cd.*gc.*ef.*hd.*ab)|(ab.*gc.*cd.*ef.*hd)|(gc.*ab.*cd.*ef.*hd)
     |(gc.*cd.*ab.*ef.*hd)|(gc.*cd.*ef.*ab.*hd)|(gc.*cd.*ef.*hd.*ab)
     |(ab.*gc.*ef.*cd.*hd)|(gc.*ab.*ef.*cd.*hd)|(gc.*ef.*ab.*cd.*hd)
     |(gc.*ef.*cd.*ab.*hd)|(gc.*ef.*cd.*hd.*ab)|(ab.*gc.*ef.*hd.*cd)
     |(gc.*ab.*ef.*hd.*cd)|(gc.*ef.*ab.*hd.*cd)|(gc.*ef.*hd.*ab.*cd)
     |(gc.*ef.*hd.*cd.*ab)|(ab.*cd.*gc.*hd.*ef)|(cd.*ab.*gc.*hd.*ef)
     |(cd.*gc.*ab.*hd.*ef)|(cd.*gc.*hd.*ab.*ef)|(cd.*gc.*hd.*ef.*ab)
     |(ab.*gc.*cd.*hd.*ef)|(gc.*ab.*cd.*hd.*ef)|(gc.*cd.*ab.*hd.*ef)
     |(gc.*cd.*hd.*ab.*ef)|(gc.*cd.*hd.*ef.*ab)|(ab.*gc.*hd.*cd.*ef)
     |(gc.*ab.*hd.*cd.*ef)|(gc.*hd.*ab.*cd.*ef)|(gc.*hd.*cd.*ab.*ef)
     |(gc.*hd.*cd.*ef.*ab)|(ab.*gc.*hd.*ef.*cd)|(gc.*ab.*hd.*ef.*cd)
     |(gc.*hd.*ab.*ef.*cd)|(gc.*hd.*ef.*ab.*cd)|(gc.*hd.*ef.*cd.*ab)
     |(ab.*cd.*ef.*hd.*gc)|(cd.*ab.*ef.*hd.*gc)|(cd.*ef.*ab.*hd.*gc)
     |(cd.*ef.*hd.*ab.*gc)|(cd.*ef.*hd.*gc.*ab)|(ab.*ef.*cd.*hd.*gc)
     |(ef.*ab.*cd.*hd.*gc)|(ef.*cd.*ab.*hd.*gc)|(ef.*cd.*hd.*ab.*gc)
     |(ef.*cd.*hd.*gc.*ab)|(ab.*ef.*hd.*cd.*gc)|(ef.*ab.*hd.*cd.*gc)
     |(ef.*hd.*ab.*cd.*gc)|(ef.*hd.*cd.*ab.*gc)|(ef.*hd.*cd.*gc.*ab)
     |(ab.*ef.*hd.*gc.*cd)|(ef.*ab.*hd.*gc.*cd)|(ef.*hd.*ab.*gc.*cd)
     |(ef.*hd.*gc.*ab.*cd)|(ef.*hd.*gc.*cd.*ab)|(ab.*cd.*hd.*ef.*gc)
     |(cd.*ab.*hd.*ef.*gc)|(cd.*hd.*ab.*ef.*gc)|(cd.*hd.*ef.*ab.*gc)
     |(cd.*hd.*ef.*gc.*ab)|(ab.*hd.*cd.*ef.*gc)|(hd.*ab.*cd.*ef.*gc)
     |(hd.*cd.*ab.*ef.*gc)|(hd.*cd.*ef.*ab.*gc)|(hd.*cd.*ef.*gc.*ab)
     |(ab.*hd.*ef.*cd.*gc)|(hd.*ab.*ef.*cd.*gc)|(hd.*ef.*ab.*cd.*gc)
     |(hd.*ef.*cd.*ab.*gc)|(hd.*ef.*cd.*gc.*ab)|(ab.*hd.*ef.*gc.*cd)
     |(hd.*ab.*ef.*gc.*cd)|(hd.*ef.*ab.*gc.*cd)|(hd.*ef.*gc.*ab.*cd)
     |(hd.*ef.*gc.*cd.*ab)|(ab.*cd.*hd.*gc.*ef)|(cd.*ab.*hd.*gc.*ef)
     |(cd.*hd.*ab.*gc.*ef)|(cd.*hd.*gc.*ab.*ef)|(cd.*hd.*gc.*ef.*ab)
     |(ab.*hd.*cd.*gc.*ef)|(hd.*ab.*cd.*gc.*ef)|(hd.*cd.*ab.*gc.*ef)
     |(hd.*cd.*gc.*ab.*ef)|(hd.*cd.*gc.*ef.*ab)|(ab.*hd.*gc.*cd.*ef)
     |(hd.*ab.*gc.*cd.*ef)|(hd.*gc.*ab.*cd.*ef)|(hd.*gc.*cd.*ab.*ef)
     |(hd.*gc.*cd.*ef.*ab)|(ab.*hd.*gc.*ef.*cd)|(hd.*ab.*gc.*ef.*cd)
     |(hd.*gc.*ab.*ef.*cd)|(hd.*gc.*ef.*ab.*cd)|(hd.*gc.*ef.*cd.*ab)
     |(ab.*gcd.*ef.*hd)|(gcd.*ab.*ef.*hd)|(gcd.*ef.*ab.*hd)
     |(gcd.*ef.*hd.*ab)|(ab.*ef.*gcd.*hd)|(ef.*ab.*gcd.*hd)
     |(ef.*gcd.*ab.*hd)|(ef.*gcd.*hd.*ab)|(ab.*ef.*hd.*gcd)
     |(ef.*ab.*hd.*gcd)|(ef.*hd.*ab.*gcd)|(ef.*hd.*gcd.*ab)
     |(ab.*gcd.*hd.*ef)|(gcd.*ab.*hd.*ef)|(gcd.*hd.*ab.*ef)
     |(gcd.*hd.*ef.*ab)|(ab.*hd.*gcd.*ef)|(hd.*ab.*gcd.*ef)
     |(hd.*gcd.*ab.*ef)|(hd.*gcd.*ef.*ab)|(ab.*hd.*ef.*gcd)
     |(hd.*ab.*ef.*gcd)|(hd.*ef.*ab.*gcd)|(hd.*ef.*gcd.*ab)).*'
```

Listing 3.3: Ein regulärer Ausdruck zum Filtern auf fünf überlappende 2-Gramme.

```
r'.*((ab|cd|ef|gc|hd)(?!.*(\1)).*(ab|cd|ef|gc|hd)(?!.*(\1|\2)).*
     (ab|cd|ef|gc|hd)(?!.*(\1|\2|\3)).*
     (ab|cd|ef|gc|hd)(?!.*(\1|\2|\3|\4)).*
     (ab|cd|ef|gc|hd))|
    ((ab|gcd|ef|hd)(?!.*(\1)).*(ab|gcd|ef|hd)(?!.*(\1|\2)).*
     (ab|gcd|ef|hd)(?!.*(\1|\2|\3)).*(ab|gcd|ef|hd)(?!.*(\1|\2|\3|\4)).*
     (ab|gcd|ef|hd)).*"
```

Listing 3.4: Ein optimierter regulärer Ausdruck für fünf überlappende 2-Gramme.

Ein auf einer kontextfreien Grammatik basierender Filter lässt sich dagegen relativ einfach und übersichtlich konstruieren. MIST ist als Sprache konzipiert und besitzt einen festen Aufbau. Ein Report besteht aus mehreren Threads, die ihrerseits aus einzelnen MIST-Instruktionen bestehen. Der Aufbau der die Threads voneinander abtrennenden *Infozeile* ist fest definiert und auch für die MIST-Instruktionen ist eine gewisse Struktur festgeschrieben. Mit diesen Informationen lässt sich relativ einfach eine kontextfreie Grammatik für MIST konstruieren. Die Grammatik besitzt mit `INFO, SEPARATOR, NEWLINE` und `HEXNUMBER` vier benannte Token und die in Listing 3.5 dargestellten Grammatikregeln.

```
mistreport        : mistthreads

mistthreads       : mistthread
                  | mistthreads mistthread

mistthread        : INFO mistinstruction
                  | empty

mistinstructions  : mistinstruction
                  | mistinstructions mistinstruction

mistinstruction   : apicall SEPARATOR NEWLINE
                  | apicall SEPARATOR attributes NEWLINE
                  | empty

apicall           : section operation

section           : HEXNUMBER

operation         : HEXNUMBER

attributes        : attribute
                  | attributes attribute

attribute         : HEXNUMBER

empty             :
                  | NEWLINE
```

Listing 3.5: Die kontextfreie Grammatik für MIST.

Die Grammatik ist selbsterklärend und bedarf an dieser Stelle keiner weiteren Erläuterung. Stattdessen werden im Folgenden die auf der Grammatik basierenden PYTHON-Klassen *MISTLexer*, *MISTParser* und *MISTFilter* direkt am Quellcode vorgestellt. Die Token und Regeln der Grammatik werden dabei als Variablen und Funktionen definiert. In diesem Zusammenhang wird auch auf die Grammatik selbst genauer eingegangen. Die drei PYTHON-Klassen basieren auf dem Modul PLY, das die eigentliche Übersetzung des Pythoncodes in einen Lexer beziehungsweise Parser übernimmt. Für Details zu PLY wird auf die Dokumentation des Moduls verwiesen [Beazley, 2009]. Bei der Beschreibung der Klassen werden jeweils nur die relevanten Teile des Codes vorgestellt. Der vollständige Quellcode jeder in diesem Abschnitt behandelten Klasse findet sich im Anhang C.

3.4.2 Auf LEX und YACC basierende MIST-Filter

Listing 3.6 zeigt einen Auszug aus dem Quellcode des *MISTLexer*. MIST-Reporte bestehen, wie bereits erwähnt, aus vier *Token*, die in Zeile 10 in die Tokenliste des Lexers eingefügt und in den Zeilen 13 bis 16 definiert werden. Der Token `INFO` passt auf die *Infozeilen*, die die einzelnen Threads eines MIST-Reports einleiten beziehungsweise voneinander abtrennen. Jede dieser Zeilen ist am Anfang und am Ende mit dem Symbol # begrenzt. Dazwischen werden der aufgezeichnete Prozess und der jeweilige Thread angegeben. Der in Zeile 13 definierte reguläre Ausdruck passt genau auf die vorgeschriebene Codierung. Alle Systemaufrufe, deren Kategorien sowie Attribute werden in MIST in hexadezimaler Schreibweise codiert. Der in Zeile 14 definierte reguläre Ausdruck für den Token `HEXNUMBER` passt auf alle Hexadezimalzahlen. Zur Trennung des Systemaufrufs von seinen Argumenten wird in MIST das Sysmbol | verwendet. Der Lexer erkennt dieses Symbol als `SEPARATOR`-Token. Mit Ausnahme der *Info*-Zeilen codieren alle Zeilen eines MIST-Reports jeweils eine MIST-Instruktion. Die Instruktionen sind nur durch den Zeilenumbruch voneinander getrennt, der durch den Token `NEWLINE` definiert ist.

Neben den Token sind für die auf dem Lexer aufbauenden MIST-Filter die in den Zeilen 22 und 23 initialisierten Listen `ngrams` und `check_ngrams` von zentraler Bedeutung. Die beiden Listen werden durch die Filter mit den 2-Grammen gefüllt, die die entsprechende Schadprogrammfamilie charakterisieren. Zur Laufzeit des Parsers werden alle gefundenen 2-Gramme aus der Liste `check_ngrams` entfernt. Der Vergleich der beiden Listen liefert letztlich das Ergebnis des Parsers, und zeigt, ob der MIST-Report zugeordnet werden konnte oder nicht.

Der relevante Teil des Quellcodes der Klasse *MISTParser* ist in Listing 3.7 dargestellt. Insgesamt werden nur 11 Regeln benötigt, um die Syntax von MIST vollständig zu beschreiben. Die ersten beiden Regeln (`p_mistreport` und `p_mistthreads`) beschreiben dabei einen MIST-Report als einen oder mehrere MIST-Threads. Jeder MIST-Thread kann entweder leer sein, oder setzt sich aus einem `INFO`-Token und einer oder mehreren MIST-Instruktionen zusammen, definiert über die Regeln `p_mistthread` und `p_mistinstructions`. Eine MIST-Instruktion kann ebenfalls leer sein. Ist dies nicht der Fall, besteht sie aus einem Systemaufruf, einem `SEPARATOR`-Token, keinem, einem oder mehreren `attribute` Symbol(en) und einem `NEWLINE`-Token, siehe Regeln `p_mistinstruction` und

p_attributes. Der Systemaufruf unterteilt sich in die Symbole Section und Operation, die jeweils auf HEXNUMBER-Token reduziert werden (Regeln p_apicall, p_section und p_opertion). Ein attribute-Symbol besteht ebenfalls aus einem HEXNUMBER-Token (Regel p_attribute). Die letzte Regel ist p_empty und reduziert das leere Wort oder einen Zeilenumbruch.

```
1  """ \mist{Mist}Lexer.py """
2
3  import sys
4  import os
5  import ply.lex as lex
6
7  class MISTLexer(object):
8
9    def __init__(self):
10     self.tokens = ['INFO', 'HEXNUMBER', 'SEPARATOR', 'NEWLINE']
11
12     # Tokens definition
13     self.t_INFO = r'\#\ process\ [0-9a-f]{8}\ [0-9a-f]{8}\ [0-9a-f]{8}\
           [0-9a-f]{8}\ thread\ [0-9a-f]{4}\ \#\n'
14     self.t_HEXNUMBER = r'[0-9a-f]+'
15     self.t_SEPARATOR = r'\|'
16     self.t_NEWLINE = r'\n'
17
18     #Ignored characters
19     self.t_ignore = ' \t'
20
21     #nGrams 2 find
22     self.ngrams = []
23     self.check_ngrams = []
24
25   def t_error(self, t):
26     raise TypeError("Illegal charachter '%s'" % (t.value[0]))
27     pass
28
29   # Build the lexer
30   def build(self, **kwargs):
31     self.lexer = lex.lex(module=self, **kwargs)
32
33   .
34   .
```

Listing 3.6: Die PYTHON-Klasse *MISTLexer*.

Mit Ausnahme der Regel p_mistreport werden alle Regeln ausschließlich zum Parsen des MIST-Reports verwendet. Neben der Reduktion werden dabei keine weiteren Aktionen innerhalb der Regeln durchgeführt. Die Regel p_mistreport wird bei dem erfolgreichen Parsen eines MIST-Reports immer als letzte abgeleitet. Folglich muss an dieser Stelle anhand der check_ngrams Liste des MIST-Lexers kontrolliert werden, ob alle gesuchten 2-Gramme gefunden wurden. Ist die Liste nach dem Parser nicht leer, wurde mindestens ein 2-Gramm nicht gefunden und der Parser löst einen Fehler aus.

```
""" \mist{Mist}Parser.py """

from ply.yacc import yacc

class MISTParser(object):

  # Parsing rules
  def p_mistreport(self, p):
    '''mistreport : mistthreads'''
    if len(self.lexer.check_ngrams) != 0:
      raise
    pass

  def p_mistthreads(self, p):
    '''mistthreads : mistthread
                   | mistthreads mistthread'''
    pass

  def p_mistthread(self, p):
    '''mistthread : INFO mistinstruction
                  | empty'''
    pass

  def p_mistinstructions(self, p):
    '''mistinstructions : mistinstruction
                        | mistinstructions mistinstruction'''
    pass

  def p_mistinstruction(self, p):
    '''mistinstruction : apicall SEPARATOR NEWLINE
                       | apicall SEPARATOR attributes NEWLINE
                       | empty'''
    pass

  def p_apicall(self,p):
    '''apicall : section operation'''
    pass

  def p_attributes(self, p):
    '''attributes : attribute
                  | attributes attribute'''
    pass

  def p_section(self, p):
    '''section : HEXNUMBER'''
    pass

  def p_operation(self, p):
    '''operation : HEXNUMBER'''
    pass

  def p_attribute(self, p):
    '''attribute : HEXNUMBER'''
    pass

  def p_empty(self, p):
    '''empty :
             | NEWLINE'''
    pass

  .
  .
```

Listing 3.7: Die PYTHON-Klasse *MISTParser*.

Mit den hier vorgestellten Klassen *MISTLexer* und *MISTParser* lassen sich alle MIST-Reporte auf eine korrekte Syntax hin untersuchen. Um die MIST-Reporte zusätzlich nach bestimmten 2-Grammen zu durchsuchen, sind sowohl an dem *MISTLexer* als auch an dem *MISTParser* Erweiterungen notwendig. Dabei wird das Konzept der Vererbung angewandt, um für jede zu filternde Schadprogrammfamilie eigene Klassen zu erstellen. Von den Basisklassen *MISTLexer* und *MISTParser* wird jeweils eine Subklasse abgeleitet. Innerhalb der Subklassen werden die zum Suchen von 2-Grammen notwendigen Eigenschaften und Methoden definiert. Die Erweiterungen werden im Folgenden an den Subklassen *VIRUTLexer* und *VIRUTParser* vorgestellt und motiviert.

Listing 3.8 zeigt den Quellcode der Subklasse VIRUTLEXER. Damit der Lexer die MIST-Instruktionen der gesuchten 2-Gramme erkennt, müssen diese der Tokenliste hinzugefügt (Zeilen 9–16) und jeweils über einen regulären Ausdruck definiert werden (Zeilen 18–25). Die einzelnen MIST-Instruktionen werden mit `PATTERN_n` bezeichnet, und die regulären Ausdrücke passen – entsprechend der Grammatikregel für eine MIST-Instruktion im *MISTParser* (siehe Zeile 30 und 31 in Listing 3.7) – immer auf die komplette Zeile der MIST-Instruktion, inklusive dem Zeilenumbruch. Um nach dem erfolgreichen Parsen eines MIST-Reports zu überprüfen, ob die gesuchten Paare von MIST-Instruktionen während dem Parsen gesehen wurden, werden die beiden Listen `ngrams` und `check_ngrams` mit den gesuchten 2-Grammen initialisiert (Zeilen 27 - 31 und Zeile 33 in Listing 3.8).

Die von der *MISTParser* erbende Subklasse *VIRUTParser* muss ebenfalls erweitert werden, um beim Parsern das Vorkommen bestimmter 2-Gramme erkennen zu können. Die Basisklasse erlaubt über die Regel `p_mistinstructions`, dass die Menge der MIST-Instruktionen aus einzelnen MIST-Instruktionen besteht. Der Aufbau einer MIST-Instruktion wird hierbei durch die Grammatikregel `p_mistinstruction` festgelegt. Die Subklasse – im vorliegenden Fall der *VIRUTParser* – muss dagegen beim Ableiten von 2-Grammen auch mögliche Überlappungen zwischen den 2-Grammen berücksichtigen.

Zum Ableiten von 2-Grammen muss die Basisklasse um die in Listing 3.9 aufgeführten Grammatikregeln erweitert werden. Die zweite Regel erlaub das Reduzieren jeder der vier gesuchten 2-Gramme zu dem neuen Grammatiksymbol `ngram`. In Zeile 13 werden die gefundenen 2-Gramme aus der Liste `check_ngrams` des Lexers entfernt. Ein `ngram` kann anschließend über die erste Regel zu dem Grammatiksymbol `mistinstructions` reduziert werden. Ein um diese beiden Grammatikregeln erweiterter Parser kann damit zwar 2-Gramme in einem MIST-Report finden, kommt aber nicht mit Überlappungen zwischen den 2-Grammen zurecht, da sich jedes `PATTERN` nur genau einmal reduzieren lässt. Sollten das zweite und dritte 2-Gramm nur in einer Überlappung in dem zu analysierenden MIST-Report vorkommen, würde ausschließlich das zweite 2-Gramm (`PATTERN_2 PATTERN_3`) erkannt und zu einem `ngram` Symbol reduziert werden[5]. Das im dritten 2-Gramm ebenfalls vorkommende `PATTERN_3` könnte anschließend nicht nochmal abgeleitet werden.

[5] Im vorliegenden Fall überlappen sich das zweite und dritte 2-Gramm in *PATTERN_3*. Damit muss das zweite 2-Gramm in der Überlappung vor dem dritten stehen, da *PATTERN_3* im zweiten 2-Gramm an zweiter und im dritten 2-Gramm an erster Stelle steht.

```
1  """ VIRUTLexer.py """
2
3  from MISTLexer import MISTLexer
4
5  class VIRUTLexer(MISTLexer):
6
7    def __init__(self):
8      \mist{Mist}Lexer.__init__(self)
9      self.tokens.append('PATTERN_0')
10     self.tokens.append('PATTERN_1')
11     self.tokens.append('PATTERN_2')
12     self.tokens.append('PATTERN_3')
13     self.tokens.append('PATTERN_4')
14     self.tokens.append('PATTERN_5')
15     self.tokens.append('PATTERN_6')
16     self.tokens.append('PATTERN_7')
17
18     self.t_PATTERN_0 = r'02\ 02\ \|\ 00006b2c\ 047c8042\ 000b9000\n'
19     self.t_PATTERN_1 = r'02\ 02\ \|\ 00006b2c\ 047c8042\ 00108000\n'
20     self.t_PATTERN_2 = r'02\ 02\ \|\ 00006b2c\ 047c8042\ 000aa000\n'
21     self.t_PATTERN_3 = r'02\ 02\ \|\ 00006b2c\ 047c8042\ 00092000\n'
22     self.t_PATTERN_4 = r'02\ 02\ \|\ 00006b2c\ 047c8042\ 00011000\n'
23     self.t_PATTERN_5 = r'02\ 02\ \|\ 00006b2c\ 047c8042\ 00091000\n'
24     self.t_PATTERN_6 = r'02\ 02\ \|\ 00006b2c\ 047c8042\ 00049000\n'
25     self.t_PATTERN_7 = r'02\ 02\ \|\ 00006b2c\ 047c8042\ 0000d000\n'
26
27     self.ngrams.append(('02\ 02\ \|\ 00006b2c\ 047c8042\ 000b9000\n',
           '02\ 02\ \|\ 00006b2c\ 047c8042\ 00108000\n'))
28     self.ngrams.append(('02\ 02\ \|\ 00006b2c\ 047c8042\ 000aa000\n',
           '02\ 02\ \|\ 00006b2c\ 047c8042\ 00092000\n'))
29     self.ngrams.append(('02\ 02\ \|\ 00006b2c\ 047c8042\ 00092000\n',
           '02\ 02\ \|\ 00006b2c\ 047c8042\ 00011000\n'))
30     self.ngrams.append(('02\ 02\ \|\ 00006b2c\ 047c8042\ 00091000\n',
           '02\ 02\ \|\ 00006b2c\ 047c8042\ 00049000\n'))
31     self.ngrams.append(('02\ 02\ \|\ 00006b2c\ 047c8042\ 0000d000\n',
           '02\ 02\ \|\ 00006b2c\ 047c8042\ 00011000\n'))
32
33     self.check_ngrams = list(self.ngrams)
```

Listing 3.8: Die für die VIRUT-Familie abgeleitete PYTHON-Klasse *VIRUTLexer*.

```
1   def p_mistinstructions_ngram(self, p):
2     '''mistinstructions : ngram
3                         | mistinstructions ngram'''
4     pass
5
6   def p_ngram(self, p):
7     '''ngram : PATTERN_0 PATTERN_1
8              | PATTERN_2 PATTERN_3
9              | PATTERN_3 PATTERN_4
10             | PATTERN_5 PATTERN_6
11             | PATTERN_7 PATTERN_4'''
12    try:
13      self.lexer.check_ngrams.remove((self.escape(p[1]),
            self.escape(p[2])))
14    except:
15      pass
16    pass
```

Listing 3.9: Erweiterungen der PYTHON-Klasse *VIRUTParser* zum Ableiten von 2-Grammen.

```
1   def p_mistinstructions_ngram(self, p):
2     '''mistinstructions : ngram
3                 | mistinstructions ngram'''
4     pass
5
6   def p_ngram(self, p):
7     '''ngram : pattern'''
8     pass
9
10  def p_pattern(self, p):
11    '''pattern : pattern pattern'''
12    p[0] = p[1]
13    try:
14      self.lexer.check_ngrams.remove((self.escape(p[1]), self.escape(p
            [2])))
15    except:
16      pass
17    pass
18
19  def p_pattern_one(self, p):
20    '''pattern : PATTERN_0
21               | PATTERN_1
22               | PATTERN_2
23               | PATTERN_3
24               | PATTERN_4
25               | PATTERN_5
26               | PATTERN_6
27               | PATTERN_7'''
28    p[0] = p[1]
29    pass
```

Listing 3.10: Erweiterungen des *VIRUTParser* zum Ableiten von überlappenden 2-Grammen.

Um das Problem der Überlappungen zu lösen, muss der Reduktionsschritt für das Grammatiksymbol `ngram` auf drei Regeln verteilt werden. Listing 3.10 zeigt die entsprechenden Grammatikregeln. Ein `ngram` kann hierbei nur aus dem zusätzlichen Grammatiksymbol `pattern` abgeleitet werden. Dieses `pattern` wiederum kann entweder von genau zwei `pattern` (Grammatikregel `p_pattern`) oder einem `PATTERN`-Token abgeleitet werden (Grammatikregel `p_pattern_one`). Die auf den ersten Blick überflüssig erscheinende Grammatikregel `p_pattern` ist notwendig, um bei der Reduktion das vordere der beiden `pattern` zu duplizieren und dem Parser erneut vorzulegen. Das Duplizieren findet in Zeile 12 statt. Hier wird das Ergebnis der Reduktion (`p[0]`) auf das erste `pattern` (`p[1]`) gesetzt. Das `pattern` wird dem Parser erneut übergeben und kann von diesem ein zweites Mal reduziert werden.

Der *VIRUTFilter* erzeugt jeweils eine Instanz des *VIRUTLexer* und des *VIRUTParser*. Dabei wird dem Parser die Lexer-Instanz übergeben. Im Anschluss wendet der Filter diese auf einen oder mehrere MIST-Reporte an. Dazu stellen sowohl der *MISTLexer* als auch der *MISTParser* eine Funktion `test` bereit, siehe Anhang C. Falls beim fehlerfreien Parsen alle gesuchten n-Gramme gefunden werden, liefert der Filter ein *positives* Ergebnis zurück.

MIST-Filter Vorlagen

Um für alle bei einer beliebigen Clusteranalyse erkannten Klassen einen entsprechenden MIST-Filter erstellen zu können, wurden zwei Klassen *MALWARELexer* und *MALWAREParser* und das Script *MALWAREFilter* entwickelt. Aus diesen drei Vorlagen, einem Bezeichner für die Familie und den gesuchten 2-Grammen lassen sich die notwendigen PYTHON-Klassen und -Scripte automatisch erstellen. Der Bezeichner muss dabei für alle drei Vorlagen identisch sein, da dieser ebenfalls als Klassenbezeichner genutzt wird und damit auch beim Importieren der beiden Klassen in den Filter Verwendung findet. Die im letzten Abschnitt beschriebenen Klassen *VIRUTLexer* und *VIRUTParser* sowie der dazugehörige Filter *VIRUTFilter* (siehe Anhang C) wurden automatisiert aus den Vorlagen und den in dem Data-Warehouse gespeicherten Informationen erstellt. Auch der dabei eingesetzte *MISTLexYacc_ParserGenerator* ist in Anhang C abgebildet. Dieser baut direkt auf dem Data-Warehouse auf und erstellt für alle darin definierten Cluster den Lexer, Parser und Filter.

Die Listings 3.11 und 3.12 zeigen die Vorlagen für den *MALWARELexer* und den *MALWAREParser*. Innerhalb des *MALWARELexer* müssen die folgenden Informationen angegeben werden: Das Pattern `$$MALWARE_FAMILY$$` wird durch den Bezeichner der Schadprogrammklasse ersetzt. Der gewählte Bezeichner darf keine Sonderzeichen enthalten, da er auch in den Namen der erstellten PYTHON-Klasse eingeht. Für jede in den gesuchten 2-Grammen vorhandene MIST-Instruktion wird an dem Pattern `$$TOKEN_NAMES$$` eine Zeile der Form

```
self.tokens.append('PATTERN_X')
```

eingefügt. Diese erzeugt einen Token für die Instruktion. Die Definition der eingefügten `PATTERN_X` erfolgt an der Stelle `$$TOKEN_DEFINITIONS$$`. Für jedes `PATTERN` muss dabei eine Zeile der Form

`self.t_PATTERN_X = r'regular_expression'`

eingefügt werden. Jedes der gesuchten 2-Gramme wird schließlich in der folgenden Codierung an der Stelle `$$NGRAMS_LIST$$` eingefügt:

`self.ngrams.append(('regex_1', 'regex_2'))`

```
1  #!/usr/bin/env python
2  # encoding: utf-8
3  """
4  $$MALWARE_FAMILY$$Lexer.py
5  Created by Philipp Trinius on 2010-03-09.
6  Copyright (c) 2010 University of Mannheim. All rights reserved.
7  """
8
9  from MISTLexer import MISTLexer
10
11 class $$MALWARE_FAMILY$$Lexer(MISTLexer):
12
13   def __init__(self):
14     \mist{Mist}Lexer.__init__(self)
15     $$TOKEN_NAMES$$
16     $$TOKEN_DEFINITIONS$$
17     $$NGRAMS_LIST$$
18
19     self.check_ngrams = list(self.ngrams)
```

Listing 3.11: Die Klassen-Vorlage MALWARELEXER.

Die in Abbildung 3.12 gezeigte Vorlage des *MALWAREParser* benötigt weniger Informationen über die Schadprogrammklasse. Der gewünschte Bezeichner ersetzt auch hier `$$MALWARE_FAMILY$$`, und die im *MALWARELexer* definierten Token werden als mögliche rechte Seite der Reduktionsregel `p_pattern_one` definiert (`$$TOKEN_LIST$$`).

In der *MALWAREFilter*-Vorlage, siehe Listing 3.13, wird lediglich der Bezeichner `$$MALWARE_FAMILY$$` ersetzt. Dieser Bezeichner wird in dem Skript auch innerhalb der Import-Anweisungen für den Lexer und den Parser verwendet.

Mit Hilfe der hier beschriebenen Vorlagen und den im Data-Warehouse abgelegten Informationen über die MIST-Reporte und die verschiedenen Klassen lassen sich einfach neue MIST-Filter für beliebige Klassen erstellen. Die zur Evaluierung des Ansatzes verwendeten Filter wurden alle mit dem *MISTLexYacc_ParserGenerator* erstellt, siehe Listing C.4. Dieser wiederum verwendet die drei Vorlagen und das Data-Warehouse.

```
#!/usr/bin/env python
# encoding: utf-8
"""
$$MALWARE_FAMILY$$Parser.py
Created by Philipp Trinius on 2010-03-09.
Copyright (c) 2010 University of Mannheim. All rights reserved.
"""

from MISTParser import MISTParser

class $$MALWARE_FAMILY$$Parser(MISTParser):

  def __init__(self):
    \mist{Mist}Parser.__init__(self)

  def p_mistinstructions_ngram(self, p):
    '''mistinstructions : ngram
                | mistinstructions ngram'''
    if len(p) == 2:
      p[0] = p[1]
    else:
      p[0] = p[2]
    pass

  def p_ngram(self, p):
    '''ngram : pattern'''
    p[0] = p[1]
    pass

  def p_pattern(self, p):
    '''pattern : pattern pattern'''
    p[0] = p[1]
    try:
      self.lexer.check_ngrams.remove((self.escape(p[1]), self.escape(p
          [2])))
    except:
      pass

  def p_pattern_one(self, p):
    '''pattern : $$TOKEN_LIST$$'''
    p[0] = p[1]
    pass
```

Listing 3.12: Die Klassen-Vorlage *MALWAREParser*.

```
#!/usr/bin/env python
# encoding: utf-8
"""
$$MALWARE_FAMILY$$Filter.py
Created by Philipp Trinius on 2010-03-09.
Copyright (c) 2010 University of Mannheim. All rights reserved.
"""

import sys
import os
from $$MALWARE_FAMILY$$Lexer import $$MALWARE_FAMILY$$Lexer
from $$MALWARE_FAMILY$$Parser import $$MALWARE_FAMILY$$Parser

if __name__ == '__main__':
  param = sys.argv[1]
  # Build the lexer
  m = $$MALWARE_FAMILY$$Lexer()
  m.build()
  # Build the parser
  p = $$MALWARE_FAMILY$$Parser()
  p.build(m, 'mistreport')

  if os.path.isdir(param):
    output = 'LexYacc Output of ' +sys.argv[0]+ ' on ' +sys.argv[1]+ '\n'
    files = os.listdir(param)
    for fname in files:
      if os.path.isfile(param + "/" + fname):
        print "Checking " + fname,
        output += "Checking " + fname
        ffile = file(param + "/" + fname, "r")
        report = ffile.read()
        ffile.close()
        if m.test(report):#, debug=True):
          print '\tLex ok',
          output += '\tLex ok'
          if p.test(report):#, debug=True):
            print '\tYacc ok'
            output += '\tYacc ok\n'
          else:
              print '\tYacc failed!'
              output += '\tYacc failed!\n'
      else:
        print '\tLex failed!\n'
        output += '\tLex failed!\n'
      ffile = open("$$MALWARE_FAMILY$$.txt", "w")
      ffile.write(output)
      ffile.close()
  elif os.path.isfile(param):
    print "Checking " + param,
    ffile = file(param, "r")
    report = ffile.read()
    ffile.close()
    if m.test(report):
      print '\tLex ok',
      if p.test(report):
        print '\tYacc ok'
      else:
        print '\tYacc failed!'
    else:
      print '\tLex failed!\n'
```

Listing 3.13: Die Skript-Vorlage *MALWAREFilter*.

3.4.3 Auswertung der MIST-Filter

Zur Evaluation der MIST-Filter wurde der Referenzdatensatz herangezogen. Dabei wurden die MIST-Filter sowohl für die von den Virenscanner bestimmten Klassen als auch für die mit MALHEUR bestimmte Cluster erstellt. Tabelle 3.7 zeigt die Anzahl der Token und 2-Gramme, für die aus den Virenscanner-Labeln resultierenden MIST-Filter. Im Schnitt teilen die MIST-Reporte jeder Klasse 236 2-Gramme und 197 Token. Mit 2.299 2-Grammen aus 2.127 Token besitzt die BANCOS-Klasse den mit Abstand am genauesten definierten Filter. Der gröbste Filter gehört zu der VIRUT-Klasse. Er besitzt lediglich 8 Token und 5 2-Gramme.

MIST Filter	**# Token**	**# 2-Gramme**
ADULTBROWSER	394	527
ALLAPLE	58	68
BANCOS	2127	2299
CASINO	167	228
DORFDO	115	120
EJIK	98	137
FLYSTUDIO	47	53
LDPINCH	13	10
LOOPER	115	166
MAGICCASINO	74	102
PODNUHA	36	39
POISON	21	23
PORNDIALER	151	193
RBOT	15	15
ROTATOR	385	572
SALITY	9	7
SPYGAMES	98	112
SWIZZOR	156	168
VAPSUP	238	343
VIKING_DLL	124	188
VIKING_DZ	91	105
VIRUT	8	5
WOIKOINER	145	181
ZHELATIN	22	22

Tabelle 3.7: Details zu den auf den Virenscanner-Labels trainierten MIST-Filtern. Der genaueste Filter ist hellgrau und der ungenaueste dunkelgrau hinterlegt.

Die Ergebnisse der MIST-Filter auf dem Referenzdatensatz sind in Tabelle 3.8 dargestellt. Erwartungsgemäß hat jeder MIST-Filter alle Reporte seiner Klasse erkannt, das heißt, es gibt keine *false negatives*. Zwanzig der 24 Filter haben ausschließlich die Reporte ihrer Familie erkannt, also auch keine *false positives* gefunden. Den gröbsten fehlerfreien MIST-Filter stellt dabei mit lediglich 7 2-Grammen der Filter für die Schadprogrammfamilie SALITY dar. Mit den Filtern für die Schadprogramme POISON, LDPINCH, RBOT und VIRUT schlagen insgesamt vier Filter auch bei Reporten an, die nicht zu ihrer jeweiligen Familie gehören. Insgesamt verhält sich die Anzahl

der *false positives* innerhalb dieser vier MIST-Filter umgekehrt proportional zu der Anzahl der verwendeten Token und 2-Grammen. Während in den Funden des POISON-Filter nur ein *false positives* enthalten ist, sind bei dem RBOT-Filter mit 65,65 Prozent rund zwei Drittel aller Funde falsch. Mit 92,62 Prozent *false positives* liefert der VIRUT-Filter das mit Abstand schlechteste Ergebnis.

MIST Filter	# Funde	# false positiv	# false negatives
ADULTBROWSER	262	0	0
ALLAPLE	300	0	0
BANCOS	48	0	0
CASINO	140	0	0
DORFDO	65	0	0
EJIK	168	0	0
FLYSTUDIO	33	0	0
LDPINCH	96	53	0
LOOPER	209	0	0
MAGICCASINO	174	0	0
PODNUHA	300	0	0
POISON	27	1	0
PORNDIALER	97	0	0
RBOT	294	193	0
ROTATOR	300	0	0
SALITY	84	0	0
SPYGAMES	139	0	0
SWIZZOR	78	0	0
VAPSUP	45	0	0
VIKING_DLL	158	0	0
VIKING_DZ	68	0	0
VIRUT	2743	2541	0
WOIKOINER	50	0	0
ZHELATIN	41	0	0

Tabelle 3.8: Ergebnisse der auf den Virenscanner-Labels trainierten MIST-Filter.

Die Clusteranalyse mit MALHEUR basiert auf den 2-Grammen der MIST-Reporte. Folglich sollten die einer Klasse zugeordneten Reporte mehr gemeinsame 2-Gramme teilen, als die auf den Virenscanner-Labels basierenden Gruppen. Damit werden die MIST-Filter genauer und liefern weniger *false positives*. In Tabelle 3.9 sind die 27 MIST-Filter mit der jeweiligen Anzahl an Token und 2-Grammen aufgelistet. Im Durchschnitt besitzt jeder Filter 471 Token und 585 2-Gramme. Das entspricht dem 2,4 beziehungsweise 2,5-fachen an Token und 2-Grammen der auf Virenscanner-Labels basierten MIST-Filter.

Der ungenaueste MIST-Filter (C000-0085) besitzt 12 2-Gramme aus ebenfalls 12 Token und passt auf genau 53 Reporte der SALITY Schadprogrammfamilie. Damit erkennt der Filter zwar weit mehr Reporte als zu seiner Konstruktion verwendet wurden – der Filter wurde für ein Cluster mit 12 Reporten erstellt – es werden aber keine Reporte anderer Schadprogrammfamilie von diesem Filter erkannt. Auch der detaillier-

MIST Filter	# Token	# 2-Gramme	MIST Filter	# Token	# 2-Gramme
C000-0001	167	229	C000-0020	91	105
C000-0002	970	1466	C000-0022	109	136
C000-0003	2127	2300	C000-0023	97	112
C000-0004	2560	2621	C000-0031	98	137
C000-0005	102	99	C000-0034	1151	1560
C000-0007	145	181	C000-0039	151	193
C000-0009	892	1470	C000-0050	87	93
C000-0010	1080	1332	C000-0058	104	102
C000-0011	115	166	C000-0085	12	12
C000-0012	30	27	C000-0088	355	406
C000-0015	58	73	C000-0090	126	190
C000-0017	238	343	C000-0094	52	58
C000-0018	1073	1602	C000-0095	74	102
C000-0019	657	691			

Tabelle 3.9: Details zu den auf den MALHEUR-Daten trainierten MIST-Filter.

teste MIST-Filter wurde für die SALITY Schadprogrammfamilie erstellt.[6] Mit 2.560 Token und 2.621 2-Grammen erkennt der Filter für das Cluster C000-0004 dabei genau die Reporte aus denen er konstruiert wurde. Die Anzahl der von den einzelnen MIST-Filtern erkannten Reporte, sowie die Familien denen die Reporte angehören, sind in Tabellen 3.10 dargestellt.

Auf den ersten Blick wirken die Ergebnisse der MIST-Filter, die auf den durch MALHEUR bestimmten Clustern konstruierten wurden, schlechter als die Ergebnisse der auf den Virenscanner-Labeln trainierten Filtern. Während bei letzteren nur 16,67 Prozent der Filter *false positives* liefern, sind es bei den auf MALHEUR aufbauenden MIST-Filtern 29,63 Prozent. Bei genauerer Betrachtung zeigt sich allerdings, dass die Fehler weit weniger ins Gewicht fallen. Fünf der acht Filter mit *false positives* (C000-0004, C000-0020, C000-0039, C000-0085 und C000-0094) wurden jeweils für genau eine Schadprogrammfamilie entwickelt. Sämtliche *false positives* Funde sind dabei auch Reporte zu Schadprogrammen derselben Familie. Auch für die verbleibenden drei MIST-Filter – diese basieren auf Reporten unterschiedlicher Familien – gilt: Es werden nur Reporte aus Familien zugeordnet, die auch in die Konstruktion des MIST-Filters eingeflossen sind. Diese Beobachtung und das Fehlen eines extrem schlechten Filters, wie dem VIRUT-Filter, machen die auf den Ergebnissen der Clusteranalyse erstellten MIST-Filter insgesamt genauer, als die auf den Virenscanner-Labels erstellten MIST-Filter.

Dass die Filter nicht an das Ergebnis von MALHEUR heranreichen, ist mit der Beschränkung auf die geteilten 2-Gramme zu begründen. In der von MALHEUR durchgeführten Clusteranalyse werden alle in den Reporten enthaltenen 2-Gramme berücksichtigt, also auch die, die einen Report von den übrigen unterscheiden. Dies hat zur Folge, dass ein Verhaltensreport der alle charakteristischen 2-Gramme mit den Mitgliedern eines Clusters teilt, aber zusätzlich noch eine Vielzahl weiterer Instruktionsfolgen

[6] Insgesamt wurde die SALITY Familie von MALHEUR in die drei Cluster C000-0004, C000-0034 und C000-0085 unterteilt, vergleiche Tabelle 3.6.

besitzt, von MALHEUR nicht dem Cluster zugeordnet wird. Das globale Verhalten des Programms weicht in diesem Fall zu weit von dem der übrigen Programme ab. Der MIST-Filter arbeitet ausschließlich mit den geteilten 2-Grammen. Folglich wird der Report von dem MIST-Filter dem Cluster zugeordnet. Ein weiteres Problem stellen unzusammenhängende Cluster dar. Der von MALHEUR verwendete Algorithmus garantiert nicht, dass überhaupt 2-Gramme von allen Reporten eines Clusters geteilt werden. Ein MIST-Filter für ein solches Cluster müsste somit weicher konstruiert werden (siehe Ausblick in 3.4.4).

MIST Filter	# Funde	# Familien	# AV Label
C000-0001	126	1	CASINO
C000-0002	298	1	ALLAPLE
C000-0003	48	1	BANCOS
C000-0004	34	1	SALITY
C000-0005	46	2	POISON (21), LDPINCH (25)
C000-0007	50	1	WOIKOINER
C000-0009	40	1	VIKING_DZ
C000-0010	137	1	VIKING_DLL
C000-0011	209	1	LOOPER
C000-0012	21	2	LDPINCH (17), VIRUT (4)
C000-0015	299	1	PODNUHA
C000-0017	45	1	VAPSUP
C000-0018	291	1	ROTATOR
C000-0019	76	1	SWIZZOR
C000-0020	68	1	VIKING_DZ
C000-0022	11	1	POISON
C000-0023	237	2	SPYGAMES (137), RBOT (100)
C000-0031	168	1	EJIK
C000-0034	12	1	SALITY
C000-0039	97	1	PORNDIALER
C000-0050	437	3	VIRUT (198), SPYGAMES (139), RBOT (100),
C000-0058	68	2	DORFDO (65), ZHELATIN (3)
C000-0085	53	1	SALITY
C000-0088	263	2	ADULTBROWSER (262), FLYSTUDIO (1)
C000-0090	21	1	VIKING_DLL
C000-0094	27	1	FLYSTUDIO
C000-0095	174	1	MAGICCASINO

Tabelle 3.10: Ergebnisse der auf den MALHEUR-Daten trainierten MIST-Filter. Die grau hinterlegten Zeilen markieren dabei die Filter, bei denen neben den zum Erstellen des Filters eingesetzten Reporten noch weitere als zugehörig erkannt wurden.

An dieser Stelle muss erwähnt werden, dass bei der Clusteranalyse 120 Reporte als *rejected* markiert wurden. Diese Reporte flossen damit nicht in einen der erstellten MIST-Filter ein, wurden den MIST-Filtern aber anschließend auch vorgelegt. Von die-

sen 120 Reporten wurden 25 von einem MIST-Filter erkannt und der entsprechenden Klasse zugeordnet. Die Übrigen besitzen nicht genügend Überschneidungen mit den MIST-Filtern.

3.4.4 Zusammenfassung

In diesem Abschnitt wurde ein auf LEX und YACC basierender Filteransatz für Verhaltensreporte in MIST-Codierung vorgestellt. Als Filterkriterium werden ausschließlich die von allen Reporten einer Klasse geteilten n-Gramme herangezogen. Die Verwendung der MIST-Codierung zur Verhaltensrepräsentation bietet sich an, da sich der einfache und fest definierte Aufbau der MIST-Sprache leicht in einer Grammatik abbilden lässt. In der sonst bei der Darstellung von Verhaltensreporten üblichen XML-Codierung stellt sich dies bedeutend aufwendiger dar.

Zum Filtern werden mit dem *MALWARELexer*, dem *MALWAREParser* und dem *MALWAREFilter* drei Klassen beziehungsweise Skripte eingesetzt, die für jede zu filternde Schadprogrammfamilie erzeugt werden müssen. Dazu werden fest definierte Vorlagen verwendet, die zusammen mit dem Data-Warehouse für MIST-Instruktionen eine schnelle Generierung verschiedener Filter erlauben. Innerhalb des *MALWARELexer* werden dabei die gesuchten MIST-Instruktionen und die sich aus diesen zusammensetzenden n-Gramme definiert. Der *MALWAREParser* importiert diesen *MALWARELexer* und kann dank geschickt geschachtelter Reduktionsregeln auch Reporte ableiten, in denen die gesuchten n-Gramme nur mit Überlappungen enthalten sind. Der *MALWAREFilter* schließlich wendet den Lexer und Parser auf einen oder eine Menge von MIST-Reporten an.

Zur Evaluierung des Ansatzes wurde der Referenzdatensatz herangezogen. Dieser wurde zum einen auf Basis der Virenscanner-Labels und zum anderen mit Hilfe von MALHEUR in 25 beziehungsweise 27 Klassen unterteilt. Für jede dieser Klassen wurde ein *MALWAREFilter* erzeugt und gegen alle Reporte des Referenzdatensatzes geprüft. Die Ergebnisse auf beiden Klassifizierungen sind vielversprechend, auch wenn sich das Ergebnis der auf den MALHEUR-Daten trainierten Filtern erwartungsgemäß besser darstellt.

Ausblick

Die MIST-Filter arbeiten momentan ausschließlich auf den von allen Reporten einer Klasse geteilten 2-Grammen. Diese hundertprozentige Abdeckung der verwendeten 2-Gramme durch alle Reporte liefert auf dem Referenzdatensatz für beide betrachteten Custeranalysen gute Ergebnisse. Trotzdem erscheint es sinnvoll, auch Filter mit 2-Grammen zu erstellen, die nur von beispielsweise 80 Prozent der Reporte geteilt werden. Dadurch könnten genauere Filter mit mehr 2-Grammen konstruiert werden, allerdings müsste die Zugehörigkeit zu der Klasse über eine untere Schranke für mindestens enthaltene 2-Gramme bestimmt werden. MALHEUR wurde bereits erweitert, um auch die von einem beliebigen Prozentsatz von Reporten geteilten n-Gramme berechnen zu können. Im *MISTParser* lässt sich diese Erweiterung einfach umsetzen, indem die Längenabfrage in Zeile 10

```
if len(self.lexer.check_ngrams) != 0:
```

durch einen Längenvergleich der beiden `ngram`-Listen ersetzt wird. Für einen Wert von 80 Prozent müsste die folgende Zeile eingesetzt werden:

```
if len(self.lexer.check_ngrams) <= (len(self.lexer.ngrams) * 0.2):
```

Eine weitere Möglichkeit, die Genauigkeit der Filter zu erhöhen besteht darin, mehr Informationen aus dem MIST-Report als Filterkriterium zu nutzen. Denkbar sind hier längere Instruktionssequenzen oder auch die Anzahl an aufgezeichneten Prozessen und Threads sowie deren Position innerhalb des Reports. Auch die Aufnahme von diskriminativen Instruktionen oder n-Grammen könnte helfen, die Genauigkeit der Filter weiter zu steigern.

3.5 Zusammenfassung und Ausblick

Zusammenfassung

Das *Malware Instruction Set* (MIST) ist eine Metasprache für Verhaltensreporte, die eine Vergleichbarkeit zwischen unterschiedlichen Analyseumgebungen herstellen kann. Durch die MIST-Repräsentation wird erstmals eine optimale Datenbasis geschaffen, die es ermöglicht, Techniken des Data Mining und des maschinellen Lernens auf die dynamische Verhaltensanalyse anzuwenden. In MIST werden die sehr detaillierten Verhaltensreporte auf die zentralen und aussagekräftigen Informationen (Argumente) reduziert. Die hier zum ersten Mal konsequent durchgeführte Neuordnung aller Argumente erlaubt dabei das Einführen von unterschiedlichen Detailstufen innerhalb eines Reports. Durch die komplett neue und auf die maschinelle Verarbeitung optimierte Notation wird zusätzlich der Speicherbedarf jeder Instruktion und damit der gesamten Reporte drastisch reduziert. Die optionale Verwendung von *Hash-Tabellen* zum Speichern der komprimierten Informationen erlaubt die vollständige Rückübersetzung der Informationen. Insgesamt können durch die Verwendung von MIST-Reporten sowohl die Laufzeit und der Speicherbedarf, als auch die Ergebnisse von Analysen, die auf *Data Mining* und dem maschinellen Lernen basieren, verbessert werden.

Mit der Einführung von MIST als klar strukturierte Sprache zur Beschreibung von Programmverhalten ist es zudem möglich, einfache Filter für Verhaltensreporte zu konstruieren, die bei der Klassifikation von neuen Verhaltensreporten genutzt werden können. Durch die Verwendung einer kontextfreien Sprache wurden leicht verständliche Filter konstruiert, die erfolgreich auf einem Referenzdatensatz getestet werden konnten. Die momentanen Filter nutzen die Mächtigkeit kontextfreier Sprachen nicht annähernd aus und besitzen darum ein noch weit größeres Potential, als es im Rahmen dieser Arbeit gezeigt werden konnte.

Ausblick

Zur Zeit ist das *Malware Instruction Set* der einzige Ansatz, der versucht, die Lücke zwischen der Verhaltensanalyse und den Methoden des maschinellen Lernens zu schließen. Aus diesem Grund sollen weitere Transformationswerkzeuge entwickelt werden, mit denen sich die Verhaltensreporte weiterer Analysesysteme wie ANUBIS

oder JOEBOX in MIST übersetzen lassen. Die Firma GFI arbeitet momentan an der festen Integration von MIST in die GFI SANDBOX, wodurch sich MIST als Metasprache für Verhaltensreporte etablieren könnte.

An MIST selbst sind aktuell keine grundlegenden Änderungen geplant. Da fast alle Analyseumgebungen zur Verhaltensanalyse nur einen Ausführungspfad aufzeichnen, erscheint die Umstellung der MIST-Reporte auf eine Baumstruktur nicht notwendig. Diese Umstellung wäre nötig, wenn auch Verhaltensreporte, die mehrere Ausführungspfade eines Programms beschreiben, dargestellt werden sollen.

Im Zusammenhang mit der Clusteranalyse und den MIST-Filtern ist eine Aufsplittung der MIST-Reporte in die darin enthaltenen Prozesse und Threads angedacht. Auf Prozess- und/oder Threadebene durchgeführte Analysen können interessante Ergebnisse über die Entwicklung von Schadprogrammen liefern. So könnten Threads mit festen Schadfunktionen, wie dem *Entpacken von Code*, oder dem *Nachladen und Ausführen von Schadprogrammen*, in Verbindung gebracht werden. Basierend darauf ließe sich ein noch genaueres Bild der Verwandtschaft zwischen verschiedenen Schadprogrammfamilien entwickeln.

KAPITEL IV

Musterbasierte Filter für Spam-Nachrichten

Für die meisten Nutzer stellt Spam nur einen lästigen Bestandteil des Internets dar. Der einzige für den Nutzer unmittelbar sichtbare „Schaden“ besteht in einem überfüllten Postfach. Der tatsächlich durch Spam verursachte Schaden, wie verlorene Arbeitszeit oder der unnötige Verbrauch von Ressourcen, ist den meisten Nutzern jedoch nicht bewusst. Auch die von Spam ausgehenden Sicherheitsrisiken werden von den Nutzern nicht wahrgenommen.

Kontext. Neben dem wirtschaftlichen Schaden stellt Spam inzwischen auch ein Sicherheitsrisiko dar. Mit dem „Härten“ der Betriebssysteme weisen diese immer weniger Schwachstellen auf. Sie bieten Angreifern damit fast keine Angriffsfläche mehr, was zu einer Neuorientierung seitens der Angreifer geführt hat. Aktuell werden die meisten Rechner über Schwachstellen in Client-Anwendungen kompromittiert. Bei der Entwicklung von Client-Anwendungen wurden Sicherheitsaspekte lange Zeit vernachlässigt, was zahlreiche, teils gravierende Schwachstellen in der Implementierung und/oder dem Design der Anwendungen zur Folge hat. Als Angriffsvektor kommen hierbei in der Regel Spam-Nachrichten zum Einsatz. Diese enthalten entweder infizierte Dokumente als Dateianhang oder dem Opfer wird im E-Mail Text der Link auf eine bösartig präparierte Webseite präsentiert, entsprechend angepriesen und/oder verschleiert. Öffnet das Opfer diese Webseite in seinem Browser, kommt es zu einem sogenannten *Drive-by*-Angriff. Bei diesem nutzen auf dem Server platzierte Skripte mögliche Schwachstellen im Browser oder in einem in diesem eingebundenen *Plugin* aus, um Schadcode auf dem Opfer-System zur Ausführung zu bringen [Provos et al., 2007, Wang et al., 2006]. Die Spam-Nachrichten können in diesem Zusammenhang als sehr zuverlässiger Angriffsvektor eingestuft werden, der dem Angreifer zusätzlich den Vorteil verschafft, dass die nachgelagerte Kommunikation vom Opfer ausgeht und damit klassische Sicherheitsmechanismen wie Firewalls nicht anschlagen können.

Problemstellung. Die meisten Spam-Nachrichten können mit aktuellen Filtern zuverlässig gefiltert werden. Trotzdem garantiert keiner der Filter eine *false-positive* Rate von 0 Prozent. Dies führt dazu, dass die als Spam klassifizierten Nachrichten meist nur entsprechend markiert oder verschoben werden. Letztlich muss weiterhin

jeder Nutzer selbst entscheiden, ob es sich bei den Nachrichten um Spam handelt oder nicht. Des Weiteren sind alle aktuellen Spam-Filter reaktive Systeme, die erst reagieren können, wenn die Spam-Nachricht mehrfach gesehen wurde, oder im Zusammenhang mit Spam schon einmal aufgetretene Daten, Server, URLs u.a. wiederverwendet werden. Eine entsprechend geplante Spam-Welle wird folglich – zumindest kurzfristig – immer Erfolg haben und zugestellt werden.

Der überwiegende Anteil an Spam wird aktuell über Botnetze versendet. Die verschickten Nachrichten werden dabei erst von den Bots selbst, aus vordefinierten Mustern für die einzelnen Spam-Kampagnen und *Fülldaten* wie Empfänger- und Absenderlisten, zusammengesetzt. Die verwendeten Muster bieten hierbei einen neuen Ansatzpunkt zum Filtern. Mit den in diesem Kapitel vorgestellten Filtern wird das Ziel verfolgt, wenigstens für den Bereich des Botnetz-Spam einen zuverlässigen Spam-Filter ohne Fehlklassifikationen zu konstruieren, der zeitnah nach dem Ausbruch einer neuen Spam-Welle zur Verfügung steht.

Ansatz. Das in dieser Arbeit vorgestellte System verfolgt einen *proaktiven* Ansatz. Statt passiv auf die Zustellung von Nachrichten zu warten und anschließend einen zu den Spam-Nachrichten passenden Filter zu erstellen, wird direkt mit den für die Spam-Nachrichten verantwortlichen Spambots interagiert. Die Spambots werden dazu in einer kontrollierten Umgebung ausgeführt, die es ihnen erlaubt, mit dem jeweiligen *Command & Control* Server zu kommunizieren und die von diesem erhaltenen Befehle auch auszuführen. Sobald die Bots mit dem Verschicken von E-Mail-Nachrichten beginnen, werden diese Nachrichten am ausgehenden *Gateway* abgefangen und zum Aufbau der Filter verwendet. Da die E-Mail-Nachrichten direkt beim Sender abgefangen werden, liegen relativ schnell genügend Nachrichten vor, um das Muster hinter den beobachteten Spam-Kampagnen zu verstehen und Filter für die laufenden Kampagnen zu konstruieren. Jeder Filter muss dabei immer nur eine Kampagne – eine Spam-Welle kann durchaus aus mehreren Kampagnen bestehen – abdecken und kann somit sehr spezifisch sein. Die Wahrscheinlichkeit, dass normale, gutartige E-Mail-Nachrichten von dem Filter fälschlich als Spam klassifiziert werden, ist somit als sehr gering einzustufen.

Da nur eine sehr geringe Anzahl an Botnetzen für den Großteil aller Spam-Nachrichten verantwortlich ist (siehe Abschnitt 2.2.3), ist es möglich, mit relativ wenig Ressourcen alle großen Botnetze kontinuierlich zu überwachen und Filter für die einzelnen Spam-Kampagnen zu erzeugen. Die fertigen Filter können anschießend sowohl auf E-Mail Servern, als auch in den E-Mail-Clients eingesetzt werden, um den von dem Botnetz versendeten Spam zuverlässig zu filtern und damit das Spam-Aufkommen weiter zu reduzieren.

Ergebnisse. Durch den proaktiven Ansatz können Spam-Filter für aktuelle Spam-Kampagnen schon kurz nach dem Ausbruch der Spam-Kampagne bereitgestellt werden. Die zur Evaluierung der sprach-basierten Filteransätze implementierten Prototypen benötigen dabei nur wenige Spam-Nachrichten, um sehr effektive Filter für die einzelnen Kampagnen zu berechnen. So erreichen die auf einer kontextfreien Grammatik basierenden Filter für hunderte Kampagnen des STORM-Botnetzes schon nach dem

Lernen von jeweils einer einzelnen Spam-Nachricht durchschnittliche Erkennungsraten von über 99 Prozent. Das heißt, die in die Kampagne gestreute Varianz – im STORM-Botnetz kommt ein musterbasiertes Spamverfahren zum Einsatz – hat in diesem Fall nahezu keinen Einfluss auf die Güte des Spam-Filters.

Ausblick auf das Kapitel. Das Kapitel gliedert sich wie folgt: In Abschnitt 4.1 werden verwandte Arbeiten vorgestellt. Der Systemaufbau wird in Abschnitt 4.2 beschrieben und in Abschnitt 4.3 ein erster, auf regulären Ausdrücken basierender Ansatz für die Filter vorgestellt und evaluiert. Die auf regulären Ausdrücken basierenden Filter besitzen verschiedene Nachteile, die in Abschnitt 4.6 diskutiert werden. Sie dienen gleichzeitig als Motivation für den ebenfalls in diesem Abschnitt vorgestellten, auf kontextfreien Grammatiken basierenden Filteransatz. Nach der Evaluation dieser Filter wird das Kapitel in Abschnitt 4.7 zusammengefasst.

4.1 Verwandte Arbeiten

Zum Thema Spam-Filtertechniken existieren zahlreiche Arbeiten. Die klassischen Methoden, wie Blacklists oder Bayes-Filter wurden bereits in Abschnitt 2.2.2 vorgestellt. Aus diesem Grund sind in diesem Abschnitt nur diejenigen Arbeiten aufgeführt, die sich mit musterbasiertem Spam befassen und/oder einen ähnlichen, proaktiven Ansatz verfolgen.

Kreibich et al. [2008] veröffentlichten mit ihrer Studie zum STORM WORM Botnetz die erste Arbeit, die das Verfahren des musterbasierten Spam für Botnetze im Detail beschreibt. Die Möglichkeit, die von den Bots verwendeten *Templates* bedingten Regelmäßigkeiten innerhalb der Nachrichten zum Filtern zu verwenden, wurde erstmals von Stern [2008] erwähnt, dort aber noch nicht umgesetzt.

Zeitgleich mit dem in dieser Arbeit vorgestellten System zum Filtern der Nachrichten mit regulären Ausdrücken [Göbel et al., 2009] wurden von Xie et al. [2008] und John et al. [2009] ähnliche Ansätze zum Analysieren aktueller Spambots vorgestellt. In beiden Systemen werden die Spambots in einer kontrollierten Umgebung ausgeführt und die von den Bots generierten Spam-Nachrichten am lokalen Gateway abgegriffen. Bei der Auswertung der Nachrichten konzentrieren sich beide Systeme lediglich auf die in den Nachrichten enthaltenen URLs. Im Unterschied zu diesen Systemen wird in dem hier vorgestellten Ansatz der gesamte Nachrichtentext ausgewertet und zur Konstruktion eines Filters verwendet, der die in der Spam-Kampagne verwendeten Muster beschreibt.

Das von Pitsillidis et al. [2010] vorgestellte System ist nahezu identisch mit dem von Göbel et al. [2009] veröffentlichten proaktiven Analysesystem. Beide Systeme verwenden reguläre Ausdrücke um die Templates zu beschreiben. Pitsillidis et al. [2010] erweitern unseren Ansatz dabei um *Dictionaries* für ausgewählte, variable Textpassagen, die mit in die regulären Filterausdrücke einfließen.

4.2 Aufbau des Filtersystems

Traditionelle Spam-Filter werden mit zwei Datensätzen trainiert. Neben den Spam-Nachrichten werden auch normale E-Mail-Nachrichten ausgewertet, um die beiden Nachrichtentypen voneinander abgrenzen zu können. Bei den daraus resultierenden Filterregeln kann es sich um einfache *Blacklists* handeln, zum Beispiel bestimmten URLs, die nicht in der Nachricht enthalten sein dürfen, oder, wie im Fall des Bayes-Filters, um komplexe Modelle, die den Aufbau einer Spam-Nachricht beschreiben. Eine gute Regel zeichnet sich dabei durch die folgenden drei Eigenschaften aus:

1. Die Regel ist genau genug und verursacht keine oder nur wenige *false-positives*.
2. Die Regel ist robust gegenüber kleinen Veränderungen.
3. Die Regel liegt bereits kurz nach dem ersten Auftreten einer neuen Spam-Kampagne vor.

Besonders die letzte Bedingung lässt sich in klassischen Ansätzen nur schwer erfüllen, da die Spam-Nachrichten erst in ausreichender Anzahl empfangen werden müssen, bevor eine Regel erstellt werden kann. Wird die Spam-Kampagne nicht bereits durch eine existierende Regel abgewehrt, bietet der Spam-Filter während dieses Zeitraums erwartungsgemäß auch keinen Schutz.

Das hier vorgestellte System versucht, die Verzögerung beim Erstellen von neuen Regeln durch einen proaktiven Ansatz zu minimieren. Statt auf die Zustellung der Spam-Nachrichten zu warten, werden gezielt Spambots in einem sogenannten *Sandnet* ausgeführt und überwacht. Sobald eine neue Spam-Kampagne beginnt, werden die Nachrichten direkt am ausgehenden Gateway abgegriffen und zum Lernen von neuen Regeln verwendet. Die Regeln werden anschließend an die E-Mail-Clients verteilt, wo sie auf die empfangenen Nachrichten angewendet werden. Abbildung 4.1 stellt den Systemaufbau schematisch dar.

Das Abgreifen der zu lernenden Spam-Nachrichten an der Quelle und das Filtern am Ziel der Nachrichten bietet ausreichend Zeit, um präzise Filter zu erstellen, die auf alle Nachrichten einer Spam-Kampagne passen ohne *false-positives* zu generieren. An dieser Stelle wird nochmals darauf hingewiesen, dass mit dem vorgestellten System ausschließlich von Botnetzen verschickter und auf Mustern basierender Spam sowie SOCKS-Spam analysiert und gefiltert werden können. Sogenannte *Targeted Attacks*, also gezielte Angriffe, bei denen personalisierte Nachrichten in geringer Anzahl verschickt werden, werden mit diesem System nicht erkannt.

Das System besteht aus drei Teilen: der **Monitorumgebung** für die Spambots (Sandnet), dem **Filtergenerator** zum Erstellen der Regeln und dem **Filter**, der in die E-Mail-Clients integriert wird. Im Folgenden werden die Schnittstellen zwischen den drei Bausteinen und die Monitorumgebung kurz beschrieben. Für den Filtergenerator existieren zwei Implementierungen, die in den Abschnitten 4.3 und 4.4 ausführlich vorgestellt werden. Auf das Filtern der Nachrichten innerhalb der E-Mail-Clients wird am Ende dieses Abschnitts eingegangen.

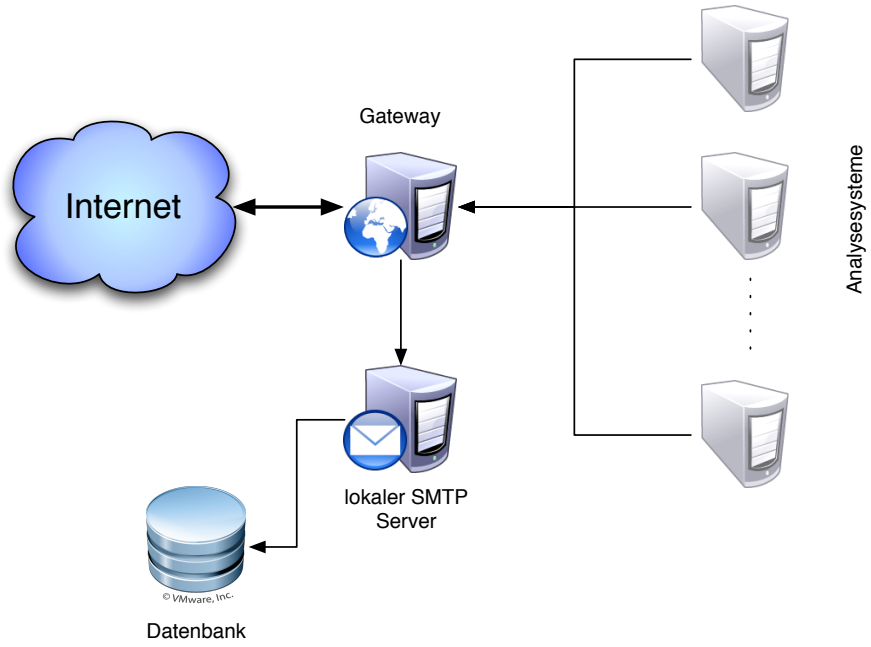


Abbildung 4.1: Systemaufbau zur Analyse von Spambots.

4.2.1 Die Monitorumgebung – das *Sandnet*

Damit die Spam-Nachrichten zum Lernen gesammelt werden können, muss entweder der Original-Bot in einer kontrollierten Umgebung ausgeführt oder die Funktionalität des Bot emuliert werden. Um den Original-Bot sicher auszuführen – „sicher“ bezieht sich in diesem Zusammenhang auf Systeme und Netzwerke von Dritten, die in keinem Fall durch den Bot angegriffen und kompromittiert werden dürfen – werden sogenannte *high-interaction Honeypot*s eingesetzt. Diese stellen voll funktionsfähige Systeme dar, die detailliert überwacht werden und das System in der Regel durch eine *Honeywall* von dem Internet abtrennen [Andreolini et al., 2005, Spitzner, 2002]. Der Vorteil des ersten Ansatzes liegt darin, dass keinerlei Codeanalyse notwendig ist. Der Spambot wird unverändert ausgeführt und die Spam-Nachrichten können ohne großen Aufwand abgegriffen werden. Neben dem Überwachen der Analyseumgebung stellt hierbei das zuverlässige und automatische Zurücksetzen des Systems in einen sauberen Zustand die größte Herausforderung dar. Diese Probleme existieren beim zweiten Ansatz nicht. Hier wird der Spambot lediglich emuliert und es werden nur vorab implementierte Operationen ausgeführt. Dieser Ansatz wird auch als *low-interaction Honeypot* bezeichnet, da der Angreifer – in diesem Fall der Command&Control-Server – nur fest definierte Operationen auf dem System ausführen kann. Damit wird die Analyseumgebung während der Analyse nicht kompromittiert und es geht folglich auch keinerlei Gefahr von ihr aus. Die Analyseumgebung selbst muss weniger genau überwacht werden und nach der Analyse auch nicht zurückgesetzt werden. Ein weiterer Vorteil ist darin zu sehen, dass bedeutend weniger Ressourcen benötigt werden, als

beim ersten Ansatz. Jeder *high-interaction* Honeypot kann zu jedem Zeitpunkt immer nur einen Spambot ausführen, das heißt, jeder Spambot belegt einen Rechner. Ein *low-interaction* Honeypot kann dagegen auch mehrere Bots emulieren und auch das Ausführen mehrerer *low-interaction* Honeypots auf einem Rechner ist ohne Probleme möglich. Der größte Nachteil von *low-interaction* Honeypots besteht darin, dass vorab der Code und die Netzwerkkommunikation des Bots eingehend analysiert werden müssen. Wie in Abschnitt 2.1.2 erwähnt, existiert eine Vielzahl von Techniken, um die Codeanalyse zu erschweren und auch die Netzwerkkommunikation des Bots kann von seinem Entwickler durch den Einsatz von Verschlüsselung effektiv abgesichert werden.

In dem hier vorgestellten System wird momentan nur der *high-interaction Honeypot* Ansatz verfolgt. Die Spambots werden dabei auf einem nativen Windows-System ausgeführt und können frei mit dem C&C-Server kommunizieren. In der Regel baut der Bot kurz nach seiner Aktivierung eine Verbindung zu dem C&C-Server auf, registriert sich bei diesem und wartet auf weitere Kommandos. Die Kommunikation zwischen dem Bot und seinem C&C-Server muss dabei von der Monitorumgebung überwacht werden, um Schaden für Dritte, beispielsweise durch einen verteilten *Denial of Service* Angriff, abzuwenden.

Fungiert der Bot nur als Proxy für das Versenden der Spam-Nachrichten, wird er einen `SOCKS`-Tunnel aufbauen, über den der Bot die zu versendenden Nachrichten empfängt. Anderenfalls erhält der Spambot ein Template für die Spam-Kampagne und die Adresslisten. Aus diesen Informationen baut er die Spam-Nachrichten anschließend selbst zusammen. In der Regel werden diese Informationen verschlüsselt an den Bot übertragen. Sind das verwendete Verfahren und der Schlüssel vorab bekannt, kann bei diesem Verfahren das Spam-Template direkt extrahiert und in eine Filterregel konvertiert werden. Ansonsten muss, genau wie bei den als Proxy eingesetzten Bots, gewartet werden, bis der Bot beginnt, Spam-Nachrichten zu verschicken.

Sämtliche von dem Bot ausgehenden SMTP-Verbindungen werden an dem lokalen *Gateway* auf einen lokalen SMTP-Server umgeleitet und von diesem bearbeitet. Versucht ein Bot beispielsweise eine Verbindung zu *smtp.google.com* aufbauen, wird die Kommunikation lokal umgeleitet und direkt von dem lokalen SMTP-Server bedient. Um diese Umleitung für den Bot möglichst transparent vorzunehmen, kann der lokale SMTP-Server zusätzlich noch das `Banner` des echten E-Mail Servers nachladen und die Verbindung mit dem Spambot unter diesem Banner aufbauen. Sobald die Verbindung steht, beginnt der Spambot mit dem Ausliefern der E-Mail-Nachrichten an den lokalen Server. Sämtliche Nachrichten werden lokal im *mbox*-Format gespeichert und können anschließend dem Filtergenerator übergeben werden.

Da in diesem Ansatz die Original-Bots zur Ausführung gebracht werden, ist der Honeypot anschließend kompromittiert und muss vor der Analyse weiterer Spambots wieder in einen sauberen Systemzustand überführt werden. Dazu wird das in Abschnitt 2.1.2 beschriebene Wiederherstellungssystem eingesetzt. Der Zeitpunkt, zu dem das Analysesystem beendet und zurückgesetzt wird, lässt sich nicht optimal bestimmen. Prinzipiell muss deshalb immer zwischen der Güte der Filter und der Dauer der Analyse abgewogen werden. Eine zu früh abgebrochene Analyse kann zu sehr spezifischen Filtern führen, weil nicht genügend Varianz in den abgegriffenen Spam-Nachrichten enthalten ist, um den Filter auf das tatsächlichen Template der Spam-

Kampagne zu reduzieren. Mit zunehmender Analysedauer nimmt in der Regel auch die Genauigkeit des Filters zu, allerdings verzögert sich die Auslieferung an die E-Mail-Clients. Eine einfache Heuristik zur Bestimmung des Zeitpunkts stellen feste Werte für die Analysezeit oder die Anzahl zu sammelnder Spam-Nachrichten dar. Komplexere Strategien können zum Beispiel die Filterregel selbst als Abbruchkriterium heranziehen: Sind an dem Filter über einen gewissen Zeitraum keine Änderungen notwendig, kann der Filter als ausreichend genau angesehen und die Analyse beendet werden. Da ein Spambot parallel mehrere Spam-Kampagnen durchführen kann, müssen mehrere Filter erzeugt und zur Bestimmung des Abbruchzeitpunkts herangezogen werden.

Momentan arbeitet das vorgestellte System mit festen Analysezeiten, die vom Analysten für jeden Bot frei gewählt werden können. Der Standardwert beträgt 6 Minuten. In diesem Zeitraum können meist tausende Nachrichten abgegriffen werden, die ausreichen, um gute Filter zu erstellen. Die hohe Analysezeit stellt zur Zeit kein Problem dar, da noch keine produktiv eingesetzten E-Mail-Clients auf die neuen Filter warten.

4.2.2 Filtergenerator

Mit Hilfe der im Sandnet gesammelten Nachrichten lassen sich proaktive Spam-Filter konstruieren. Da es sich bei den gesammelten Nachrichten ausschließlich um Spam handelt, können beispielsweise die in den Nachrichten enthaltenen Links sofort in eine UR-Blacklist aufgenommen werden. Eine detaillierte Analyse der Nachrichten liefert in den meisten Fällen zusätzliche, Bot-spezifische Artefakte, anhand derer sich Nachrichten, die von dem Botnetz verschickt wurden, identifizieren lassen. Bei den Artefakten handelt es sich zum Beispiel um spezielle Header-Felder, deren Anordnung, oder auch untypische Werte für Felder. Zusätzlich lassen sich durch die Analyse tausender Nachrichten aus einer Spam-Kampagne Rückschlüsse auf das in der Kampagne verwendete *Template* ziehen. Sämtliche Nachrichten der Kampagne müssen auf dieses Muster passen und ein dieses Muster beschreibender Filter kann im Ergebnis zuverlässig und fehlerfrei alle Nachrichten der Kampagne herausfiltern.

In dem vorgestellten System wurden nacheinander zwei verschiedene Filterformen implementiert und ausgewertet. Der erste Filter wird in Abschnitt 4.3 vorgestellt und verwendet einen Algorithmus zur Berechnung des *Longest Common Substring (LCS)*, um die in allen Nachrichten enthaltenen Teilsequenzen zu extrahieren. Anschließend werden aus den Informationen reguläre Ausdrücke konstruiert, die zum Filtern eingesetzt werden können. Der im darauf folgenden Abschnitt eingeführte Filteransatz beschreibt die Spam-Nachricht als kontextfreie Grammatik und nutzt LEX und YACC zum Filtern der Nachrichten. Damit lassen sich mächtigere und gleichzeitig leicht verständliche Filter für musterbasierte Spam-Nachrichten entwickeln.

4.2.3 Filterplugin für E-Mail-Clients

Das Plugin zum Filtern der Nachrichten [Gräßlin, 2010] wurde in das AKONADI-Framework [KDE-PIM, 2010] integriert und kann damit theoretisch von jeder auf AKONADI aufsetzenden E-Mail-Anwendung verwendet werden. Aktuell existiert nur die ebenfalls von Grässlin implementierte Anbindung an KMAIL. Die neuen Filter werden dem Filter-Plugin über einen *Really Simple Syndication (RSS)-Feed* bereit-

gestellt. Das Plugin liest die neuen Filter periodisch aus und wendet diese auf alle Nachrichten im Posteingang an.

4.3 Filtern mit regulären Ausdrücken

Bei musterbasierten Spam-Kampagnen werden sogenannte Vorlagen (*Templates*) verwendet, die die Grundstruktur der E-Mails beinhalten. Neben der Vorlage erhält der Spambot noch eine Liste von E-Mail-Adressen, an die die Nachricht verschickt werden soll, sowie *Fülltext*, der an markierten Stellen in der Vorlage eingesetzt werden kann. Bei dem Fülltext kann es sich um Wortlisten aber auch ganze Textblöcke handeln, die entsprechend der Markierungen in die Vorlage eingesetzt werden. In Abbildung 4.2 sind Teile einer Vorlage und eines Fülltextes aus dem XARVESTER-Botnetz dargestellt.

```
1  ...
2  From: "{teml:do base64e(6, "windows-1251", teml:var("sendername"))}" ...
3  To: <{receiver_addr}>
4  Subject: {teml:var subj_val}
5  Date: {localdatetime}
6  ...
7
8  This is a multi-part message in MIME format.
9
10 --{teml:var boundary}
11 Content-Type: text/plain;
12         cahrset="iso-8859-1"
13 Content-Transfer-Encoding: quoted-printable
14
15 {file "body.html", html-plain-quoted-printable}
16 --{teml:var boundary}
17 Content-Type: text/html;
18         cahrset="iso-8859-1"
19 Content-Transfer-Encoding: quoted-rpintable
20
21 {file "body.html", quoted-printable}
22
23 --{teml:var boundary}--
```

(a) Auszüge aus der Vorlage einer Spam-Kampagne.

```
...
<DIV align=center><FONT face=Arial><STRONG></STRONG></FONT> </DIV>
<DIV align=center><FONT face=Arial><STRONG><A href="{rndline "links.txt}
    ">Browse on shop</A></STRONG></FONT></DIV></BODY></HTML>
...
```

(b) Auszüge aus der in die Vorlage einzubindenden Datei *body.html*.

Abbildung 4.2: Beispiel für musterbasierte Spam-Nachrichten [Neale, 2009].

Sämtliche geklammerten Ausdrücke der Vorlage werden von dem Spambot ersetzt. Neben den Informationen für die Header-Felder erzeugt der Bot einen zufälligen Wert für die *Multipart*-Grenzen und setzt die Datei `body.html` in den E-Mail-*Body* der

Nachricht ein. Der HTML-Code dieser Datei enthält weitere variable Teile, die ebenfalls ersetzt werden müssen. Im vorliegenden Fall wird dazu eine Zeile der Datei `links.txt` ausgelesen und als Ziel eines HTML-Links eingesetzt.

In der Regel werden die Vorlage und die übrigen Informationen und Dateien verschlüsselt an den Bot übertragen. Das Ziel des Filtergenerators ist es daher, aus den im Sandnet gesammelten Spam-Nachrichten das ursprüngliche Template so gut wie möglich zu extrahieren, um es anschließend als Filter für die Spam-Kampagne einsetzen zu können.

4.3.1 Filter-Generierung

Um die von allen Nachrichten der Spam-Kampagne geteilten Textblöcke, also die verwendete Vorlage, zu identifizieren, wird das PYTHON Modul TEMPLATEMAKER verwendet [Holovaty, 2007]. Dieses basiert auf einem Algorithmus für den *Longest Common Substring (LCS)* und erlaubt, den gemeinsamen Text aus zwei identisch formatierten Texten zu extrahieren. Die unterschiedlichen Textpassagen können entweder ignoriert oder durch einen Platzhalter markiert werden. Über den im Folgenden beschriebenen Algorithmus werden sukzessiv die von allen gesammelten Nachrichten verwendeten Textpassagen extrahiert. Nur diese in allen Nachrichten vorkommenden Teile können auch in der ursprünglich an den Bot übermittelten Vorlage enthalten gewesen sein. Die nicht konstanten Bestandteile der E-Mail werden erst von dem Spambot eingesetzt. Der zur Extraktion der Vorlage verwendete Algorithmus besitzt die folgenden fünf Schritte:

1. Als erstes liest der Algorithmus alle während eines Analyselaufs abgegriffenen E-Mail-Nachrichten ein und sortiert sie der Länge nach. Dadurch wird erreicht, dass die längsten Nachrichten, also die mit den meisten Informationen, immer zuerst verarbeitet werden.
2. Im zweiten Schritt wird die erste Nachricht aus der sortierten Liste entnommen. Diese Nachricht stellt die erste Vorlage dar, die mit α bezeichnet wird.
3. Anschließend wird die nächste Nachricht der Liste entnommen und mit α über den LCS-Algorithmus zu einer zweiten Vorlage β verschmolzen. Die Vorlage β ist somit um einen Schritt genauer als α.
4. Jetzt werden die beiden Vorlagen α und β miteinander verglichen und es wird der Textanteil bestimmt, der in β durch Platzhalter ersetzt wurde. Liegt der Prozentsatz an verloren gegangenem Text unter einer Schranke ϵ, wird β das neue α und mit Schritt 3 fortgefahren. Liegt der Prozentsatz an ersetztem Text über der Schranke ϵ, ist das aktuelle β zu generisch und wird verworfen. Die E-Mail, mit der β generiert wurde, wird an das Ende der E-Mail-Liste angehangen und anschließend wird mit Schritt 3 fortgefahren. Durch den Vergleich der Textanteile fließen nur diejenigen Nachrichten in die finale Vorlage ein, die den Informationsgehalt von α nicht zu stark reduzieren.
5. Solange noch Nachrichten in der E-Mail-Liste enthalten sind, wird mit Schritt 3 fortgefahren. Ab einem gewissen Zeitpunkt werden die in diesem Schritt erstellten Vorlagen β immer wieder verworfen, das heißt die Vorlage α bleibt konstant. An diesem Punkt ist die Vorlagen-Extraktion abgeschlossen und α wird als er-

ste Filter-Vorlage ausgegeben. Anschließend wird mit Schritt 1 fortgefahren und die nächste Vorlage aus den noch in der E-Mail Liste enthaltenen Nachrichten extrahiert.

Der Algorithmus extrahiert in der Regel mehrere Filter-Vorlagen. Im schlechtesten Fall kann das eine Filter-Vorlage für jede abgegriffene Nachricht sein. In diesem Fall wurde von dem analysierten Spambot kein musterbasiertes Spam-Verfahren verwendet. Da der Filteransatz speziell für musterbasierte Spam-Kampagnen entwickelt wurde, sollte die Filter-Vorlage daher verworfen werfen. Falls bei der Analyse Links oder andere bot-spezifische Artefakte in den Nachrichten gefunden wurden, können stattdessen diese zum Trainieren klassischer Filtermethoden verwendet werden.

Um die extrahierten Filter-Vorlagen als Filter nutzen zu können, müssen sie in reguläre Ausdrücke überführt werden. Dazu müssen die Platzhalter in den Filter-Vorlagen ersetzt werden. Die einfachste Möglichkeit stellt das Ersetzen der Platzhalter durch das reguläre Pattern $.*$ dar. Damit passt der Filter auf alle bei der Vorlagen-Extraktion verwendeten Nachrichten, es gibt also keine *false negatives* beim Filtern. Allerdings wird der Filter dadurch auch sehr ungenau. Ist der konstante Teil der Nachrichten nur gering, entsteht durch dieses Ersetzen ein sehr generischer Filter, was beim Filtern in der Regel zu *false positives* führt. Gerade diese Fehler sollten aber durch das musterbasierte Filtern vermieden werden.

Statt für jeden Platzhalter beliebige Texte zuzulassen, werden alle in die Filter-Vorlage eingeflossenen Nachrichten untersucht. Dabei werden für jeden Platzhalter die an dieser Stelle verwendeten Textblöcke extrahiert und ausgewertet. Für jeden Textblock wird ein regulärer Ausdruck erstellt, der neben der Länge auch die Zeichengruppen (Buchstaben, Ziffern oder Sonderzeichen) der in ihm enthaltenen Zeichen bestimmt. Für den Textblock „`MagicCasino`“ würde beispielsweise der folgende reguläre Ausdruck erstellt:

```
([A-Za-z]){11}
```

Sind alle Nachrichten ausgewertet, werden die verschiedenen regulären Ausdrücke der einzelnen Platzhalter zu einem Ausdruck verschmolzen. Dieser enthält alle Zeichengruppen und Sonderzeichen der Teilausdrücke. Die Länge des durch diesen regulären Ausdruck abgedeckten Textes wird durch den jeweils kürzesten und längsten Teilausdruck bestimmt. Ein möglicher regulärer Ausdruck für einen Platzhalter könnte beispielsweise wie folgt aussehen:

```
([\"\<\@\-\.\>\sA-Za-z]){4,13}
```

Die regulären Ausdrücke lassen immer noch Spielraum für unbekannte Textpassagen und beschreiben die Spam-Kampagne in Kombination mit den konstanten Bestandteilen ausreichend genau. Die von Pitsillidis et al. [2010] vorgestellte Erweiterung um *Dictionaries*, bei der auch ganze Wörter in den regulären Ausdruck mit aufgenommen werden, kann bei sehr wenig konstanten Teilen verhindern, dass der Filter zu

generisch wird, nimmt aber gleichzeitig variable Bestandteile der Kampagne in den Filter auf.

Nachdem alle Platzhalter der Filter-Vorlage durch einen regulären Ausdruck ersetzt worden sind, ist die Filter-Generierung abgeschlossen. Der Filter ist ein regulärer Ausdruck, der die in der Spam-Kampagne verwendete Vorlage präzise beschreibt und auf alle Nachrichten passt, die in dem beobachteten Analyselauf von dem Spambot versendet wurden.

Listing 4.1 zeigt das Ergebnis eines solchen Filter-Generierungsprozesses. Der Filter wurde aus 1.138 Spam-Nachrichten generiert, die von einem einzelnen Spambot der BOBAX-Familie abgegriffen wurden. Während der Analyse des Bots am 15.12.2008 führte dieser nur eine Spam-Kampagne aus. Alle versendeten Nachrichten können daher von nur einem Filter abgedeckt werden. Der dargestellte Filter besitzt insgesamt nur wenige variable Teile. Der meiste Text der E-Mail ist konstant, lediglich einige HTML-spezifische Elemente, das Betreff-Feld und Teile der beworbenen URL wurden durch einen regulären Ausdruck ersetzt. Selbst die Domain der beworbenen URL blieb im vorliegenden Fall in allen Nachrichten gleich.

In dem hier vorgestellten Filter werden nur das *Subject*, der *X-Mailer* und der komplette Body der E-Mail betrachtet. Alle übrigen Informationen, inklusive der Dateianhänge, werden vorher aus der E-Mail entfernt. Die meisten Header-Felder, wie das *From*- und *To*-Feld, enthalten nur variable Teile und bieten deshalb keinen Mehrwert für den Filter. Andere Felder enthalten *Sandnet*-spezifische Informationen, zum Beispiel die *Received*- und das *Date*-Feld, und sind ebenfalls nicht zum Filtern geeignet. Abbildung 4.3 verdeutlicht dies nochmals anhand der Header zweier Nachrichten, die kurz nacheinander von dem analysierten Spambot versendet wurden. Der Textteil der beiden Nachrichten war dabei nahezu identisch.

In dem beschriebenen Fall konnte die gesamte Spam-Kampagne mit nur einem Filter abgedeckt werden. Dies ist in den meisten Fällen nicht möglich. Besonders wenn Permutationen von Wörtern oder ganzen Sätzen zwischen den Spam-Nachrichten existieren, wird bei diesem Ansatz entweder ein sehr generischer Filter oder jeweils ein Filter pro Permutation erzeugt. In beiden Fällen reduziert sich die Güte des Filter-Mechanismus: Entweder kommt es zu *false positives* oder die benötigte Zeit zum Filtern einer Nachricht steigt aufgrund der vielen Filter erheblich. Der in Abschnitt 4.4 vorgestellte Filteransatz behebt dieses Problem, indem die Filter in einer kontextfreien statt regulären Sprache konstruiert werden.

```
1  Subject: ([\:\'\,\?\w]){3,11} ([\,\'\?\s\w]){14,35}
2  X-Mailer: Microsoft Outlook Express 6.00.2900.2180
3
4  Body:
5  Lose Weight -Burn Fat -Look great and feel=20
6  great -
7   
8  http://([A-Za-z]){9,14}.chat.ru
9   
10 And do all this with a miracle weight loss fruit &#8211; ACAI BERRY
       &#8211;=
11 Best=20
12 of all &#8211; it&#8217;s completely FREE for a limited time! Click here
       to=
13  receive your=20
14 completely free bottle of ACAI BERRY supplemnt!
15  
16
17 Next Body Part:
18 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
19 <HTML><HEAD>
20 <META http-equiv=3DContent-Type content=3D"text/html;
21 charset=3D([A-Za-z]){1,6}s([A-Za-z]){0,1}-([\-\=\s\w]){5,6}"([\=\s])
       {0,2}>
22 <META content=3D"MSHTML 6.00.([\d]){4,4}.([\d]){4,4}" name=3DGENERATOR>
23 <STYLE></STYLE>
24 </HEAD>
25 <BODY bgColor=3D#ffffff>
26 <DIV><FONT face=3DArial size=3D2>Lose Weight -<BR>Burn Fat -<BR>Look
       great =
27 and feel=20
28 great -</FONT></DIV>
29 <DIV><FONT face=3DArial size=3D2></FONT> </DIV>
30 <DIV><FONT face=3DArial size=3D2><A=20
31 href=3D"http://([A-Za-z]){9,14}.chat.ru">
32 http://([A-Za-z]){9,14}.chat.ru</A></F([\=\s]){0,2}ONT></DI([\=\s]){0,2}V
       >
33 <DIV><FONT face=3DArial size=3D2> </DIV>
34 <DIV><BR>And do all this with a miracle weight loss fruit &#8211; ACAI
       BERR=
35 Y &#8211;<BR>Best=20
36 of all &#8211; it&#8217;s completely FREE for a limited time! Click here
       to=
37  receive your=20
38 completely free bottle of ACAI BERRY supplemnt!</FONT></DIV>
39 <DIV><FONT face=3DArial size=3D2></FONT> </DIV></BODY></HTML>
```

Listing 4.1: Ein *regex* Filter für eine Spam-Kampagne des BOBAX-Botnetzes.

```
1  Return-path: <potyart1965@SHAEVEL.COM>
2  Envelope-to: epbnjhnvdnhmxvkdd@ucfc.com
3  Delivery-date: Fri, 16 May 2008 22:38:57 +0200
4  Received: from [10.0.2.10] (helo=hlab.informatik.uni-mannheim.de)
5          by sandnet-router with esmtp (Exim 4.67)
6          (envelope-from <potyart1965@SHAEVEL.COM>)
7          id 1Jx6hc-000663-6G
8          for epbnjhnvdnhmxvkdd@ucfc.com; Fri, 16 May 2008 22:38:56 +0200
9  User-Agent: Microsoft-Entourage/12.1.0.080305
10 Date: Fri, 16 May 2008 20:46:18 +0200
11 Subject: Funny naked girls
12 From: Pignatello <potyart1965@SHAEVEL.COM>
13 To: "epbnjhnvdnhmxvkdd@ucfc.com" <epbnjhnvdnhmxvkdd@ucfc.com>
14 Message-ID: <014A9832.9%potyart1965@SHAEVEL.COM>
15 Thread-Topic: Funny naked girls
16 Thread-Index: Aci3leRh+N8gO6I3T9ubOADIyCI6EQ==
17 [...]
```

(a)

```
1  Return-path: <Ismaell-prohtniw@SHAEVEL.COM>
2  Envelope-to: epbk@wananchi.com
3  Delivery-date: Fri, 16 May 2008 22:38:57 +0200
4  Received: from [10.0.2.10] (helo=hlab.informatik.uni-mannheim.de)
5          by sandnet-router with esmtp (Exim 4.67)
6          (envelope-from <Ismaell-prohtniw@SHAEVEL.COM>)
7          id 1Jx6hc-000665-6R
8          for epbk@wananchi.com; Fri, 16 May 2008 22:38:56 +0200
9  User-Agent: Microsoft-Entourage/12.1.0.080305
10 Date: Fri, 16 May 2008 20:46:18 +0200
11 Subject: She will want MORE of you
12 From: Ismaell <Ismaell-prohtniw@SHAEVEL.COM>
13 To: "epbk@wananchi.com" <epbk@wananchi.com>
14 Message-ID: <B9E46EC1.F%Ismaell-prohtniw@SHAEVEL.COM>
15 Thread-Topic: She will want MORE of you
16 Thread-Index: Aci3leRkwwZOrdPVT+yyu1YQ4iQ8pw==
17 [...]
```

(b)

Abbildung 4.3: Gegenüberstellung von zwei E-Mail-Headern einer Spam-Kampagne.

4.3.2 Auswertung der Filter mit regulären Ausdrücken

Zur Evaluierung des Ansatzes wurden verschiedene Spambots aus dem Datenbestand des MwAnalysis-Projekts in der in Abschnitt 4.2 beschriebenen Analyseumgebung für 30 Minuten ausgeführt [Göbel et al., 2009]. Alle während der Ausführung über SMTP verschickten Nachrichten wurden am ausgehenden Gateway auf einen lokalen Mail-Server umgeleitet und auf diesem im `mbox` Format abgespeichert. Nachdem für die `mbox`-Dateien der einzelnen Ausführungen der beschriebene Filter-Generierungsprozess durchlaufen wurde, wurden die Filter gegen ein privates Spam-Archiv getestet. Die Ergebnisse dieses Tests werden im ersten Abschnitt der Evaluierung präsentiert. In den darauf folgenden Abschnitten werden die Ergebnisse auf wesentlich umfangreichere Datensätze im Detail dargestellt. Die Datensätze aus drei

verschiedenen Botnetzen wurden von A. Pitsillidis zur Verfügung gestellt und umfassen bis zu einer Millionen Nachrichten pro Kampagne.

Analyse von Spambots des MWAnalysis-Projekts.

Aus dem MWAnalysis-Projekt wurden 40 verschiedene[1] Spambots entnommen und in der in Abschnitt 4.2 dargestellten Analyseumgebung ausgeführt. Insgesamt konnten dabei nur 100.977 E-Mail-Nachrichten abgegriffen werden. Zur Evaluierung der aus diesen Nachrichten generierten Filter wurde ein privates Spam-Archiv mit 20.290 E-Mails herangezogen. Die maximale Überdeckung lag bei 30 Prozent, das heißt, fast ein Drittel aller in dem Archiv enthaltenen Nachrichten war Teil einer von den analysierten Bots verschickten Kampagne. Zum Erreichen dieser Überdeckung mussten die generierten Filter jedoch händisch angepasst werden. Dabei wurde die Genauigkeit der Filter nachträglich herabgestuft, indem beispielsweise zusätzliche Zeichengruppen oder eine höher Anzahl an Zeichen in den variablen Bestandteilen der regulären Ausdrücke zugelassen wurden. Diese Nachbearbeitung war aufgrund der geringen Varianz innerhalb der gesammelten Nachrichten notwendig. Das folgende Beispiel veranschaulicht das Problem:

Für eine ein Spielcasino bewerbende Spam-Kampagne wurden während der Analyse des Spambots 71 E-Mail-Nachrichten abgefangen. Aus dem Spam-Archiv gehörten 493 E-Mail-Nachrichten zu dieser Kampagne. Der aus den 71 Nachrichten generierte Filter passte ohne manuelle Veränderungen jedoch nur auf 26 Nachrichten. Dies entspricht einer Erkennungsrate von lediglich 5 Prozent. Der Filter ist somit als zu genau anzusehen, was letztlich in den zur Generierung des Filters herangezogenen Nachrichten begründet ist. Durch die Hinzunahme einer E-Mail-Nachricht aus dem Spam-Archiv lässt sich die Erkennungsquote auf 26 Prozent steigern. Die Erweiterung des Lerndatensatzes um zwei Nachrichten führt zu einem nahezu perfekten Filter mit einer Erkennungsrate von 99 Prozent. Die Güte der regulären Filter basiert damit sehr stark auf der Varianz innerhalb der zur Generierung herangezogenen Nachrichten.

Analyse von 437 Spam-Kampagnen des STORM-Botnetzes.

Die 437 Spam-Kampagnen des STORM-Botnetzes umfassen jeweils 5.000 Nachrichten und wurden von Pitsillidis et al. [2010] zur Verfügung gestellt. Für jede dieser Kampagnen wurden fünf Filter generiert. Diese basieren auf jeweils einer, fünf, fünfzig und fünfhundert zufällig ausgewählten Spam-Nachrichten. Zusätzlich wurden auch Filter für alle 5.000 Nachrichten berechnet, um zu verifizieren, dass alle Nachrichten auch zur selben Kampagne gehören. Alle Filter wurden anschließend auf die je 5.000 Nachrichten entsprechenden Kampagne angewendet, um die Erkennungsraten zu bestimmen. Die für die einzelnen Stufen über alle Kampagnen erreichten minimalen, maximalen und durchschnittlichen Erkennungsraten der Filter sind in Tabelle 4.1 dargestellt. Eine vollständige Auflistung aller Filterergebnisse findet sich in Anhang D.

Aus den einzelnen Werten lässt sich die Abhängigkeit der Erkennungsraten von der Anzahl zur Filtergenerierung verwendeter Nachrichten gut ablesen. Im Durchschnitt

[1] In diesem Zusammenhang wurde der SHA1-Fingerabdruck als Unterscheidungsmerkmal verwendet, Familienzugehörigkeit wurde nicht untersucht beziehungsweise beachtet.

Lerndatensatz		**Erkennungsrate**		
# E-Mails	*in %*	*minimal*	*maximal*	*durchschnittlich*
1	0,02	0.06 %	2,34 %	1,0567 %
5	0,1	3,54 %	83,62 %	32,5692 %
50	1,0	77,86 %	100,00 %	92,5036 %
500	10	98,52 %	100,00 %	99,9936 %
5.000	100	100,00 %	100,00 %	100,0000 %

Tabelle 4.1: Erkennungsraten der auf regulären Ausdrücken basierten Spam-Filter für den STORM-Spamdatensatz.

werden 50 E-Mail-Nachrichten benötigt, um eine akzeptable Erkennungsrate von 92 Prozent zu erreichen. Das heißt, nachdem lediglich ein Prozent der Nachrichten verarbeitet wurde, steht für die meisten Kampagnen bereits ein guter Filter zur Verfügung. Bei 13 Kampagnen konnten mit einem auf 50 E-Mail-Nachrichten erstellten Filter sogar Erkennungsraten von 100 Prozent erreicht werden. Dieser perfekte Filter steht für fast alle Kampagnen nach der Auswertung von 10 Prozent der gesendeten Nachrichten zur Verfügung. Nur für zwei Kampagnen wurden auf dieser Stufe mit 98,54 Prozent und 99,65 Prozent leicht niedrigere Erkennungsraten erreicht. Der Ansatz liefert damit auf den ausgewerteten Datensätzen relativ schnell sehr gute Filterergebnisse. Mit 10 Prozent werden zwar relativ viele Nachrichten zur Filtergenerierung herangezogen, aber da für jede Kampagne nur ein Filter erzeugt wurde, ist davon auszugehen, dass die aus 500 Nachrichten generierten Filter die gesamte Kampagne, also weit mehr als die hier herangezogenen 5.000 Nachrichten, abdecken werden. Damit reduziert sich das Verhältnis von gelernten und gefilterten Nachrichten erheblich.

Analyse eines Spam-Laufs des GHEG-Botnetzes.

Der Spam-Datensatz des GHEG-Botnetzes besteht aus 254.592 E-Mails. Aus diesem Datensatz wurden zufällig 3, 26, 256, 2.556 und 25.459 Nachrichten entnommen und zur Filtergenerierung verwendet. Das Erstellen eines Filters aus 100 Prozent der Nachrichten war mit der aktuellen Implementierung des Prototypen nicht möglich, da der Lernprozess auf so großen Datensätzen nicht skaliert. Der Algorithmus berechnet für jeden der fünf Trainingsdatensätze genau einen Filter. Die Erkennungsquoten der einzelnen Filter sind in Tabelle 4.2 aufgetragen.

Lerndatensatz		**Filterergebnisse**	
# E-Mails	*in %*	*Erkennungsrate*	*passender E-Mails*
3	0,001	0,3366 %	857
26	0,01	93,1785 %	237.225
256	0,1	99,9996 %	254.591
2.556	1,0	99,9996 %	254.591
25.459	10,0	99,9996 %	254.591

Tabelle 4.2: Erkennungsraten der auf regulären Ausdrücken basierenden Spam-Filter auf dem GHEG-Spamdatensatz. Die Erkennungsrate ist jeweils in Abhängigkeit von der zur Filtergenerierung herangezogenen Anzahl an E-Mail-Nachrichten angegeben.

Mit 256 E-Mail-Nachrichten werden nur 0,1 Prozent der in dem Analyselauf aufgezeichneten Nachrichten benötigt, um einen fast perfekten Filter für die Kampagne zu generieren. Bis auf eine Nachricht der Spam-Kampagne werden alle Nachrichten von diesem Filter abgedeckt. Auch die auf einem Prozent und zehn Prozent der Nachrichten generierten Filter erreichen hier kein besseres Ergebnis. Die händische Analyse der in allen drei Fällen nicht erkannten Nachricht ergab, dass dieser Nachricht drei Zeilen fehlen, die in allen anderen Nachrichten der Kampagne enthaltenen sind. Die fehlende Zeilen enthalten sowohl Text, als auch HTML-Kodierungsinformationen. Durch die fehlenden Kodierungsinformationen wird die Nachricht nicht korrekt dargestellt. Es ist daher unwahrscheinlich, dass es sich dabei um eine gewollte Variation der Nachricht handelt. Vielmehr scheinen hier nachträglich Informationen verloren gegangen zu sein.

Insgesamt zeigt die Analyse des GHEG-Spam-Datensatzes, dass der vorgestellte Ansatz sehr gute Ergebnisse auf musterbasierten Spam-Kampagnen liefert und im vorliegenden Fall mit 0,1 Prozent der Nachrichten nur einen Bruchteil der E-Mail-Nachrichten benötigt, um einen auf die gesamte Kampagne passenden Filter zu erstellen.

Analyse eines Spam-Laufs des SRIZBI-Botnetzes.

Der analysierte Spam-Datensatz des SRIZBI-Botnetzes besteht aus 1.000.000 E-Mail-Nachrichten. Analog zu der Analyse auf dem GHEG-Spam-Datensatz wurden zufällig 0,001 Prozent, 0,01 Prozent, 0,1 Prozent, ein Prozent und zehn Prozent der Nachrichten zur Filtergenerierung entnommen. Abhängig von der Anzahl verwendeter E-Mail-Nachrichten werden für den SRIZBI-Datensatz allerdings zwischen 5 und 11 Filter generiert. Die Filter und die jeweiligen Erkennungsquoten sind in Tabelle 4.3 aufgetragen.

	Anzahl passender Nachrichten je Filter				
	0,001 %	*0,01 %*	*0,1 %*	*1,0 %*	*10 %*
Filter 1	261	275.597	325.481	334.809	335.649
Filter 2	1	1	3.655	14.299	15.010
Filter 3	2	110.069	227.252	235.242	235.585
Filter 4	1	16.121	105.621	113.247	113.754
Filter 5	1	87.163	172.883	177.954	178.078
Filter 6	-	1.041	12.923	19.737	20.285
Filter 7	-	3	21.748	29.837	30.932
Filter 8	-	2	2.990	9.247	10.045
Filter 9	-	2	28.478	33.786	34.947
Filter 10	-	2	7.550	16.965	17.390
Filter 11	-	1	1.014	6.718	7.589
Σ*Filter* i	266	490.002	908.595	991.841	999.264

Tabelle 4.3: Die Anzahl der von den einzelnen Filtern auf dem SRIZBI-Spamdatensatz erkannten Nachrichten.

Die Zuordnung der auf den einzelnen Stufen generierten Filter zueinander erfolgte händisch. Da die für die Filtergenerierung herangezogenen E-Mail-Nachrichten zufällig aus dem Datensatz entnommen wurden, ist eine automatische Zuordnung nicht möglich. Als Entscheidungskriterium wurden daher die von den Filtern abgedeckten E-Mail-Nachrichten herangezogen. Dies ist möglich, da die Filter nur disjunkte Teilmengen der Nachrichten abdecken können. Damit wird jeder auf 0,001 Prozent der Nachrichten generierte Filter zwingend gegen die ihm zugeordneten Filter höherer Stufe konvergieren.

Auf zehn zufällig aus dem Spam-Lauf entnommenen E-Mail-Nachrichten generiert der Algorithmus fünf Filter, von denen allerdings nur zwei auf mehr als einer Nachricht basieren. *Filter 1* wurde aus fünf Nachrichten erzeugt und besitzt damit genug Varianz, um beim Filtern des gesamten Datensatzes auch auf andere Nachrichten der Kampagne zu passen. *Filter 3* wurde zwar aus zwei Nachrichten erstellt, ist aber immer noch zu spezifisch, um auf weitere Nachrichten zu passen. Die auf einer Nachricht basierenden Filter besitzen keinerlei Varianz und passen damit nur auf die E-Mail-Nachricht, aus der sie erstellt wurden. Insgesamt können mit den fünf Filtern lediglich 0,027 Prozent der E-Mail-Nachrichten einer der fünf erkannten Kampagnen zugeordnet werden.

Werden hundert zufällig ausgewählte E-Mail-Nachrichten zur Filtergenerierung herangezogen, wird bereits für jede in dem Datensatz enthaltene Kampagne ein Filter erzeugt. Insgesamt werden 49 Prozent aller Nachrichten erfolgreich einem dieser Filter zugeordnet. Die durchschnittliche Erkennungsrate für die Filter liegt bei rund 18 Prozent. Da keine sichere Aufteilung aller E-Mail-Nachrichten des Spam-Laufs in die einzelnen Kampagnen vorliegt, wurden die Filterergebnisse auf den einzelnen Stufen jeweils mit den Filterergebnissen des auf 10 Prozent der Nachrichten erstellten Filters ins Verhältnis gesetzt, um zumindest eine Abschätzung der Erkennungsraten liefern zu können. Insgesamt liegen die Erkennungsraten für die auf 0,01 Prozent der Nachrichten erzeugten Filter weit unter der, die für das GHEG-Botnetz berechnet wurde. Dieser Unterschied kann darauf zurückgeführt werden, dass das SRIZBI-Botnetz parallel mehrere Kampagnen verarbeitet und sich die zur Filtergenerierung herangezogenen E-Mail-Nachrichten auf mehrere Kampagnen und Filter verteilen. Da die Auswahl der Nachrichten zufällig erfolgt, kommt es zu großen Schwankungen bei den Erkennungsraten der einzelnen Filter. Sechs der elf Filter basieren auf weniger als vier Nachrichten und sind damit sehr spezifisch. Die Filter passen deshalb nur auf die Nachrichten, aus denen sie generiert wurden und besitzen Erkennungsquoten von weniger als 0,02 Prozent. Für die übrigen fünf Filter wurden zwischen 7 (*Filter 6*) und 36 (*Filter 1*) Nachrichten herangezogen. *Filter 1* wurde auf 0,01 Prozent der der Kampagne zugeordneten Nachrichten erzeugt und erreicht mit 82,11 Prozent eine gute Erkennungsrate. Die vollständige Auflistung aller Erkennungsraten liefert Tabelle 4.4.

Für die auf 0,1 Prozent, einem Prozent und zehn Prozent der E-Mail-Nachrichten erstellten Filter werden die Filterergebnisse und Erkennungsraten stetig besser. Aus 0,1 Prozent, also 1.000 Nachrichten, werden elf Filter generiert, mit denen sich 90,86 Prozent aller E-Mail-Nachrichten den einzelnen Kampagnen zuordnen lassen. Die Erkennungsraten der einzelnen Filter liegen hier zwischen 13,36 Prozent und 97,08 Prozent. Die aus einem Prozent der Nachrichten generierten Filter erkennen bereits über 99 Prozent der Nachrichten und liefern Erkennungsraten von durchschnittlich 96,63

	Erkennungsraten			
	0,001 %	*0,01 %*	*0,1 %*	*1,0 %*
Filter 1	0,08 %	82,11 %	96,97 %	99,75 %
Filter 2	0,01 %	0,01 %	24,35 %	95,26 %
Filter 3	0,01 %	48,72 %	96,46 %	99,85 %
Filter 4	0,01 %	14,17 %	92,85 %	99,55 %
Filter 5	0,01 %	48,95 %	97,08 %	99,93 %
Filter 6	-	5,13 %	63,71 %	97,30 %
Filter 7	-	0,01 %	70,31 %	96,46 %
Filter 8	-	0,02 %	29,77 %	92,06 %
Filter 9	-	0,01 %	81,49 %	96,68 %
Filter 10	-	0,01 %	43,42 %	97,56 %
Filter 11	-	0,01 %	13,36 %	88,52 %

Tabelle 4.4: Erkennungsraten der aus dem SRIZBI-Spamdatensatz generierten Filter. Als Bezugsgröße wurden dabei die Filterergbnisse der aus 10 Prozent der Nachrichten erstellten Filter herangezogen.

Prozent. Werden 100.000 Nachrichten zur Filtergenerierung herangezogen, so steigt das Filterergebnis auf 99,93 Prozent.

Die aus Göbel et al. [2009] entnommene Evaluierung konnte lediglich das Potential des Filteransatzes aufzeigen. Die darauf basierenden Schlussfolgerungen konnten an dieser Stelle, durch die drei zusätzlichen Analysen auf großen Datensätzen, bestätigt werden. Die Evaluierung zeigt, dass sich der Ansatz auch auf reale Daten anwenden lässt. Auf den dazu herangezogenen Datensätzen aus über 400 Spam-Kampagnen des STORM-Botnetz sowie den beiden umfangreichen Spam-Kampagnen der GHEG- und SRIZBI-Botnetze, wurden durchweg sehr gute Filterergebnisse erreicht. Der Spam-Lauf des SRIZBI-Botnetze, bei dem mehrere Kampagnen parallel verarbeitet werden müssen, zeigt zudem, dass der Ansatz die einzelnen Kampagnen gut isoliert und relativ schnell zu sehr guten Filtern für die einzelnen Kampagnen konvergiert.

4.4 Kontextfreie Grammatiken für E-Mail-Nachrichten

Die auf regulären Ausdrücken aufbauenden Spam-Filter stoßen relativ schnell an ihre Grenzen. Das gesuchte Spam-Template wird hier durch einen regulären Ausdruck abgebildet, in dem keine Oder-Verknüpfungen zwischen Teilausdrücken erlaubt sind. Das bedeutet, dass schon das Vertauschen von zwei aufeinanderfolgenden Wörtern in einer der untersuchten Nachrichten entweder den Verlust beider Worte in der Filter-Vorlage, oder die Generierung einer zusätzlichen Filter-Vorlage zur Folge hat. Dieses Problem wird bei Permutationen zwischen ganzen Sätzen noch größer. Obwohl sich der Inhalt“ der Nachrichten nicht unterscheidet, verliert die generierte Filter-Vorlage an Genauigkeit. Abbildung 4.4 veranschaulicht das Problem an einem Beispiel. In den Abbildungen 4.4 (a) und (b) sind Auszüge aus zwei Spam-Nachrichten einer Kampagne dargestellt. Gegenüber der ersten Nachricht sind in der zweiten die Zeilen 6 und 7 vertauscht und ist Zeile 11 um zwei Zeilen nach oben verschoben. Bei Anwendung des im vorherigen Abschnitt vorgestellten Filter-Generators wird für diese beiden Nachrichten der in Abbildung 4.4(c) dargestellte Filter erzeugt.

Durch die Permutationen innerhalb der Aufzählung gehen in dem Filter die Informationen eines Eintrags vollständig verloren. Um dies zu verhindern, müsste man entweder einen zusätzlichen Filter erzeugen, oder Oder-Verknüpfungen innerhalb der regulären Ausdrücke zulassen. Abbildung 4.4(d) zeigt einen solchen regulären Ausdruck für das vorliegende Beispiel. Eine entsprechende Erweiterung des Filter-Ansatzes ist möglich und wurde von Pitsillidis et al. [2010] umgesetzt. Allerdings beschränken sich Pitsillidis et al. darauf, einzelne Worte, die in den Nachrichten ersetzt wurden, über Oder-Verknüpfungen abzubilden. Das größte Problem bei der Erweiterung der Filter um diese Verknüpfungen besteht darin, automatisiert zu entscheiden, für welche Worte, Wortreihen oder auch Sätze ein entsprechendes *Subpattern* aufgebaut werden soll. Jedes zusätzliche Subpattern birgt die Gefahr, dass unnötiges, weil variables Wissen über die Kampagne gelernt wird. Dadurch wird der resultierende Filter sehr spezifisch und unnötig komplex, was sich letztlich negativ auf die Laufzeit des Filterns auswirkt.

Der in diesem Abschnitt vorgestellte Filteransatz versucht, dem Problem von Permutationen durch den Einsatz einer mächtigeren Sprache zu begegnen. Dazu wird im ersten Schritt eine kontextfreie Grammatik aufgestellt, die den allgemeinen Aufbau von E-Mail-Nachrichten beschreibt. Anschließend wird für die Nachrichten einer Spam-Kampagne gelernt, aus welchen Wörtern und Sätzen die Nachrichten bestehen und welche E-Mail-Adressen, Links und Dateianhänge in diesen vorkommen. Der resultierende Filter (*Parser*) kann anschließend genutzt werden, um beliebige Nachrichten auf das Vorkommen der gelernten Informationen hin zu überprüfen.

Im Gegensatz zu dem in Abschnitt 4.3 vorgestellten Filter-Ansatz, werden hier nicht die Spam-Vorlagen der Kampagne gelernt, sondern der für den menschlichen Betrachter sichtbare Inhalt der Nachrichten. Das heißt, der Filter enthält letztlich nur die in den Nachrichten enthaltenen Wörter, E-Mail-Adresse, URLs und Dateianhänge, sowie die sich aus diesen zusammensetzenden Sätze. Die beispielsweise bei HTML-codierten Nachrichten verwendeten *Tags* zur Syntaxbeschreibung oder vorhandene HTML-Kommentare fließen, ebenso wie die Zeilenumbrüche im Text, nicht in den Filter ein. Dieser freiwilligen Beschränkung liegt die Annahme zugrunde, dass jede versendete Spam-Nachricht einer Kampagne dem Adressaten dieselben Informatio-

```
[...]
100% kostenlos anmelden und
    Profil anlegen unter: .../
    kontakte-germany/

Hier die Fakten:
- 1:1 Textchat
- WebCam
- private Nachrichten versenden
- Fotogalerien
- Videogalerien
- Profil der Woche
- u.v.m
[...]
```

(a)

```
[...]
100% kostenlos anmelden und
    Profil anlegen unter: .../
    kontakte-germany/

Hier die Fakten:
- WebCam
- 1:1 Textchat
- private Nachrichten versenden
- Profil der Woche
- Fotogalerien
- Videogalerien
- u.v.m.
[...]
```

(b)

```
[...]
100\%\ kostenlos\ anmelden\ und\ Profil\ anlegen\ unter\:\ \.\.\.\/
    kontakte\-germany\/\
\
Hier\ die\ Fakten\:\
\-\ ([\-\sA-Za-z]){0,14}1\:1\ Textchat([\-\sA-Za-z]){0,9}\
\-\ private\ Nachrichten\ versenden\
\-\ ([\-\sA-Za-z]){0,19}Fotogalerien\
\-\ Videogalerien([\-\sA-Za-z]){0,19}\
\-\ u\.v\.m\.\
[...]
```

(c)

```
[...]
100\%\ kostenlos\ anmelden\ und\ Profil\ anlegen\ unter\:\ \.\.\.\/
    kontakte\-germany\/\
\
Hier\ die\ Fakten\:\
\-\ (WebCam\n\-\ 1\:1\ Textchat|1\:1\ Textchat\n\-\ WebCam)\
\-\ private\ Nachrichten\ versenden\
\-\ (Profil\ der\ Woche\n\-\ Fotogalerien\n\-\ Videogalerien|
    Fotogalerien\n\-\ Videogalerien\n\-\ Profil\ der\ Woche)\
\-\ u\.v\.m\.\
[...]
```

(d)

Abbildung 4.4: Veranschaulichung der Probleme von auf regulären Ausdrücken basierenden Filtern bei Permutationen. (a) und (b) zeigen die relevanten Auszüge aus den beiden zugrundeliegenden Nachrichten. Der passende, auf regulären Ausdrücken basierende Filter wird in (c) dargestellt und (d) zeigt ein um *Dictionaries* erweiterten Filter.

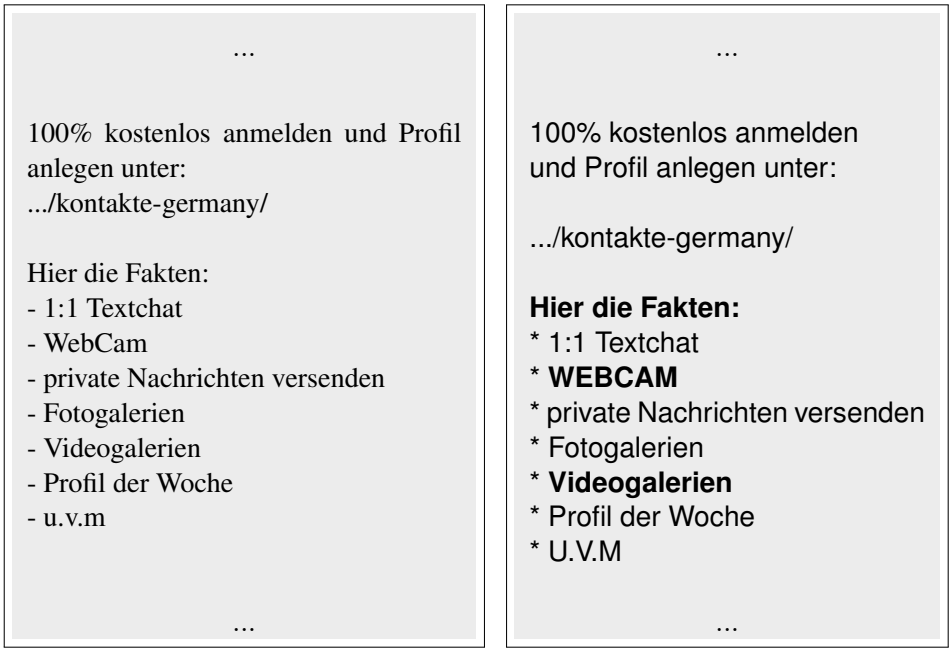


Abbildung 4.5: Inhaltlich identische Spam-Nachrichten mit unterschiedlicher Formatierung.

nen liefern muss. Da die Spam-Nachricht mit dem Ziel verschickt wird, die Aufmerksamkeit des Empfängers zu wecken, müssen die darin enthaltenen Sätze auch einen Sinn ergeben, der Inhalt beschränkt sich also eher auf einige wenige Formulierungen. Wie diese Informationen, oder die für den Benutzer ohnehin nicht sichtbare HTML-Kommentare innerhalb der Nachricht dargestellt werden, ist irrelevant.

Abbildung 4.5 verdeutlicht diesen Zusammenhang an zwei Versionen einer fiktiven Nachricht. Der Nachrichteninhalt ist in beiden Versionen identisch. Trotzdem sind für die Darstellung der in Abbildung 4.5(b) gezeigten Nachricht eine Vielzahl an HMTL-Befehlen notwendig. Ein Filter-Ansatz, der diese, allein die Formatierung betreffenden, Informationen lernt – dies ist bei allen bisher existierenden und auf regulären Ausdrücken basierenden Ansätzen der Fall – kann schon durch das Ändern der Formatierung (der HTML-Tags) ausgehebelt werden.

4.4.1 Grundgrammatik für E-Mail-Nachrichten

Die Grundgrammatik für E-Mail-Nachrichten besitzt die in Listing 4.2 dargestellten fünf Token. Zusammen mit den in Listing 4.3 dargestellten Grammatikregeln lässt sich jede beliebige E-Mail-Nachricht auf diese Token reduzieren.

```
NEWMAIL     = r'\bfrom\ \b.*\n'

HEADER      = r'([\w-]+:\ .+\n((\t|\ )+.*\n)*)
               |([\w-]+:\t.+\n((\t|\ )+.*\n)*)
               |([\w-]+:\n)'

NEWLINE     = r'\n'

WORD        = r'[\w]+'

PUNCTUATION = r'[\!\"\#\$\%\&\'\(\)\*\+\,\-\.\/\:\;\=\<\>\?\@\[\\\]
                 \^\_\`\{\|\}\~]'
```

Listing 4.2: Die Token der Grundgrammatik für E-Mail-Nachrichten.

Der NEWMAIL-Token beschreibt die optionale Absender-Zeile vor dem eigentlichen Header. Diese wird lediglich von dem empfangenden SMTP-Server ausgewertet und ist für den Empfänger der Nachricht nicht sichtbar.

Ein HEADER-Token passt auf jedes beliebige Feld des Headers, inklusive dem das Feld abschließenden Zeilenumbruch. Auch leere oder sich über mehrere Zeilen erstreckende Felder, wie zum Beispiel die *Received*-Felder oder das *Content-Type*-Feld, werden von genau einem Token abgedeckt. Um auch den Inhalt bestimmter Header-Felder zu lernen, sind zusätzliche, speziellere Token notwendig, auf die später eingegangen wird.

Ein NEWLINE-Token stellt einen einzelnen Zeilenumbruch dar. Dieser einfache Token ist notwendig, um den Wechsel zwischen dem E-Mail-Header und dem -Body zu erkennen und spezielle Grammatikregeln und Token, wie zum Beispiel die passende IGNORE-Regel, zu aktivieren. Der WORD-Token passt auf jede beliebige Zeichenfolge.

Alle Satzzeichen werden durch den PUNCTUATION-Token dargestellt. Da beim Parsen der Nachrichten die enthaltenen Sätze im Vordergrund stehen, müssen die Satzzeichen, die diese voneinander trennen, bewusst verarbeitet werden.

Zusätzlich zu den fünf Token exitieren noch zwei IGNORE-Token, die beschreiben, welche Zeichen innerhalb des Headers beziehungsweise innerhalb des Bodies ignoriert werden sollen.

```
ignore_header   = r' \t'
ignore_body     = r' \t\n'
```

Innerhalb des Headers werden dabei nur Leerzeichen und *Tabs* ignoriert. Die innerhalb des Bodies zusätzlich ignorierten Zeilenumbrüche müssen weiterhin beachtet werden, da sie die einzelnen Header-Felder voneinander trennen. Im Body können ein-

zelne Zeilenumbrüche hingegen ignoriert werden, da sie keinerlei Bedeutung für die enthaltenen Sätze besitzen.

```
spammail        : NEWMAIL header NEWLINE body
                | header NEWLINE body

header          : header_fields

header_fields   : HEADER header_fields
                | HEADER
                | empty

body            : sentences

sentences       : sentences sentences
                | sentence sentences
                | sentence PUNCTUATION

sentence        : words

words           : WORD words
                | WORD
                | empty

empty           :
```

Listing 4.3: Die Ableitungsregeln der Grundgrammatik für E-Mail-Nachrichten.

Eine E-Mail-Nachricht besteht demnach aus einem optionalen NEWMAIL-Token, einem Header (`header`-Symbole), dem trennenden NEWLINE-Token und dem Body (`body`-Symbol). Der Header setzt sich aus einzelnen Feldern (`header_fields`) zusammen, die sich jeweils auf einzelne HEADER-Token reduzieren lassen. Der Body besteht dagegen aus Sätzen (`sentences`), die jeweils aus einem Satz (`sentence`) und einem PUNCTUATION-Token bestehen. Ein Satz besteht aus einer Aneinanderreihung von Wörtern (`words`), wobei jedes Wort leer sein kann (`empty`) oder einen WORD-Token darstellt.

Diese einfache Grammatik genügt, um jede E-Mail-Nachricht abzuleiten. Allerdings ist sie zum Lernen von Inhalten noch zu unpräzise. Alle Felder des Headers werden durch einen Token repräsentiert. Dieser macht keinerlei Unterschied zwischen den Bezeichnern und dem Wert des Feldes. Mögliche Dateianhänge – diese werden als *base64*-codierter Text in den Body der Nachricht integriert – werden als Wörter gelernt und E-Mail-Adressen und URLs werden an den „Satzzeichen“ zerlegt und ebenfalls wortweise gelernt. Um diese Fehlinterpretationen zu unterbinden und auch Informationen aus einem Header-Feld extrahieren und lernen zu können, sind zusätzliche Token und Grammatikregeln notwendig.

4.4.2 Erweiterte Grundgrammatik für E-Mail-Nachrichten.

Aus dem Header sind (momentan) lediglich die Felder *Subject* und *Content-Type* von Interesse [Göbel et al., 2009]. Zur Verarbeitung der beiden Felder wurde die Grammatik um die in Listing 4.4 dargestellten Token und Regeln erweitert:

```
*** TOKEN ***

SUBJECT = r'(\bsubject\b: )
           |(\bsubject\b:\t)
           |(\bsubject\b:)'

CONTENT_TYPE = r'content-type:(((\t|\ )+.*\n((\t|\ )+.*\n)*)|(\n))'

*** RULES ***

header  : header_fields subject header_fields

subject : SUBJECT sentences NEWLINE
        | SUBJECT sentence NEWLINE'''
```

Listing 4.4: Erste Anpassung der Grundgrammatik für E-Mail-Nachrichten.

Das SUBJECT-Token passt dabei ausschließlich auf den Feldbezeichner. Der Wert des Feldes wird über die zusätzliche Grammatikregel als Satz oder Sätze, gefolgt von einem NEWLINE-Token, definiert. Der Wert des *Subject*-Felds wird damit genauso behandelt wie jeder beliebige Satz aus dem Nachrichten Body. Da sich das *Subject*-Feld und Sätze des Bodies gerade bei Spam-Nachrichten vielfach gegeneinander austauschen lassen, macht eine gesonderte Betrachtung des *Subject*-Feldes wenig Sinn. Das neue `subject`-Symbol wird durch die veränderte Regel zum Ableiten des `header`-Symbols in die Grammatik integriert.

Der *Content-Type* ist das zweite interessante Header-Feld. Es wird durch den dargestellten Token erkannt und anschließend ausgewertet, um ein eventuell enthaltenes *Boundary*-Attribut auszulesen und zu überprüfen, ob die E-Mail Nachricht HTML-codiert ist. Anhand der *Boundary* lassen sich die einzelnen Teile der Nachrichten erkennen und voneinander trennen. Wurde ein entsprechender Token gefunden, wird der Nachrichten-Body beim Übergang des Lexer vom Header zum Body wie folgt optimiert: Aus den HTML-codierten Blöcken werden alle HTML-Tags entfernt, wobei darin enthaltenen URLs erhalten bleiben. Des Weiteren werden *base64*-codierte Blöcke durch ihren MD5-Hashwert ersetzt. Dadurch wird zum einen verhindert, dass unnötige Wörter aus dem „Multi-Part Header“ oder Dateianhängen gelernt werden und zum anderen können die in der Nachricht enthaltenen Dateianhänge selbst gelernt werden. Zusätzlich werden die „Multi-Part Header“ innerhalb des Nachrichten-Body, ihrem Typ entsprechend, durch eine der folgenden drei Zeichenketten ersetzt:

```
PT_ATTACHMENT_MD5_{md5_of_bas64_encoded_text}_MD5_PT
PT_MULTIPART_HTML_PT
PT_MULTIPART_PT
```

Um die neuen Zeichenketten korrekt ableiten zu können, sind zwei zusätzliche Token notwendig. Die beiden MULTIPART-Zeichenketten werden lediglich zu einem Token abgeleitet. Bei der Ableitung der ATTACHMENT-Zeichenkette wird zusätz-

lich der MD5-Hashwert ausgelesen und als neuer Dateianhang gelernt, siehe dazu Abschnitt 4.5.

Neben dem Dateianhang kommt den im Nachrichten-Body eventuell enthaltenen E-Mail-Adressen und URLs eine besondere Rolle beim Filtern der Nachrichten zu. Um diese erkennen zu können, wird die Grammatik um die in Listing 4.5 dargestellten Token zum Erkennen von E-Mail-Adressen und URLs erweitert:

```
URL   = r'((https?|ftp):((\/\/)|(\\\\))+[\d\w:@\/()~_?\+\-=\\\.&]*)
                                      |(www\.[\d\w:@\/()~_?\+\-=\\\.&]*)'
E-Mail = r'[\d\w\-\+\.\_]+@[\d\w\-\+\.\_]+\.[\w]{2,3}'
```

Listing 4.5: Zweite Anpassung der Grundgrammatik für E-Mail-Nachrichten.

Analog zu den Dateianhängen, werden auch die in den Nachrichten enthaltenen URLs und E-Mail-Adressen während des Parsens gelernt. Details dazu werden in Abschnitt 4.5 präsentiert.

Die erweiterte Grundgrammatik besitzt neben den hier vorgestellten Erweiterungen noch zusätzliche Token und Regeln, zum Beispiel um Zahlen erkennen zu können, auf die an dieser Stelle nicht näher eingegangen wird. Insgesamt besteht die erweiterte Grammatik aus 14 Token und 25 Grammatikregeln, die in Tabelle 4.5 und Listing 4.6 dargestellt sind.

Position	Token	Zustände
1	NEWMAIL	INITIAL
2	SUBJECT	INITIAL
3	CONTENT_TYPE	INITIAL
4	HEADER	INITIAL
5	NEWLINE	INITIAL, SSUBJECT
6	PUNCTUATION	SSUBJECT, BBODY
7	MULTIPART_INFO	BBODY
8	MULTIPART	BBODY
9	URL	SSUBJECT, BBODY
10	E-Mail	SSUBJECT, BBODY
11	ATTACHMENT	BBODY
12	INTEGER	SSUBJECT, BBODY
13	UNIT	SSUBJECT, BBODY
14	WORD	SSUBJECT, BBODY

Tabelle 4.5: Token der erweiterten Grundgrammatik. Der MULTIPART_INFO-Token passen auf den Satz „this is a multi-part message in mime format.“ und verhindert so, dass unnötige Worte gelernt werden.

Aufgrund der Überschneidungen zwischen einzelnen Token, ein HEADER-Token könnte auch in WORD und PUNCTUATION-Token aufgeteilt werden, müssen die Token dem Lexer in der gewünschten Verarbeitungsreihenfolge übergeben werden. In Tabelle 4.5 sind die Token bereits entsprechend sortiert.

Um beim Ableiten selektiv Token zuzulassen, wird der Lexer in Abhängigkeit von den bereits gelesenen Token in unterschiedliche Zustände versetzt. Insgesamt kann der Lexer in drei Zuständen arbeiten, wobei jeweils die in Tabelle 4.5 angegebenen Token aktiv sind [2]:

INITIAL Initial befindet sich der Lexer im Zustand INITIAL. In diesem Zustand sind nur die vier den E-Mail-Header lesenden Token NEWMAIL, SUBJECT, CONTENT_TYPE und NEWLINE gültig. Dadurch ist es unmöglich, dass Teile eines Header-Feldes als WORD oder, wie im Fall des *From*- und *To*-Feldes durchaus möglich, EMAIL-Token abgeleitet werden.

SSUBJECT Sobald der SUBJECT-Token gelesen wurde, wird der Lexer in den Zustand SSUBJECT versetzt. In diesem sind die TOKEN NEWLINE, PUNCTUATION, URL, EMAIL, INTEGER, UNIT und WORD gültig und ermöglichen es dem Lexer, beliebigen Text inklusive mehrerer Zeilenumbrüche abzuleiten. Nach Abschluss des *Subject*-Felds, das heißt, nachdem der finale Zeilenumbruch gelesen wurde, wechselt der Lexer wieder in den Zustand INITIAL.

BBODY Nachdem der Header der E-Mail-Nachricht vollständig abgearbeitet wurde, wird ein einzelner NEWLINE Token gelesen. Daraufhin wird der Lexer in den Zustand BBODY gesetzt und besitzt mit PUNCTUATION, MULTIPART_INFO, MULTIPART, URL, EMAIL, ATTACHMENT, INTEGER, UNIT und WORD neun gültige Token, mit denen der Inhalt der Nachricht vollständig abgeleitet werden kann.

Listing 4.6 zeigt nochmals alle Regeln der erweiterten Grundgrammatik. Die zusätzlichen Regeln erweitern die Grundgrammatik nur um die zum Lernen notwendigen Details, wie EMAIL- und MULTIPART-Token, und bedürfen daher keiner weiteren Erläuterung.

[2] In der konkreten Umsetzung existiert noch ein vierter Zustand in den bei der Bearbeitung des *Subject*-Feldes gewechselt wird, um den letzten Zeilenumbruch zuverlässig erkennen zu können.

```
spammail        : NEWMAIL header NEWLINE body
                | header NEWLINE body

header          : header_fields subject header_fields

header_fields   : HEADER header_fields
                | HEADER
                | CONTENT_TYPE
                | empty

subject         : SUBJECT sentences NEWLINE
                | SUBJECT sentence NEWLINE
                | SUBJECT sentences sentence NEWLINE

body            : MULTIPART_INFO sentences
                | sentences

sentences       : sentences sentences
                | sentence delimiter
                | sentence email
                | sentence url
                | email
                | url
                | attachment
                | MULTIPART

sentence        : words

words           : word words
                | word
                | empty

word            : WORD
                | INTEGER
                | UNIT

delimiter       : PUNCTUATION

email           : E-Mail

url             : URL

attachment      : ATTACHMENT

empty           :
```

Listing 4.6: Die Regeln der erweiterten Grundgrammatik.

4.5 Lernen von Spam-Vorlagen mit LEX und YACC

Der Prototyp zum Parsen, Lernen und anschließenden Filtern von E-Mail-Nachrichten ist in PYTHON implementiert und verwendet die von dem Module PLY [Beazley, 2009] bereitgestellte LEX/YACC-Implementierung. Analog zum Aufbau des Filters für MIST, existieren auch hier zwei Basis-Klassen *SpamLexer* und *SpamParser*, die die Grundfunktionalität zum Parsen der Nachrichten bereitstellen. Im Gegensatz zu den in Abschnitt 3.4.2 vorgestellten Filtern werden diese aber nicht nur zur Überprüfung von bekanntem Wissen verwendet – bei den MIST Filtern wurden die gesuchten n-Gramme von außen dem Filter übergeben – sondern können selbstständig, ohne Vorwissen aus einem E-Mail Datensatz einen oder auch mehrere neue Filter lernen. Damit sind die hier präsentierten *SpamLexer* und *SpamParser* bedeutend komplexer.

Um das Lernen von neuen Wörtern, E-Mail-Adressen, URLs, Dateianhängen und den aus diesen zusammengesetzten Sätzen zu ermöglichen, wurden selbstmodifizierende PYTHON-Klassen implementiert. Die Klassen modifizieren ihren eigenen Quellcode und sind so in der Lage neue Token und Regeln zu lernen, die nach einem erneuten *Import* der Klassen auch in die dem Parser zugrunde liegende Grammatik eingehen. Die zu lernenden Informationen werden im Folgenden allgemein als *Daten* bezeichnet.

In der Regel verschicken Spambots während der Analyse E-Mail-Nachrichten aus verschiedenen Spam-Kampagnen [Göbel et al., 2009]. Das heißt, für die abgefangenen Nachrichten müssen mehrere Filter (Grammatiken) gelernt werden. Eine alle Kampagnen umfassende Grammatik wäre zu generisch und damit letztlich unbrauchbar. Um alle Kampagnen optimal abzudecken, muss daher für jede abgegriffene E-Mail-Nachricht entschieden werden, welcher Kampagne sie zuzuordnen ist. Dem System sind beim Lernen einer neuen Grammatik weder die letztlich die Spam-Kampagnen abdeckende Grammatik, noch die zum Parsen der verarbeiteten Nachricht notwendigen Token und Regeln bekannt. Somit muss die Nachricht immer erst vollständig gelernt werden, bevor entschieden werden kann, ob die Nachricht zu der momentan bearbeiteten Spam-Kampagne gehört.

Der Lernalgorithmus läuft in 5 Schritten ab:

1. Von den Basisklassen *SpamLexer* und *SpamParser* werden zwei Klassen (*PrototypeLexer* und *ProtoypeParser*) abgeleitet, die die Grundgrammatik definieren.
2. Im zweiten Schritt wird die erste Nachricht dem Datensatz entnommen und dem *ParserGenerator* übergeben. Dieser importiert die Klassen *PrototypeLexer* und *ProtoypeParser* und erzeugt ein Lexer/Parser-Paar. Beim Parsen der Nachricht wird die Grammatik solange erweitert, bis die Nachricht fehlerfrei abgeleitet werden kann. Die um die Token und Regeln erweiterten Klassen *PrototypeLexer* und *ProtoypeParser* definieren die Grammatik $grammar_1$.
3. Anschließend wird die nächste Nachricht dem Datensatz entnommen und die beiden Prototyp-Klassen werden kopiert. Der *ParserGenerator* importiert die Kopien der beiden Prototyp-Klassen und erzeugt ein Lexer/Parser-Paar. Die von diesen definierte Grammatik wird solange um Token und Regeln erweitert, bis die Nachricht fehlerfrei verarbeitet werden kann. Die erzeugte Grammatik wird mit $grammar_2$ bezeichnet.

4. Die beiden Grammatiken $grammar_1$ und $grammar_2$ werden miteinander verglichen. Dabei wird bestimmt, wie viel Prozent der beim Ableiten der Nachricht verwendeten Token und Regeln schon in $grammar_1$ definiert sind (α_t, α_r) und wie viel Prozent der in $grammar_1$ definierten Token und Regeln beim Parsen angewendet wurden (β_t, β_r). Unterschreiten alle Werte $\alpha_t, \alpha_r, \beta_t$ und β_r eine Schranke ϵ, gehört die Nachricht nicht zu der aktuell bearbeiteten Spam-Kampagne und die Nachricht wird der Liste verworfener Nachrichten hinzugefügt. Liegt eines der Wertepaare α, β für die Token oder Regeln über der Schranke ϵ, wird die Nachricht als zugehörig eingestuft und die Klassen *PrototypeLexer* und *PrototypeParser* werden mit ihren in Schritt 3 erweiterten Kopien überschrieben. Der Algorithmus wird anschließend in Schritt 3 fortgesetzt.
5. Nachdem der komplette Datensatz einmal durchlaufen ist, ist die Generierung des Parsers abgeschlossen. Die die gesuchte Grammatik beschreibenden Prototyp-Klassen werden gesichert und stellen die Basis des kontextfreien Filters dar. Anschließend wird der Algorithmus auf den verworfenen Nachrichten neu gestartet.

In Abbildung 4.6 ist der Lernalgorithmus nochmals schematisch dargestellt.

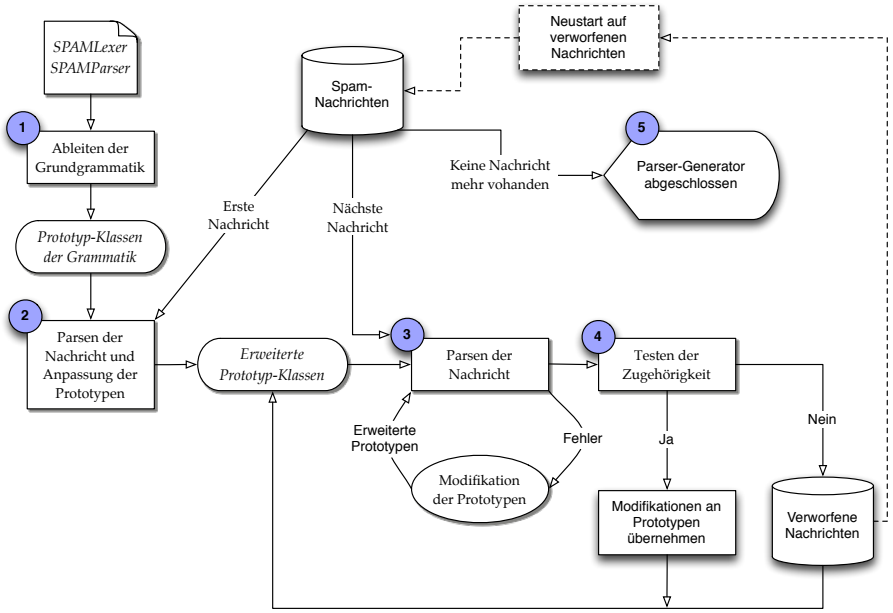


Abbildung 4.6: Schematischer Ablauf des Lernens. Die fünf Schritte des Algorithmus sind farblich markiert.

Wie dargelegt, wird das Lernen einer neuen Grammatik durch Manipulationen am Quellcode, der die Grammatik definierenden Klassen, umgesetzt. Das Lernen neuer Daten wird entweder nach dem erfolgreichen Parsen einer Nachricht, oder aus der Fehlerbehandlungsroutine heraus angestoßen: Der erste Fall tritt ein, wenn bei der Verarbeitung der Nachricht ein neuer Satz aus noch unbekannten Token geparsed wird. Der Parser kann diesen ohne Fehler aus den generischen Token (WORD, UNIT, E-

MAIL, etc.) und Regeln (`word`, `words`, `sentence`, etc.) ableiten. Anschließend muss die Erweiterung der Grammatik um neue Token und die dem Satz entsprechende Grammatikregel angestoßen werden. Wenn beim Parsen neben unbekannten Token auch bekannte Token innerhalb eines Satzes abgeleitet werden, tritt hingegen ein Fehler auf. In der Lernphase dürfen bekannte Token auch nur innerhalb der definierten Grammatikregeln geparsed werden. Erst diese Einschränkung des Parsers macht das Lernen von Modifikationen in bekannten Sätzen möglich. Die Fehlerbehandlungsroutine untersucht dabei den aktuellen Parserzustand und versucht den Fehler aufzulösen. Anschließend wird die Modifikation des Satzes in die Grammatik aufgenommen und der Parser setzt die Bearbeitung der Nachricht fort.

4.5.1 Lernen neuer Token

Das Lernen neuer Token für Wörter, E-Mail-Adressen, URLs und Dateianhänge lässt sich relativ einfach umsetzen. Unbekannte Werte werden von den generischen Token erkannt und von dem *SpamParser* zu dem entsprechenden Grammatiksymbol (`word`, `url` etc.) abgeleitet. Dabei wird der neue Wert in die Liste neu zu lernender Token aufgenommen. Nachdem das Parsen der Nachricht abgeschlossen ist, wird für alle in der Liste enthaltenen Werte ein neuer Token in den Prototypen aufgenommen. Abbildung 4.7 zeigt die Ableitungsfunktion für den WORD-Token und die dazugehörige Regel der Grundgrammatik. Die Funktionen für E-Mail-Adressen, URLs und die Dateinanhänge sind analog aufgebaut.

```
1  def t_bbody_ssubject_WORD(self, t):
2    r'[\w]+'
3    self.debug_msg('WORD FOUND (%s)' % (t.value))
4    self.token_stack.append(t)
5    return t
6    pass
```

(a)

```
1  def p_word(self, p):
2    '''word : WORD'''
3    p[0] = p[1]
4    self.word_lists.add_learn(p[1])
5    pass
```

(b)

Abbildung 4.7: Ableiten und Lernen eines noch unbekannten Wortes. (a) Token-Funktion des *SpamLexer*. (b) Grammtikregel für das Symbole `word` aus dem *SpamParser*.

In Listing 4.7 ist die für eine neue URL erzeugte Token-Funktion dargestellt. Der neue Token ist, genau wie der generische URL-Token, nur in den *SpamLexer*-Stati SSUBJECT und BBODY gültig und passt ausschließlich auf die gelernte URL *http://laqbeget.cn*.

```
def t_bbody_ssubject_URL0(self, t):
  r'http\:\/\/laqbeget\.cn'
  self.token_stack.append(t)
  return t
  pass
```

Listing 4.7: Die Grammatikregel und Funktion zum Ableiten einer gelernten URL.

4.5.2 Lernen neuer Sätze

Beim Generieren des Filters werden alle unbekannten Sätze, die sich nicht durch eine Modifikation aus bereits bekannten Sätzen herleiten lassen, als neue Sätze eingestuft. Dabei gibt es genau zwei Zustände, die das Lernen eines neuen Satzes auslösen können: (1) der *SpamParser* hat die zu verarbeitende E-Mail-Nachricht vollständig geparst und dabei Sätze über die generische Regel `sentence : words` der Grundgrammatik abgeleitet, oder (2) beim Parsen wurde eine Folge von (teilweise) bekannten Token gefunden, die nicht zu den über Grammatikregeln definierten Sätzen passt oder sich aus diesen herleiten lässt.

Im ersten Fall wird die den Satz darstellende Folge von Wörtern beim Ableiten in die Liste neu zu lernender Sätze mit aufgenommen. Diese wird, ähnlich den Listen für neu zu lernende Token, nach dem Beenden des Parsers durchlaufen und für jeden darin enthaltenen Satz wird eine neue Grammatikregel erstellt. Listing 4.8 zeigt die Funktion einer neu erstellten Grammatikregel für einen Satz aus sechs WORD-Token.

```
def p_learned_sentence_3_1(self, p): # 'any meds you want prescriptions
    written'
  '''sentence : WORD4 WORD3 WORD5 WORD6 WORD7 WORD8 '''
  self.debug:msg('sentence 3 found.')
  self.sentence_lists.seen.append('3')
  self.word_lists.seen.append('p_learned_word_4')
  self.word_lists.seen.append('p_learned_word_3')
  self.word_lists.seen.append('p_learned_word_5')
  self.word_lists.seen.append('p_learned_word_6')
  self.word_lists.seen.append('p_learned_word_7')
  self.word_lists.seen.append('p_learned_word_8')
  p[0] = p[1] + ' ' + p[2] + ' ' + p[3] + ' ' + p[4] + ' ' + p[5] + ' ' +
      p[6] + ' ' + ''
  if self.error.error:
    self.resolve_error(3)
  self.lexer.token_stack.pop(index(self.lexer.token_stack, p[1]))
  self.lexer.token_stack.pop(index(self.lexer.token_stack, p[2]))
  self.lexer.token_stack.pop(index(self.lexer.token_stack, p[3]))
  self.lexer.token_stack.pop(index(self.lexer.token_stack, p[4]))
  self.lexer.token_stack.pop(index(self.lexer.token_stack, p[5]))
  self.lexer.token_stack.pop(index(self.lexer.token_stack, p[6]))
  pass
```

Listing 4.8: Die Grammatikregel und Funktion zum Ableiten eines gelernten Satzes.

Jede für einen Satz generierte Funktion besitzt diesen Aufbau. Neben dem eigentlichen Ableiten des Satzes (Zeile 11) wird der Satz selbst und jedes in ihm enthaltene Wort in die Liste gesehener Sätze beziehungsweise Wörter eingefügt. Dies ist notwendig, um nach dem Parsen der Nachricht bestimmen zu können, wie viele der dem *SpamLexer* bekannten Token und dem *SpamParser* bekannten Sätze in der Nachricht enthalten sind. Über das Verhältnis bekannter und neuer Wörter, E-Mail Adressen, URLs, Dateinanhänge und Sätze wird anschließend bestimmt, ob die E-Mail-Nachricht der Spam-Kampagne zugeordnet wird und damit, ob die Änderungen an den Prototypen beibehalten oder verworfen werden.

Der erste Fall tritt insbesondere beim Lesen der ersten E-Mail-Nachricht ein. Zu diesem Zeitpunkt befinden sich die Prototypen des *SpamLexer*s und *SpamParser*s noch in ihrem initialen Zustand und kennen lediglich die Token und Grammatikregeln der erweiterten Grundgrammatik. Alle gelesenen Wörter werden von dem, in Abbildung 4.7 dargestellten, generischen WORD-Token abgedeckt. Jede Folge von WORD-Token lässt sich anschließend mit der generischen Regel

```
sentence : words
```

der Grundgrammatik auf einen Satz reduzieren. Nach Abschluss des Parserlaufs müssen die Token und Grammatikregeln aus den Listen gelernter Token und Sätze in die Prototypen übernommen werden.

Im zweiten Fall muss ebenfalls eine neue Regel für den unbekannten Satz erstellt werden. Allerdings wurde der Satz beim Auftreten des Fehlers noch nicht vollständig gelesen und ist folglich noch nicht bekannt. In der Fehlerbehandlungsroutine müssen deshalb solange Token manuell gelesen werden, bis ein das Satzende anzeigender PUNCTUATION-Token erreicht ist. Falls dabei WORD-Token gelesen wurden, müssen für die noch unbekannten Wörter zusätzliche Token erzeugt und dem *SpamLexer*-Protoypen hinzugefügt werden. Anschließend wird für den Satz eine Grammatikregel – diese unterschiedet sich im Aufbau nicht von den im ersten Fall erzeugten Regeln – erstellt, die in den Quellcode des *SpamParser* Prototpyen aufgenommen wird. Nachdem der *SpamParser* erneut importiert und initialisiert wurde, bekommt er die E-Mail-Nachricht nochmals übergeben und das Lernen wird fortgesetzt.

4.5.3 Lernen von Modifikationen in bekannten Sätzen

Ob eine Nachricht der aktuell bearbeiteten Spam-Kampagne zugeordnet werden kann, wird über das Verhältnis zwischen den gesehenen und erwarteten Token und Grammatikregeln bestimmt. Somit ist es wichtig, dass der *SpamParser* erkennt, ob es sich bei einem unbekannten Satz um einen gänzlich neuen Satz handelt, oder ob er nur die Modifikation eines bereits bekannten Satzes darstellt. Ohne diese Unterscheidung würde jeder unbekannte Satz als neuer Satz in den *SpamParser* aufgenommen und in die Liste der gelernten Sätze eingetragen. Für Sätze, die durch eine Modifikation aus einem bereits bekannten Satz hervorgehen, geht die „Verwandtschaft“ zu dem bekannten Satz verloren. Dadurch verschiebt sich auch das zur Zuordnung verwendete Verhältnis zwischen alten und neuen Sätzen, da statt des bekannten Satzes ein neuer erkannt wurde.

Dies kann sehr schnell zu einer Fehlklassifizierung der Nachricht führen. Das Problem wird an dem folgenden Beispiel verdeutlicht.

```
100 Prozent kostenlos anmelden und Profil anlegen unter:
http://../kontakte-germany
```

(a)

```
100 Prozent kostenlos anmelden und ein Profil anlegen unter:
http://../kontakte-germany
```

(b)

Abbildung 4.8: Die in den Listings (a) und (b) gezeigten Nachrichten unterscheiden sich in nur einem Wort.

Die in Abbildung 4.8 dargestellten (fiktiven) E-Mail-Nachrichten unterscheiden sich nur in dem in der zweiten Nachricht zusätzlich enthaltenen Wort „ein“. Nach dem Parsen der ersten Nachricht sind dem Prototypen sowohl der Satz, als auch die URL bekannt. Das Parsen der zweiten Nachricht wird beim Lesen des zusätzlichen Wortes „ein“ mit einem Fehler unterbrochen. Ohne die Überprüfung auf Modifikationen würde für den aufgrund des zusätzlichen Wortes neuen Satz ein zweiter Satz in den *SpamParser* aufgenommen. Nach der Neuinitialisierung des *SpamParser*s könnte dann auch die zweite Nachricht erfolgreich abgeleitet werden. Die anschließende Überprüfung, ob die Nachricht auch Teil derselben Spam-Kampagne ist, würde – zumindest auf die Sätze bezogen – negativ ausfallen. Von den dem *SpamParser* vor dem Bearbeiten der zweiten Nachricht bekannten Sätzen wurden 0 Prozent in der zweiten Nachricht gefunden und keiner der in der zweiten Nachricht gefundenen Sätze war vorab bekannt.

In dem momentan implementierten *SpamParser* werden Sätze, die sich durch genau eine Modifikation in einen bereits bekannten Satz überführen lassen, als bereits bekannt eingestuft. Mögliche Modifikationen sind hierbei das Auslassen eines Wortes, das Ersetzen eines Wortes, das Einfügen eines zusätzlichen Wortes oder das Vertauschen zweier aufeinander folgender Wörter.

Jedes unerwartete Lesen eines bekannten Wortes, oder das Lesen eines beliebigen unerwarteten Wortes innerhalb eines bekannten Satzes hat das Auslösen eines Fehlers durch den *SpamParser* zur Folge. Innerhalb der Fehlerbehandlungroutine erfolgt die Prüfung, ob sich der Fehlerzustand durch eine der Modifikationen beheben lässt. Ist dies der Fall, werden die dazu nötigen Modifikation durchgeführt. Die vorgenommenen Veränderungen werden dabei solange in einem Fehlerfeld (`error`) des *SpamParser*s zwischengespeichert, bis der modifizierte Satz erfolgreich abgeleitet werden kann. Das Fehlerfeld wird erst beim Ableiten überprüft und stößt – falls notwendig – das Erweitern des *SpamParser*s um eine zusätzliche Grammatikregel für den aktuellen Satz an (siehe Zeile 12 und 13 in Listing 4.8). Die übergebene Satznummer fließt

in den Funktionsbezeichner (`p_learned_sentence_`*satznummer*`_X`) ein und legt fest, unter welcher Nummer der neue Satz in die Liste gesehener Sätze aufgenommen wird. Nachdem die neue Grammatikregel dem *SpamParser* Prototypen hinzugefügt wurde, wird dieser neu gestartet.

4.5.4 Fehlerbehandlungsroutine

Beim Lernen (Parsen) neuer Nachrichten dürfen bekannte Token nur innerhalb der definierten Grammatikregeln angewendet werden. Sobald eine *Vermischung* zwischen bekannten und unbekannten Token stattfindet, ist diese Bedingung nicht mehr erfüllt und der Parser löst einen Fehler aus. In der *Fehlerbehandlungsroutine* wird anschließend untersucht, was den Fehler ausgelöst hat und wie sich dies beheben lässt. Dazu werden die folgenden Informationen über den momentanen Zustand des Parsers aus den Parser- und Lexer-Objekten ausgelesen und ausgewertet:

`found_token`. Der den Fehler auslösende Token wird der Fehlerbehandlungsroutine direkt übergeben. Die in dem Token enthaltenen Informationen sind der *Wert* und der *Typ* des Tokens sowie die Position des Tokens innerhalb des zu verarbeitenden Textes.

`error`. Das Fehlerfeld des *SpamParser*s speichert alle für den letzten in der Fehlerbehandlungsroutine gelösten Fehler interessanten Informationen, bis der bei der Lösung entstandene Satz erfolgreich vom Parser abgeleitet wird. Ist das Fehlerfeld beim Eintritt in die Fehlerbehandlungsroutine bereits belegt, unterschiedet sich der vorliegende Satz an mindestens zwei Stellen von bekannten Sätzen und muss als neuer Satz gelernt werden. Zu den im Fehlerfeld gespeicherten Informationen gehören unter anderem die *Fehlerart* und die bei der Lösung des Fehlers durchgeführten Modifikationen.

`token_stack`. In dem *Token-Stack* sind alle seit der letzten Satz-Reduktion gelesenen Token enthalten. Der Stack wird während der Ableitung von Token aufgebaut, siehe Zeile 4 in Abbildung 4.7(a) und Zeile 3 in Listing 4.7, und beim erfolgreichen Ableiten von Sätzen wieder abgebaut (vergleiche die Zeilen 14 bis 19 in Listing 4.8).

`upcoming_token`. Alle bis zur nächsten Reduktion – bis zum nächsten Satzzeichen – noch von der *Eingabe* zu lesenden Token werden in dieser Liste gespeichert.

`parser_state`. Der aktuelle *Stand* des *SpamParsers* wird benötigt, um die vom *SpamParser* als nächstes erwarteten Token bestimmen zu können. Um aus der Fehlerbehandlungsroutine auf den Stand des *SpamParsers* zugreifen zu können, sind Änderungen an dem Modul *ply* notwendig. Der *Patch* ist im Anhang A in Listing A.3 dargestellt.

`expected_token`. Die bis zur nächsten Reduktion erwarteten Token werden in dieser *Baumstruktur* gespeichert. Die Baumstruktur ist notwendig, weil der Parser in den meisten Zuständen verschiedene Token als *Eingabe* akzeptiert.

Der Ablauf der Fehlerbehandlungsroutine ist in Abbildung 4.9 vereinfacht dargestellt. Als erstes wird hier überprüft, ob schon ein Fehler im Fehlerfeld `error` eingetragen ist. Ist dies der Fall, existiert in dem aktuell verarbeiteten Satz mehr als ein

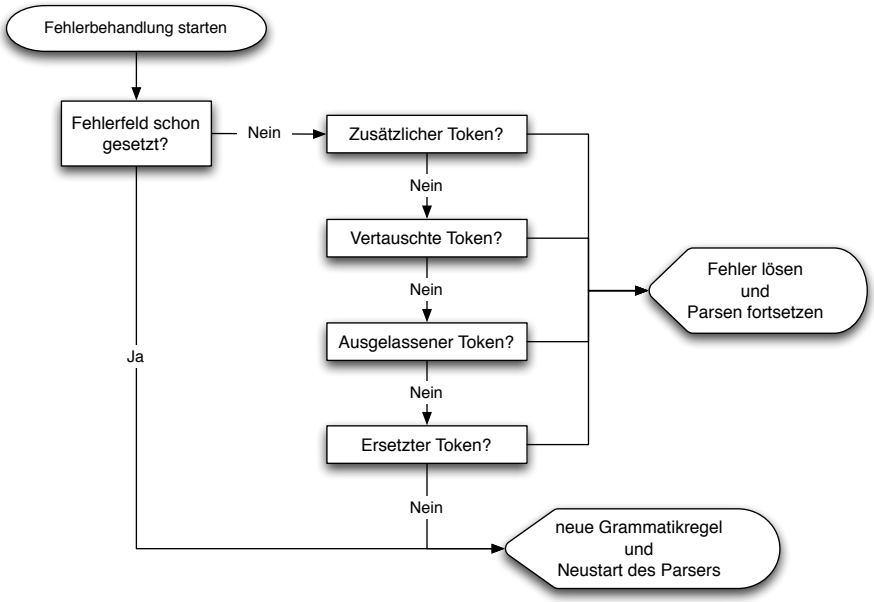


Abbildung 4.9: Schematischer Ablauf der Fehlerbehandlungsroutine.

Unterschied zu den bekannten Sätzen und der Fehler lässt sich nicht lösen. Stattdessen wird für den Satz eine neue Grammatikregel erzeugt und in den *SpamParser* aufgenommen. Dazu wird die Liste bereits gelesener Token (`token_stack`) mit der Liste der noch kommenden Token (`upcoming_token`) verknüpft. Der daraus entstehende Satz, sowie alle darin vorkommenden noch unbekannten Token, werden der Grammatik hinzugefügt. Anschließend wird der *SpamParser* neu gestartet und erhält die aktuelle Nachricht nochmals zur Bearbeitung vorgelegt.

Ist das Fehlerfeld beim Eintreten in die Fehlerbehandlungsroutine leer, werden nacheinander die Routinen `check4additional`, `check4swap`, `check4skip` und `check4replacement` ausgeführt. Mit diesen Heuristiken lassen sich viele Fehler schon in der Fehlerbehandlungsroutine „beheben“. Dazu wird der Fehler, anhand der über den Fehler gesammelten Informationen, analysiert und falls möglich durch die entsprechende Modifikation in dem zu parsenden Text gelöst. Die für die Erweiterung des *SpamParser*s notwendigen Informationen werden im Fehlerfeld gespeichert. Anschließend wird der Fehlerstatus des *SpamParsers* zurückgesetzt und der *SpamParser* setzt die Verarbeitung der Nachricht fort.

Im Folgenden werden die einzelnen Routinen zur Fehlerbehandlung im Detail vorgestellt:

`check4additional`. Um zu überprüfen, ob es sich bei dem den Fehler auslösenden Token lediglich um einen zusätzlichen Token innerhalb eines bereits bekannten Satzes handelt, werden die kommenden Token (`upcoming_token`) mit den erwarteten Token (`expected_token`) verglichen. Wenn der auf den den Fehler auslösenden Token folgende Token (`upcoming_token[1]`) in der Liste

der erwarteten Token (`expected_token`) enthalten ist, werden – falls vorhanden – auch die nachfolgenden Token der beiden Listen miteinander verglichen. Dadurch wird sichergestellt, dass es sich bei dem zusätzlichen Token um den einzigen Unterschied zwischen den Sätzen handelt.

Um den Fehler zu beheben, wird der gefundene Token (`found_token`) unter dem Fehlertype `addw` im Fehlerfeld gespeichert und dem *SpamParser* der erwartete Token (`upcoming_token[1]`) übergeben. Mit diesem kann anschließend der komplette Satz abgeleitet werden. Bei der Ableitung des Satzes wird, wie in Listing 4.8 dargestellt, das Fehlerfeld ausgewertet und eine neue Variante der Grammatikregel gelernt.

`check4swap`. Diese Routine versucht, den Fehler durch das Vertauschen von zwei aufeinander folgenden Token zu lösen. Dazu wird überprüft, ob der als nächstes zu bearbeitende Token (`upcoming_token[1]`) in der Liste der erwarteten Token (`expected_token`) enthalten ist und ob auf diesen der gefundene Token (`found_token`) folgen darf (`expected_token[upcoming_token[1]] = found_token`).

Sind beide Bedingungen erfüllt, wird der gefundene Token unter dem Fehlertype `swap` im Fehlerfeld abgespeichert und die beiden zu vertauschenden Token werden in der *Eingabe* selbst vertauscht. Damit kann der Parser den bekannten Satz fehlerfrei ableiten. Neben dem Fehlertyp wird auch die ursprüngliche Token-Folge im Fehlerfeld gespeichert. Diese wird nach dem Ableiten des Satzes ausgelesen und als neue Variante des Satzes in die Grammatik aufgenommen.

`check4skip`. Die Überprüfung, ob ein erwarteter Token (`exptok`) ausgelassen werden muss, um den Fehler zu lösen, ähnelt der Überprüfung auf einen zusätzlichen Token. Allerdings wird hier der gefundene Token (`found_token`) selbst mit den Nachfolgern der erwarteten Token verglichen. Werden dabei Übereinstimmungen gefunden, erfolgt auch hier eine schrittweise Überprüfung der jeweils folgenden Token.

Stimmt die Liste der gefundenen Token (`upcoming_token`) mit einem Pfad durch die erwarteten Token (`expected_token[exptok]`) überein, werden im Fehlerfeld der Fehlertyp `skip` und der ausgelassene Token abgespeichert. Anschließend wird dem *SpamParser* der ausgelassene Token übergeben. Bei der Ableitung des Satzes erkennt der Parser den `skip`-Fehler und erstellt auf Basis der im Fehlerfeld enthaltenen Informationen eine entsprechend modifizierte Grammatikregel.

`check4replacement`. Die Routine zur Überprüfung, ob der aufgetretene Fehler durch das Ersetzen eines einzelnen Wortes hervorgerufen wurde, stellt – wie das Ersetzen eines Tokens an sich – eine Kombination zwischen dem Auslassen und dem Hinzufügen eines Tokens dar. Dazu wird überprüft, ob der auf den gefundenen Token folgende Token (`upcoming_token[1]`) mit einem Nachfolger der erwarteten Token übereinstimmt. Ist dies der Fall, wird auch hier ein mit der Eingabe übereinstimmender Pfad in der Liste der erwarteten Token gesucht.

Falls ein passender Pfad gefunden wird, werden der Fehlertype `repl` und der den Fehler auslösende Token im Fehlerfeld gespeichert. Dem *SpamParser* wird

der erste Token auf dem gefunden Pfad übergeben. Nachdem der Satz vollständig geparsed ist, wird das Generieren einer neuen Variante des Satzes, mit der Ersetzung, angestoßen.

Das beschriebene Vorgehen stellt nur die Grundidee für diese Routine dar und wird lediglich bei den ersten Ersetzungen innerhalb eines gelernten Satzes so angewandt. Prinzipiell deutet die Ersetzung eines Tokens innerhalb eines Satzes darauf hin, dass der Token selbst nicht zur Vorlage der Kampagne gehört, sondern als variabler Teil dort eingefügt wird. Falls dies tatsächlich der Fall ist, würden bei dem beschriebenen Vorgehen leicht unzählige, unnötige Sätze gelernt, wodurch die Performanz des *SpamParser*s stark abnimmt. Um dem vorzubeugen, wird innerhalb der Routine untersucht, ob schon mehrere Varianten des Satzes existieren, die durch einen generischen Satz abgedeckt werden könnten. Als Schranke wird dabei das Verhältnis zwischen der Satzlänge und der Anzahl der bekannten Satzvarianten verwendet. Sobald die Anzahl an Varianten die doppelte Satzlänge überschritten hat, wird ein generischer Satz erzeugt. Innerhalb dieses generischen Satzes wird der zu ersetzende Token durch einen generischen WORD- Token ausgetauscht und passt damit auf jede an dieser Stelle mögliche Zeichenkette[3].

Falls keine der Fehlerbehandlungsroutinen den Fehler lösen kann, werden die Listen für bereits gelesene Token (`token_stack`) und noch kommende Token (`upcoming_token`) miteinander verknüpft und der *SpamParser* wird um eine neue Grammatikregel für diesen Satz erweitert. Nachdem der *SpamParser* neu initialisiert ist, wird die aktuelle Nachricht von neuem geparst.

4.5.5 Zuordnung der Nachrichten zu Spam-Kampagnen

Die Zuordnung der einzelnen Nachrichten zu den möglicherweise parallel laufenden Spam- Kampagnen wird über das Verhältnis zwischen den gesehenen und erwarteten Daten bestimmt. Entsprechende Listen existieren für die Wörter, die E-Mail-Adressen, die URLs, die Dateianhänge und die Sätze. Die Schnittmenge zwischen den jeweiligen Listen enthält diejenigen Daten, die gesucht und auch abgeleitet wurden. Mit Hilfe dieser Schnittmenge lässt sich für jedes Datum bestimmen, wie viel Prozent der gesuchten Daten gefunden wurden und wie viele der gefundenen Daten bereits bekannt waren.

In der aktuellen Implementierung werden nur die Listen für die Wörter und Sätze ausgewertet und bei der Zuordnung beachtet. Für die übrigen Daten konnten noch keine optimalen Schranken bestimmt werden.

[3] Nicht als WORDX-Token gelernte Zeichenketten, zum Beispiel Zahlen, werden ohnehin schon durch einen generischen Token ersetzt.

4.6 Filtern mit LEX und YACC

Die *SpamFilter* bekommen die zu filternden Nachrichten und die generierten Prototypen für den *SpamLexer* und den *SpamParser* übergeben. Während der Lexer unverändert eingesetzt werden kann, muss der Parser noch um Grammatikregeln zum Ableiten der einzelnen Token erweitert werden. Listing 4.9 zeigt die zu erstellende Grammatikregel an einem Beispiel. Durch die zusätzlichen Regeln ist es dem *SpamParser* möglich, die gelernten Wörter auch innerhalb neuer Sätze zu verwenden.

```
1  def p_learned_word_31(self, p):
2    '''word : WORD31'''
3    self.debug_msg( 0, 'word WORD31 found.')
4    self.seen_words.append(31)
5    p[0] = p[1]
6    pass
```

Listing 4.9: Die Grammatikregel und Funktion zum Ableiten eines gelernten Wortes.

Die zusätzlichen Grammatikregeln sind notwendig, damit sich beim Parsen der Nachrichten auch unbekannte, aber aus bekannten Wörtern bestehende Sätze ableiten lassen. Diese zusätzliche Möglichkeit beim Parsen stellt den Hauptunterschied zwischen dem lernenden *SpamParser* und dem filternden *SpamParser* dar. In der Lernphase wurde auf die Ableitungsregeln für einzelne Wörter verzichtet, um über die Fehlerbehandlungsroutine Modifikationen der bekannten Sätze oder vollständig neue Sätze zu erkennen und lernen zu können. Beim Filtern ist dies nicht mehr notwendig. Stattdessen muss hier sichergestellt werden, dass die Nachricht ohne Fehler geparst wird, um anschließend die gefundenen Daten mit denen des Filters abgleichen zu können. Die Zuordnung der Nachrichten zu einer Spam-Kampagne erfolgt anhand derselben Kriterien, die auch bei der Konstruktion des *SpamParser*s eingesetzt wurden.

4.6.1 Auswertung der *kontextfreien* Spam-Filter

Um die Effektivität der auf kontextfreien Grammatiken beruhenden Spam-Filter zu evaluieren, wurden dieselben Spam-Läufe herangezogen, die auch zur Evaluierung der auf regulären Ausdrücken basierenden Spam-Filter verwendet wurden. Basierend auf den Filterergebnissen können die beiden Ansätze damit direkt gegenübergestellt und miteinander verglichen werden. Tabelle 4.6 zeigt nochmals die Details zu den verwendeten Spam-Läufen. Die Kampagnenanzahl ist dabei von den Filterergebnissen abgeleitet, da hierzu keine gesicherten Informationen existieren.

Spambot	*Spam-Läufe*	*Nachrichten*	*Kampagnen*
STORM	437	2.185.000	437
GHEG	1	255.622	1
SRIZBI	1	1.000.000	5-13

Tabelle 4.6: Zur Evaluierung der Spam-Filter herangezogene Botnetze und Spam-Läufe.

Analog zu der in Abschnitt 4.3.2 vorgestellten Evaluierung der einfachen, auf regulären Ausdrücken basierenden Filter wurden die Filter jeweils für Teilmengen der Nachrichten, das heißt, für jeweils 0,001 Prozent, 0,01 Prozent, 0,1 Prozent, einem Prozent, 10 Prozent und – falls möglich – 100 Prozent der Nachrichten, erzeugt und anschließend zum Filtern der Gesamtdaten verwendet. Dieser Prozess wurde für jeden Datensatz und jede Kampagne bis zu 10 mal wiederholt, um stabile Ergebnisse zu erhalten. In Tabelle 4.7 sind die Anzahl der Wiederholungen für die untersuchten Spambots und den Prozentsatz der jeweils zur Filtergenerierung herangezogenen Nachrichten aufgelistet.

Spambot	*0,001%*	*0,01%*	*0,1%*	*1%*	*10%*	*100%*
STORM*	-*[2]	10	10	10	10	1
SRIZBI*[3]	10	10	10	10	5	-
GHEG	10	10	10	10	7	10

Tabelle 4.7: Die Anzahl der durchgeführten Filtergenerierungsprozesse je Spambot. (* Die Analyseläufe wurden für jeden der 437 Spam-Läufe des STORM-Botnetzes durchgeführt. *[2] Das Minimum von einer Nachricht liegt bereits für 0,01 Prozent der Nachricht vor. *[3] Aktuell liegen nur die Filterergebnisse jeweils einer Ausführung vor.)

Der hier vorgestellte Filteransatz ist sehr rechenintensiv. Das stetige Anpassen der Grammatik während der Lernphase macht die Filtergenerierung langsam und lässt sich nicht parallelisieren. Deshalb ist es umso wichtiger, dass der Filter-Generierungsprozess nur sehr wenige Nachrichten benötigt, um Filter zu berechnen, die die gesamte Spam-Kampagne abdecken, ohne dabei *false-positives* zu verursachen.

Analyse von 437 Spam-Kampagnen des STORM-Botnetzes.

Aus dem STORM-Botnetz wurden 437 jeweils 5.000 E-Mail-Nachrichten umfassende Spam-Läufe zur Evaluierung des Filteransatzes herangezogen. Mit über 2 Millionen Nachrichten ist dies die umfassendste Analyse. Tabelle 4.8 zeigt die durchschnittlich erreichten Erkennungsraten über alle 437 Spam-Läufe.

Lerndatensatz		**Filterergebnisse**	
# E-Mails	*in %*	*Kampagnen**	*Erkennungsrate*
1	0,01	1	99,7578 %
5	0,1	1	99,5373 %
50	1,0	1	99,0036 %
500	10	1	99,9997 %
5000	100	1	100,0000 %

Tabelle 4.8: Filterergebnisse für das STORM-Botnetz. Die Angaben sind Durchschnittswerte über alle 437 Spam-Läufe und die 10 pro Lauf und Teilmenge erzeugten Filter. (* Die Anzahl an Kampagnen bezieht sich jeweils auf einen Spam-Lauf.)

Der Filteransatz liefert mit über 99 Prozent Erkennungsrate durchweg sehr gute Ergebnisse. Schon der auf einer einzigen Nachricht basierende Filter ist gut genug, um die Spam-Kampagnen zuverlässig zu filtern. Für die auf regulären Ausrücken basierenden Filter liegt die durchschnittliche Erkennungsrate nach dem Lernen einer Nachricht

noch bei rund einem Prozent. Erst Filter, die auf 50 Nachrichten basieren, können ähnlich hohe Erkennungsraten aufweisen, auch wenn der Durchschnitt für diese Filter mit 92,5 Prozent immer noch bedeutend schlechter ist.

Aus welchem Grund die Erkennungsrate für die aus 5 und 50 Nachrichten gelernten Filter geringer ist, als die der nur auf einer Nachricht basierenden Filter, kann aufgrund der großen Anzahl an Filter nicht zweifelsfrei geklärt werden. Erste Analysen zeigen, dass die schlechteren Ergebnisse nicht auf einzelne Ausreißer mit sehr schlechten Erkennungsraten zurückzuführen sind, sondern dass viele der Filter leicht schwächere Ergebnisse liefern. Das Gesamtergebnis wird damit von sehr vielen Filtern beeinflusst, was eine detaillierte Untersuchung zusätzlich erschwert.

Die auf einer Grammatik basierenden Filter bestehen, stark vereinfacht, aus gelernten Worten und Sätzen. Diese beschreiben, wie die während der Filtergenerierung bearbeiteten E-Mail-Nachrichten aufgebaut sind. Das heißt, der Filter kennt genau die Wörter und Sätze, die ihm zum Lernen vorgelegt wurden. In Tabelle 4.9 sind die Anzahl der im Durchschnitt von den Filtern gelernter Wörter und Sätze aufgelistet.

Lerndatensatz	**Filterdaten**	
# E-Mails	*# gelernter Wörter*	*# gelernter Sätze*
1	112,28	23,07
5	123,83	28,09
50	189,26	65,93
500	231,54	98,61
5.000	231,56	98,63

Tabelle 4.9: Anzahl der im Mittel gelernten Wörter und Sätze für die 437 im STORM-Botnetz generierten Spam-Läufe.

Aus den gelernten Daten lässt sich ableiten, dass eine E-Mail-Nachricht der analysierten Spam-Läufe durchschnittlich aus 112 Wörtern und 23 Sätzen besteht und dass die innerhalb einer Kampagne verschickten Nachrichten zusammen aus rund doppelt so vielen Wörtern, und fast viermal so vielen Sätzen bestehen. Da schon die auf nur einer Nachricht generierten Filter eine Erkennungsrate von über 99 Prozent aufweisen, muss innerhalb der Spam-Kampagnen jeweils eine Grundmenge von Wörtern und Sätzen existieren, die in jeder E-Mail-Nachricht einer Kampagne enthalten sind. Die nur langsam steigende Anzahl an Wörtern und Sätzen weist zudem darauf hin, dass die Nachrichten neben der Grundmenge nur sehr wenige zusätzliche Wörter und Sätze enthalten. Das Vorhandensein der Grundmenge und die geringe Varianz durch zusätzliche Wörter und Sätze kommen dem auf Grammatiken basierten Filteransatz entgegen und erklären die für das STORM-Botnetz erreichten, sehr guten Ergebnisse.

Analyse eines Spam-Laufs des GHEG-Botnetzes.

Aus dem GHEG-Botnetz wurde ein Spam-Lauf mit insgesamt 255.622 E-Mail-Nachrichten analysiert. Für den gesamten Spam-Lauf wurde durch den Algorithmus ein einzelner Spam-Filter generiert, das heißt, der GHEG-Bot hat nur eine Spam-Kampagne verschickt. Die durchschnittlichen Erkennungsquoten der zehn pro Stufe generierten Filter liegen für die auf drei Nachrichten gelernten Filter bei 93,108 Pro-

zent und für den auf 99,6 Prozent der E-Mail-Nachrichten gelernten Filter bei 99,994 Prozent. Ein auf allen Nachrichten basierender Filter, der zwangsläufig auch eine hundertprozentige Erkennungsrate besitzt, konnte für diesen Spam-Lauf nicht berechnet werden, da 0,4 Prozent der Nachrichten Steuerzeichen innerhalb der Betreffzeile besitzen, die von dem Filtergenerator fehlerhaft abgeleitet wurden. Beim Filtern tritt dieser Fehler nicht auf, weil hier keine Wörter oder Sätze mehr gelernt werden. Tabelle 4.10 zeigt die durchschnittlichen Erkennungsraten für die einzelnen Filterstufen.

Lerndatensatz		**Filterergebnisse**	
# E-Mails	*in %*	*Kampagnen*	*Erkennungsrate*
3	0,001 %	1	93,108 %
26	0,01 %	1	97,353 %
256	0,1 %	1	99,993 %
2.556	1 %	1	98,422 %
25.459	10 %	1	99,993 %
254.591	99,6 %	1	99,994 %

Tabelle 4.10: Über alle Analyseläufe gemittelte Filterergebnisse auf den GHEG-Spam-Daten.

Mit einer rund 93-prozentigen Erkennungsrate bei nur drei gelernten Nachrichten ist der auf kontextfreien Grammatiken basierende Filteransatz bedeutend besser als der auf regulären Sprachen basierende Ansatz. Für letzteren liegt die Erkennungsrate des schwächsten Filters bei nur 0,3366 Prozent. Eine Erkennungsrate von 93 Prozent konnte für die in Abschnitt 4.3.2 evaluierten Filter erst auf der nächsthöheren Stufe, das heißt, für den auf 26 Nachrichten generierten Filter, erreicht werden.

Lerndatensatz	**Filterdaten**	
# E-Mails	*# gelernter Wörter*	*# gelernter Sätze*
3	151,80	39,80
26	189,80	58,40
256	202,00	65,00
2.556	198,30	63,40
25.459	202,43	65,40
256.591	208,00	71,00

Tabelle 4.11: Anzahl der im Mittel gelernten Wörter und Sätze für den im GHEG-Botnetz generierten Spam-Lauf. Jeweils in Abhängigkeit von der Anzahl zur Filtergenerierung verwendeter E-Mail-Nachrichten.

Die Anzahl der im Durchschnitt pro Filter gelernten Wörter und Sätze ist in Tabelle 4.11 dargestellt.[4] Mit rund 152 Wörtern und 40 Sätzen sind nach dem Lernen von nur drei Nachrichten schon über 72 Prozent beziehungsweise 56 Prozent aller in der Kampagne verwendeten Wörter und Sätze bekannt. Dies führt zu den sehr guten Erkennungsraten. Ähnlich den Kampagnen im STORM-Botnetz, steigt auch für diesen Spam-Lauf des GHEG-Botnetzes die Anzahl der verwendeten Wörter und Sätze

[4] Die Tabellen E.1 bis E.6 in Anhang E liefert eine detaillierte Auflistung der einzelnen Filterergebnisse und der pro Filter gelernter Wörter und Sätze.

nur sehr langsam an und besitzt mit 208 Wörtern und 71 Sätzen ein niedriges Maximum. Nach diesem Muster aufgebaute Spam-Kampagnen werden – wie bereits im Zusammenhang mit der Analyse des STORM-Botnetzes erwähnt – von dem hier vorgestellten, auf kontextfreien Grammatiken beruhenden Filteransatz, sehr gut und schnell abgedeckt.

Analyse eines Spam-Laufs des SRIZBI-Botnetzes.

Für die aus dem SRIZBI-Botnetz abgefangenen E-Mail-Nachrichten berechnet der Filtergenerator in allen Analyseläufen mehrere Kampagnen. Die Anzahl schwankt dabei, in Abhängigkeit von der Anzahl zur Filtergenerierung herangezogener Nachrichten, zwischen 5 und 13 Kampagnen. Von den zehn auf nur 0,001 Prozent der Nachrichten durchgeführten Filtergenerierungsprozesse werden dabei nur 5–6 Kampagnen berechnet. Dies lässt sich mit der geringen Anzahl analysierter Nachrichten erklären. Mit 10 Nachrichten wurden weniger Nachrichten analysiert, als Kampagnen im Datensatz enthalten sind. Zudem ist die Wahrscheinlichkeit, dass die zehn zufällig ausgewählten Nachrichten aus zehn unterschiedlichen Kampagnen stammen sehr gering. Die auf 0,1 Prozent, einem und zehn Prozent der Nachrichten gestarteten Filtergenerierungsprozesse berechnen zwischen 11 und 13 Kampagnen. Da in diese Analyseläufe mehr Nachrichten einflossen und auch die in Abschnitt 4.3.2 präsentierte Analyse mit regulären Filtern 11 Kampagnen identifizierte, kann davon ausgegangen werden, dass der Datensatz Nachrichten aus mindestens 11 unterschiedlichen Spam-Kampagnen enthält. Tabelle 4.12 gibt einen Überblick über die Anzahl identifizierter Kampagnen und die Erkennungsraten, die die Filter insgesamt auf dem Datensatz erreichen. Da aktuell nicht für alle berechneten Filter auch die Filterergebnisse auf dem eine Millionen Nachrichten umfassenden Datensatz vorliegen, beziehen sich die in Tabelle 4.12 dargestellten und im Folgenden im Detail präsentierten Erkennungsraten auf jeweils einen Filtergenerierungsprozess pro Teilmenge.

Lerndatensatz		**Filterergebnisse**	
# E-Mails	*in %*	*Kampagnen*	*Erkennungsraten insgesamt*
10	0,001%	5	56,7392 %
100	0,01%	9	97,27 %
1.000	0,1%	13	99,90 %
10.000	1%	12	100,00 %
100.000	10%	13	100,00 %

Tabelle 4.12: Filterergebnisse auf den SRIZBI-Spam-Daten. Neben der Anzahl in dem SRIZBI-Datensatz erkannter Kampagnen ist auch die insgesamt erreichte Erkennungsrate, das heißt, der Prozentsatz der von den generierten Filtern zusammen abgedeckten Nachrichten, dargestellt.

Die Erkennungsraten der für den Datensatz generierten Filter sind insgesamt sehr gut. Mit den 9 auf insgesamt nur 100 Nachrichten basierenden Filtern lassen sich bereits über 97 Prozent (972.721 Nachrichten) der im SRIZBI-Datensatz enthaltenen Nachrichten filtern. Die auf 1.000 Nachrichten basierenden Filter erkennen zusammen 99.9 Prozent aller Nachrichten und die aus einem und zehn Prozent der Nachrichten berechnenden Filter decken alle im Datensatz enthaltenen Nachrichten ab. Die Erken-

nungsraten für die einzelnen Filter sind in Tabelle 4.13 dargestellt. Mit Ausnahme der Filter 12 und 13 erreichen alle Filter schon sehr früh eine hundertprozentige Erkennungsrate innerhalb ihrer Kampagne. Als Bezugsgröße wurden wieder die Filterergebnisse der auf 10 Prozent der Nachrichten generierten Filter herangezogen.[5]

Insgesamt zeigt die Analyse auch für den SRIZBI-Spam-Datensatz wieder den größten Vorteil des kontextfreien Filteransatzes: die Filter sind flexibel und können schon nach dem Lernen nur weniger Nachrichten den Großteil der zu der jeweiligen Kampagne gehörenden Nachrichten erkennen.

	Erkennungsrate in Prozent				
	0,001%	*0,01%*	*0,1%*	*1,0%*	*10%*
Filter 1	89,43	100,00	100,00	100,00	100,00
Filter 2	88,96	100,00	100,00	100,00	100,00
Filter 3	8,34	100,00	100,00	100,00	100,00
Filter 4	100,00	87,26	100,00	100,00	100,00
Filter 5	100,00	-	87,46	100,00	100,00
Filter 6	-	100,00	100,00	100,00	100,00
Filter 7	-	100,00	100,00	100,00	100,00
Filter 8	-	100,00	100,00	100,00	100,00
Filter 9	-	100,00	100,00	100,00	100,00
Filter 10	-	-	100,00	100,00	100,00
Filter 11	-	100,00	100,00	100,00	100,00
Filter 12	-	-	25,26	43,10	100,00
Filter 13	-	-	100,00*	-	95,20

Tabelle 4.13: Erkennungsraten für die auf dem SRIZBI-Spamdatensatz generierten Filter. Als 100 Prozent Bezugsgröße wurden die Filterergebnisse der aus 10 Prozent der Nachrichten erstellten Filter herangezogen. (* Da das Filterergebnis des auf einem Prozent der Nachrichten generierten Filters besser ist, wurde dieser als 100 Prozent Bezugsgröße verwendet.)

Die in Tabelle 4.13 enthaltene Zuordnung der Filter zu den Kampagnen basiert auf den Filterergebnissen der einzelnen Filter. Eine exakte Zuordnung der Filter anhand der in diese eingeflossenen Nachrichten ist nicht möglich, da mit der aktuellen Implementierung kein Filtergenerierungsprozess auf allen Nachrichten des SRIZBI-Datensatzes durchgeführt werden kann. Zudem erfolgt die Auswahl der zum Lernen der Filter herangezogenen Nachrichten zufällig und die Filter basieren damit fast ausschließlich auf unterschiedlichen Nachrichten. Trotzdem kann die auf den Filterergebnissen basierende Zuordnung – wenigstens für die Filter 1 bis Filter 11 – als korrekt angesehen werden. Die Filterergebnisse der einzelnen Filter innerhalb der Kampagnen sind größtenteils identisch und zwischen den elf Kampagnen existieren fast keine Überschneidungen.[6] Mit Ausnahme von Filter 12 und Filter 13 lassen sich somit alle Filter allein anhand ihres Filterergebnises zweifelsfrei einer Kampagne zuordnen. Bei den beiden letzten Kampagnen ist die Überdeckung der Filterergebnisse nicht so ausgeprägt. Diese Filter basieren nur auf sehr wenigen Nachrichten, siehe Tabelle 4.14,

[5] Die in Tabelle 4.12 dargestellten Erkennungsraten werden trotz der schlechten Erkennungsrate von Filter 12 und 13 erreicht, da die beiden Kampagnen nur wenige Nachrichten umfassen, die zudem bereits von den übrigen elf Filtern abgedeckt werden.

[6] Tabelle E.7 in Anhang E zeigt die Filterergebnisse der einzelnen Filter im Detail.

und sind damit relativ spezifisch. Sie filtern zwar den Großteil der in der Kampagne enthaltenen Nachrichten, besitzen aber noch nicht die notwendige Varianz, um diese vollständig abzudecken.

Tabelle 4.14 zeigt die Anzahl der Nachrichten, die in die einzelnen Filter eingeflossen sind. Die zufällig zur Filtergenerierung herangezogenen Nachrichten verteilen sich sehr ungleichmäßig auf die einzelnen Filter. Wie bereits erwähnt, ist insbesondere die Basis von Filter 12 und 13 sehr dünn. Für die vierte, fünfte, achte und elfte Kampagne existieren allerdings ebenfalls Filter, die auf nur einer Nachricht basieren. Im Gegensatz zu den Filtern 12 und 13 besitzen diese trotzdem eine 100-prozentige Erkennungsrate innerhalb ihrer Kampagne.

	Anzahl der gelernten E-Mail-Nachrichten				
	0,001%	*0,01%*	*0,1%*	*1,0%*	*10%*
Filter 1	4	33	358	3.345	33.464
Filter 2	3	21	224	2.325	23.610
Filter 3	1	22	164	1.847	17.946
Filter 4	1	3	26	356	3.442
Filter 5	1	-	7	71	751
Filter 6	-	12	111	1.096	11.477
Filter 7	-	4	35	342	3.136
Filter 8	-	1	20	213	2.035
Filter 9	-	3	24	174	1.695
Filter 10	-	-	19	131	1.452
Filter 11	-	1	10	98	988
Filter 12	-	-	1	1	3
Filter 13	-	-	1	-	1

Tabelle 4.14: Die Anzahl der E-Mail-Nachrichten, auf denen die für den SRIZBI-Spamdatensatz generierten Filter basieren.

Die Anzahl der im Durchschnitt pro Filter gelernten Wörter und Sätze ist in Tabelle 4.15 dargestellt. Für die ersten elf Filter werden durchschnittlich je 72,15 Wörter und 18,72 Sätze gelernt. Die für eine Kampagne charakteristischen Wörter und Sätze werden meist schon sehr früh, nachdem nur wenige Nachrichten analysiert wurden, gelernt. Dies erklärt die sehr guten Filterraten und die nahezu identischen Filterergebnisse der einzelnen Filter innerhalb der Kampagnen. Die niedrige Anzahl an Daten pro Kampagne erklärt zudem, wieso der kontextfreie Filteransatz auf dem SRIZBI-Datensatz so gute Ergebnisse liefert: Innerhalb der einzelnen Kampagnen werden jeweils nur sehr wenige Wörter und Sätze verwendet. Dieses Ergebnis bestätigt die Annahme auf der der in dieser Arbeit präsentierte kontextfreie Filteransatz basiert: Jeder Spam-Kampagne liegt nur eine begrenzte Anzahl an Wörtern und Formulierungen zugrunde.

Lerndatensatz	**Anzahl der gelernten Wörter und Sätze**				
	0,001%	*0,01%*	*0,1%*	*1,0%*	*10%*
Filter 1	67 / 16	70 / 20	70 / 20	70 / 20	70 / 20
Filter 2	73 / 16	78 / 21	78 / 21	78 / 21	78 / 21
Filter 3	39 / 11	67 / 31	70 / 33	70 / 33	70 / 33
Filter 4	73 / 11	77 / 13	84 / 18	84 / 18	84 / 18
Filter 5	67 / 14	-	71 / 18	75 / 21	75 / 21
Filter 6	-	72 / 19	73 / 20	73 / 20	73 / 20
Filter 7	-	72 / 14	77 / 19	77 / 19	77 / 19
Filter 8	-	69 / 11	71 / 13	71 / 13	71 / 13
Filter 9	-	71 / 13	72 / 14	72 / 14	70 / 14
Filter 10	-	-	73 / 17	73 / 17	73 / 17
Filter 11	-	63 / 19	72 / 22	71 / 22	72 / 22
Filter 12	-	-	35 / 76	42 / 13	44 / 46
Filter 13	-	-	38 / 108	-	35 / 137

Tabelle 4.15: Die Anzahl der gelernten Wörter und Sätze für den im SRIZBI-Botnetz generierten Spam-Lauf, jeweils in Abhängigkeit von der Anzahl zur Filtergenerierung verwendeter E-Mail-Nachrichten.

4.7 Zusammenfassung und Ausblick

Zusammenfassung

Die in diesem Kapitel vorgestellten Filteransätze für Spam-Nachrichten versuchen, die bei musterbasierten Spam-Kampagnen existierenden Regelmäßigkeiten in den Nachrichten zur Konstruktion von Spam-Filtern zu verwenden. Dazu werden Spambots in einer Analyseumgebung ausgeführt, in der alle von diesen Bots zur Laufzeit verschickten Nachrichten auf einen lokalen E-Mail-Server umgeleitet werden. Durch den Vergleich der abgefangenen Spam-Nachrichten miteinander wird versucht, die bei den Spam-Kampagnen verwendeten Muster aus den Nachrichten zu extrahieren und einen auf alle Nachrichten der Kampagne passenden Filter zu konstruieren. Der Filter kann anschließend in E-Mail-Clients zum Filtern der Kampagne verwendet werden. Da bei der Konstruktion der Filter sichergestellt ist, dass diese nicht zu generisch werden, besitzen die Filter zudem eine minimale *false-positive*-Rate.

Für das Extrahieren des Spam-Musters wurde ein auf regulären Ausdrücken und ein auf LEX/YACC basierender Filter vorgestellt. Ersterer versucht, das Spam-Muster für die gesamte Nachricht in einem Ausdruck darzustellen und stößt zum Beispiel bei Permutationen innerhalb der Nachrichten an seine Grenzen. Bei dem auf LEX/YACC basierenden Filter wird eine kontextfreie Grammatik für E-Mail-Nachrichten zugrunde gelegt, und der Filter lernt die in der Nachricht enthaltenen Wörter, die daraus zusammengesetzten Sätze und die E-Mail-Adressen, URLs und Dateianhänge. Das Spam-Muster wird schließlich durch eine (weiche) Grammatik beschrieben, und die Zugehörigkeit einer Nachricht zu der jeweiligen Spam-Kampagne lässt sich über variable Schranken bestimmen.

Beide Filteransätze wurden in Prototypen umgesetzt und auf den Spam-Daten verschiedener Botnetze evaluiert. Die Filteransätze benötigen nur einen Bruchteil der tatsächlich gesendeten Nachrichten, um sehr gute Filter zu berechnen. Dies gilt insbesondere für die auf einer kontextfreien Grammatik basierenden Spam-Filter. Selbst aus nur einer Nachricht generierte Filter liefern hier teilweise schon Erkennungsraten von über 99 oder auch 100 Prozent.

Ausblick

Die einfachen, auf regulären Ausdrücken basierenden Filter können als nahezu „ausgereizt“ angesehen werden. Erweiterungen, wie das Zulassen von „Oder“-Verknüpfungen für Wörter, Sätze oder ganze Teile der Nachrichten sind denkbar, liefern aber keinen den Aufwand rechtfertigenden Mehrwert für den Filter. Letztlich versuchen die Ansätze lediglich, die durch die regulären Ausdrücke bedingten Grenzen zu erweitern, ohne den Grund für diese Grenzen – die (freiwillige) Beschränkung auf eine reguläre Sprache – zu beseitigen. Gerade dieser Schritt wird bei dem zweiten Filteransatz vollzogen.

Durch die Umstellung auf eine kontextfreie Grammatik lassen sich Probleme, wie Permutationen von Wörtern oder auch Sätzen, lösen, ohne dabei Wissen über die Spam-Kampagne zu verlieren. Auch wenn der in dieser Arbeit vorgestellte Ansatz sehr gute Ergebnisse liefert, reizt die Grammatik die sich bietenden Möglichkeiten noch

lange nicht aus. Beispielsweise ließen sich über entsprechende Token und Grammatikregeln sprach-spezifische Grammatiken erzeugen, die die in einer E-Mail-Nachricht enthaltenen Sätze nicht nur als eine Aneinanderreihung von gleichartigen Token definiert, sondern auch den Aufbau der Sätze beachtet. Dadurch könnte der Inhalt der Kampagne noch besser erfasst und die erlaubte Varianz genauer eingestellt werden.

KAPITEL V

Zusammenfassung und Ausblick

5.1 Zusammenfassung

In der vorliegenden Arbeit wurden neue Verfahren zum Filtern von Schadprogrammen und Spam-Nachrichten vorgestellt. Sowohl für das zum Filtern der Schadprogramme herangezogene Programmverhalten, als auch für die Spam-Nachrichten wurden kontextfreie Grammatiken entwickelt, die sich zu spezifischen Filtern für Programmfamilien beziehungsweise Spam-Kampagnen erweitern lassen. Die Evaluierung der Schadprogramm- und Spam-Filter erfolgte an realen Datensätzen und lieferte für beide Ansätze sehr gute Ergebnisse.

5.1.1 Musterbasierte Filter für Schadprogramme

Mit Hilfe der musterbasierten Filter lässt sich die dynamische Programmanalyse für bekannte Programmfamilien vollständig automatisieren. Die Filter übernehmen dabei den einzigen bisher noch nicht zu automatisierenden Teil der Verhaltensbewertung. Analog zu den Signaturen statischer Virenscanner, stellen die Filter Verhaltenssignaturen für Programmfamilien dar, mit denen unbekannte Programme auf eine Familienzugehörigkeit hin untersucht werden können.

Die auf einer kontextfreien Grammatik basierenden Verhaltensfilter sind eine konsequente Weiterentwicklung der speziell zur Verhaltensrepräsentation entwickelten Sprache MIST. Das *Malware Instruction Set* ist eine Metasprache für Verhaltensreporte unterschiedlicher Analyseumgebungen und macht diese untereinander vergleichbar. Gleichzeitig optimiert MIST die Darstellung des aufgezeichneten Verhaltens im Hinblick auf eine automatisierte Analyse der Verhaltensreporte und macht so den Einsatz aktueller Techniken des Data Mining und des maschinellen Lernens auf den Reporten möglich. Dazu werden die in den Verhaltensreporten enthaltenen Informationen auf die zentralen Details verdichtet, indem zufällige Informationen, wie beispielsweise Zeitstempel oder Prozess-IDs, entfernt und zur Beschreibung des Verhaltens notwendige Informationen so weit wie möglich normiert werden. Dadurch werden sehr variable oder nur von der Analyseumgebung abhängende Informationen, zum Beispiel

Rechner- und Nutzernamen, aus den Verhaltensreporten entfernt. Erst auf diesen normalisierten Verhaltensreporten lässt sich das Verhalten unterschiedlicher Programme miteinander vergleichen. Diese Vergleichbarkeit ist die Grundvoraussetzung, um beispielsweise durch eine Clusteranalyse der Verhaltensreporte Programmfamilien allein anhand des aufgezeichneten Verhaltens zu identifizieren. Für gefundene Programmfamilien lassen sich anschließend deren charakteristische Verhaltensmuster berechnen, die wiederum als Grundlage für musterbasierte Filter fungieren. Die mit dem Einsatz von MIST einhergehende Effizienzsteigerung der Clusteranalyse und Klassifikation von Verhaltensreporten wurde in den Abschnitten 3.2.3 und 3.3 im Detail dargestellt. Neben der bedeutend besseren Verhaltensrepräsentation bringt der Einsatz von MIST zusätzlich eine erhebliche Datenreduktion mit sich. Diese wird durch eine vollständig neue Kodierung aller Informationen erreicht und wirkt sich positiv auf den Speicherbedarf und die Laufzeit der Algorithmen des Data Mining und maschinellen Lernens aus. Dadurch ist es möglich, die in der Regel sehr ressourcenintensiven Algorithmen auch auf sehr großen Datensätzen anzuwenden. Dieses wurde sowohl in wissenschaftlichen Beiträgen, als auch im Produktivbetrieb nachgewiesen. Die auf MIST-codierten Verhaltensreporten basierende Clusteranalyse und Klassifikation wurde für insgesamt 14 Monate im MWAnalysis-Projekt erfolgreich eingesetzt, um die von den verschiedenen Sensoren gesammelten Schadprogramme in Familien zu klassifizieren und neue Familien in den Daten zu identifizieren.

Das *Malware Instruction Set* ist eine kontextfreie Sprache, mit der sich Programmverhalten beschreiben lässt. Durch die Angabe bestimmter Verhaltensmuster in Form von Instruktionsfolgen, lassen sich in der Sprache Filter für die sich dahinter verbergenden Programmfamilien definieren. Damit können Programme, allein auf Basis ihres Verhaltens, auf die Zugehörigkeit zu einer Programmfamilie hin untersuchen werden. Der zur Evaluierung des Filteransatzes implementierte Prototyp basiert auf Lex und YACC. Ein Filter stellt dabei ein von der Grundgrammatik des MIST-Reports abgeleitetes Paar aus Lexer und Parser dar. Um die Reporte auf die Familien-spezifischen Verhaltensmuster hin überprüfen zu können, wird der Lexer um Token für die gesuchten Instruktionen und der Parser um die Instruktionsfolgen abdeckende Grammatikregeln erweitert. Zur Berechnung der Filter müssen dem Filtergenerator somit nur die gesuchten Instruktionsfolgen übergeben werden. Die Evaluierung der Verhaltensfilter erfolgte auf einem Datensatz mit über 3.000 Verhaltensreporten aus 24 verschiedenen Schadprogrammfamilien. Abhängig von der Güte der zur Filtergenerierung verwendeten Verhaltensmuster liefern die Filter hierbei gute bis sehr gute Ergebnisse.

5.1.2 Musterbasierte Filter für Spam-Nachrichten

Der überwiegende Anteil der Spam-Nachrichten wird heute über Botnetze verteilt, die musterbasierte Spam-Verfahren einsetzen. Dabei werden die Spam-Nachrichten auf Basis einer Vorlage und verschiedener Fülltexte durch die Bots selbst zusammengesetzt. Durch diese Spam-Verfahren variieren die innerhalb einer Kampagne verschickten E-Mail-Nachrichten leicht, was in klassischen Spam-Filtern nur sehr schwer abgefangen werden kann. Die in dieser Arbeit präsentierten Filteransätze sind speziell auf diese musterbasierten Spam-Verfahren abgestimmt und nutzen die in jeder Kampagne vorhandenen Regelmäßigkeiten (Vorlagen), um passende Filter zu berechnen.

Um möglichst frühzeitig Filter für neue Kampagnen erstellen und verteilen zu können, wurde zudem eine Analyseumgebung aufgebaut, in der die Bots der relevanten Botnetze kontrolliert ausgeführt werden können. Sobald die Bots mit dem Versenden einer neuen Kampagne beginnen, werden die Nachrichten lokal umgeleitet und zur Erstellung neuer Filter verwendet.

Die entwickelten Filtermethoden basieren auf regulären Ausdrücken und einer eigens entwickelten kontextfreien Grammatik für E-Mail-Nachrichten. Die erste Methode beschreibt das Muster hinter einer Spam-Kampagne über jeweils einen regulären Ausdruck. Dieser Ausdruck enthält die von allen Nachrichten geteilten Textblöcke. Nur diese Textblöcke können auch in der Kampagnen-Vorlage fest sein und bilden damit das Grundgerüst für alle E-Mail-Nachrichten der Spam-Kampagne. Die übrigen Textblöcke sind variable Bestandteile der Kampagne, das heißt, für diese Stellen existieren verschiedene Fülltexte, die von dem Bot selbstständig eingesetzt werden. Sie sind damit nicht Teil der Spam-Vorlage und werden in dem Filter durch Platzhalter ersetzt, die die ersetzten Texte möglichst genau spezifizieren. Die Kombination aus festen und variablen Teilen stellt am Ende den regulären Ausdruck dar, der als Spam-Filter verwendet werden kann. Das Konzept wurde in einem Prototypen implementiert und auf mehreren Spam-Kampagnen verschiedener Botnetze evaluiert. Die Erkennungsrate der automatisch generierten Filter ist auf allen betrachteten Datensätze sehr gut. Zudem müssen in der Regel nur wenige Nachrichten analysiert werden, um Filter zu generieren, die 90 Prozent oder mehr Nachrichten einer Kampagne erkennen.

Spam-Nachrichten bewerben Produkte und sollen als solche das Interesse des Lesers für das Produkt wecken. Dabei stehen den Spammern nur endlich viele Wörter und Sätze zur Verfügung, um sinnvolle und verständliche Nachrichten zu formulieren. Diese Notwendigkeit macht sich der zweite, auf einer kontextfreien Grammatik basierende Filteransatz zu nutze. Statt der ganzen Nachricht, inklusive möglicher Formatierungsinformationen, werden nur die tatsächlich für einen Anwender sichtbaren Wörter und die sich aus diesen zusammensetzenden Sätze gelernt. Wie sich die einzelnen E-Mail-Nachrichten der Kampagne letztlich aus diesen Daten zusammensetzen, wird hier nicht berücksichtigt. Damit ist dieser Ansatz robust gegenüber Änderungen innerhalb einer Kampagne und erkennt auch Nachrichten, in denen Wörter oder ganze Sätze vertauscht oder ausgewechselt wurden. In der Evaluierung dieses Filteransatzes konnte gezeigt werden, dass diese Filter schon sehr früh sehr gute Erkennungsraten liefern. Für mehrere hundert analysierte Kampagnen des STORM-Botnetzes war nur eine einzige Nachricht notwendig, um einen Filter mit 99,75-prozentiger Erkennungsrate innerhalb der Kampagne zu generieren. Dies ist nur möglich, weil der neue Filteransatz die tatsächliche Varianz innerhalb der Kampagne nicht mehr kennen muss. Stattdessen passt sich der Filter beim analysieren potentieller Nachrichten selbstständig an und bestimmt die Zugehörigkeit der E-Mail-Nachrichten zu einer Kampagne über den Grad der notwendigen Anpassung.

Der implementierte Prototyp basiert auf LEX und YACC und setzt auf der eigens entworfenen Grundgrammatik für E-Mail-Nachrichten auf. Während der Filtergenerierung und beim Filtern unbekannter Nachrichten passen sich der Lexer und der Parser selbstständig an, indem sie die bekannten Token und Grammatikregeln um die neuen Wörter und Sätze ergänzen. Ein Eingreifen des Anwenders ist hier zu keinem Zeitpunkt erforderlich. Insgesamt kann dieser Filteransatz die bereits sehr guten Ergeb-

nisse des auf regulären Ausdrücken basierenden Filteransatzes noch mal übertreffen und mit minimalem Wissen (eine Spam-Nachricht pro Kampagne) eine fast perfekte Erkennungsrate liefern.

5.2 Ausblick

In diesem Abschnitt werden noch einige Ideen angesprochen, wie sich die in der Arbeit präsentierten Methoden und Prototypen zum Filtern von Schadprogrammen und Spam weiterentwickeln lassen. Dabei werden sowohl aktuelle Schwächen in der Implementierung der Prototypen, als auch funktionale Erweiterungen berücksichtigt.

5.2.1 Optimierung und Anpassung der Implementierung

Sämtliche Prototypen sind in PYTHON implementiert. Als Skriptsprache ist PYTHON sehr gut geeignet, um Ideen schnell in Prototypen umzusetzen und damit evaluieren zu können. Die Vielzahl an existierenden Modulen, beispielsweise die Module TEMPLATEMAKER und PLY, bilden dabei eine gute Grundlage. Leider besitzen die existierenden Module für komplexere Operationen schlechte Laufzeiten und verfügen teilweise auch nicht über alle benötigten Schnittstellen und Methoden. Beispielsweise musste der Quellcode von PLY angepasst werden, um in der zum Lernen der Nachrichten notwendigen Fehlerbehandlungsroutine die benötigten Informationen über den aktuellen Zustand des Parsers auslesen zu können. Auch die Laufzeit der als Spam-Filter fungierenden Parser ist schlecht, was allerdings erst bei der Analyse von mehreren Millionen Nachrichten negativ auffällt. Vor einem Produktiveinsatz des auf einer kontextfreien Grammatik basierenden Spam-Filters sollte dieser daher in eine Hochsprache überführt werden.

Das *Malware Instruction Set* ist aktuell der einzige frei verfügbare Ansatz, mit dem sich die Lücke zwischen der Verhaltensanalyse und den Methoden des maschinellen Lernens schließen lässt. Da die in MIST definierten Instruktionen momentan aber noch eins zu eins die von der CWSANDBOX überwachten Systemaufrufe widerspiegeln, konnte die Transformationstabelle bisher noch nicht veröffentlicht werden. Mit der Anbindung weiterer Analysesysteme, wie zum Beispiel ANUBIS, JOEBOX oder PYBOX, wird diese eindeutige Zurechenbarkeit der Instruktionen verloren gehen. Die dann mögliche Veröffentlichung der universellen Transformationstabelle wird die Chance zusätzlich erhöhen MIST als Metasprache für Verhaltensreporte zu etablieren.

5.2.2 Programmanalyse auf Prozess- und Thread-Ebene

Aktuell werden die Verhaltensreporte immer vollständig in MIST übersetzt und zur Clusteranalyse oder zum Filtern verwendet. Da einzelnen Funktionen – beispielsweise das Infizieren des Systems, die Kommunikation mit dem *C&C*-Server, das Nachladen von Schadcode, oder das Versenden von Spam-Nachrichten – in der Regel auch in eigens dafür aufgesetzten Prozessen und Threads durchgeführt werden, ist es sinnvoll, auch die Analyse auf Prozess- oder Thread-Ebene durchzuführen. Eine entsprechende (Cluster-)Analyse könnte eine Zuordnung von Prozessen oder Threads zu einzel-

nen Funktionen liefern. Mit diesem Wissen ließen sich die Verhaltensreporte in leicht verständliche *high-level* Reporte übersetzen. Darauf basierend könnten zudem Querverbindungen zwischen Schadprogrammen oder auch Programmfamilien identifiziert werden, die die Code-Verwandtschaften zwischen diesen offenlegen.

5.2.3 *Multipath*-Unterstützung für MIST

Fast alle Sandbox-Systeme betrachten bei der Programmanalyse nur einen Ausführungspfad. Diese Beschränkung wurde auch für MIST übernommen. Damit bei der Anbindung von Sandbox-Systemen, die eine *multipath* Analyse unterstützen, kein Informationsverlust entsteht, sollte MIST ebenfalls mehrere Ausführungspfade unterstützen. Durch die Umstellung werden die MIST-Reporte eine Baumstruktur erhalten, in der die verschiedenen Ausführungspfade jeweils in eigenen Ästen codiert sind. Allerdings ist mit der Umstellung ein sehr hoher Implementierungsaufwand verbunden, da sich sämtliche Anpassungen unmittelbar auf das Analysewerkzeug MALHEUR und die Filter auswirken. Durch die *multipath*-Unterstützung könnten nicht nur entsprechende Analysesysteme angebunden werden, sondern auch verschiedene Ausführungen eines Programms in einem nicht *multipath*-fähigen Analysesystem zu einem Report mit mehreren Ausführungspfaden verschmolzen werden. Auch wenn ein auf diese Weise erstellter Verhaltensreport nicht alle möglichen Ausführungspfade enthalten kann, würden damit zumindest die tatsächlich beobachteten Pfade abgedeckt. Damit könnte MIST jedes beliebige Sandbox-System nachträglich um eine eingeschränkte *multipath*-Unterstützung erweitern.

5.2.4 Erweiterte E-Mail-Grammatik

Mit Hilfe der vorgestellten kontextfreien Grammatik lassen sich schon jetzt sehr gute Spam-Filter konstruieren. Die zur Beschreibung von E-Mail-Nachrichten entworfene Grammatik arbeitet momentan nur mit Wörtern und Sätzen und ist deshalb noch sehr oberflächlich. Mit einer die Sprache selbst besser erfassenden Grammatik, beispielsweise durch die Prüfung der Wortstellung, ließen sich E-Mail-Nachrichten und damit auch Spam-Kampagnen deutlich besser beschreiben und voneinander abgrenzen. Durch spezielle Token für die einzelnen Wortarten würden Sätze nicht mehr nur als eine Aneinanderreihung von gleichartigen Token aufgefasst. Ob eine solche, umfassende Erweiterungen der Grammatik bei den aktuellen Filterergebnissen gerechtfertigt ist, scheint fraglich. Trotzdem sollten die Gegenmaßnahmen kontinuierlich verbessert werden, da davon auszugehen ist, dass sich auch die musterbasierten Spam-Verfahren weiterentwickeln werden.

Literaturverzeichnis

E. Alpaydin. *Introduction to Machine Learning*. 2004.

M. Andreolini, A. Bulgarelli, M. Colajanni, and F. Mazzoni. HoneySpam: Honeypots Fighting Spam at the Source. In *Proceedings of the SRUTI'05*, 2005.

I. Androutsopoulos, J. Koutsias, K. V. Chandrinos, G. Paliouras, and C. D. Spyropoulos. An Evaluation of Naive Bayesian Anti-Spam Filtering. In *Workshop on Machine Learning in the New Information Age*, 2000.

M. Apel, C. Bockermann, and M. Meier. Measuring Similarity of Malware Behavior. In *Proceedings of the 5th LCN Workshop on Security in Communications Networks*, pages 891–898, 2009.

M. Apel, J. Biskup, U. Flegel, and M. Meier. Early Warning System on a National Level. In *1st European Workshop on Internet Early Warning and Network Intelligence (EWNI)*, 2010.

E. Aridogan. Aktuelle SpamBOTs (Studienarbeit), 2010.

P. Bächer, M. Kötter, T. Holz, F. Freiling, and M. Dornseif. The Nepenthes Platform: An Efficient Approach to Collect Malware. In *Proceedings of Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 165–184, 2006.

J. W. Backus, F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. Revised report on the algorithmic language ALGOL 60. *The Computer Journal*, 5(4):349–367, 1963.

M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated Classification and Analysis of Internet Malware. In *Proceedings of Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 178–197, 2007.

M. T. Banday, J. A. Qadri, and N. A. Shah. Study of Botnets and Their Threats to Internet Security, 2009. URL: *http://sprouts.aisnet.org/9-24*.

U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *Proceedings of Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, April 2006a.

U. Bayer, A. Moser, C. Kruegel, and E. Kirda. Dynamic Analysis of Malicious Code. *Journal in Computer Virology*, 2(1):67–77, 2006b.

U. Bayer, P. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Proceedings of Symposium on Network and Distributed System Security (NDSS)*, 2009.

D. M. Beazley. PLY (Python Lex-Yacc), 2009. URL: *http://www.dabeaz.com/ply/ply.hmtl*.

L. Böhne. Peeking into Pandora's Bochs - Instrumenting a Full System Emulator to Analyse Malicious Software. In *Hackito Ergo Sum 2010*, 2010.

N. Chomsky. Three models for the description of language. In *Proceedings of IRE Transactions on Information Theory*, pages 113–124, 1956.

M. Christodorescu and S. Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of USENIX Security Symposium*, pages 12–12, 2003.

M. Christodorescu, S. Jha, S. A. Seshia, D. X. Song, and R. E. Bryant. Semantics-Aware Malware Detection. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 32–46, 2005.

R. Der. Vorlesung Digitale Informationsverarbeitung, 2001. URL: *http://www.informatik.uni-leipzig.de/~der/Vorlesungen/DIV/divfolien.html*.

A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of Conference on Computer and Communications Security (CCS)*, pages 51–62, 2008.

Dr. Kaiser Systemhaus. PC-Wächter. URL: *http://www.dr-kaiser.de/*.

H. Drucker, D. Wu, and V. Vapnik. Support vector machines for spam categorization. *IEEE Transactions on Neural Networks*, 10(5):1048–1054, 1999.

M. Engelberth, F. Freiling, J. Göbel, C. Gorecki, T. Holz, P. Trinius, and C. Willems. InMAS Documentation: Reliable Sandbox Setup, 2008.

M. Engelberth, F. Freiling, J. Göbel, C. Gorecki, T. Holz, P. Trinius, and C. Willems. Frühe Warnung durch Beobachten und Verfolgen von bösartiger Software im Deutschen Internet: Das Internet-Malware-Analyse System (InMAS). In *11. Deutscher IT-Sicherheitskongress*, 2010a.

M. Engelberth, F. C. Freiling, J. Göbel, C. Gorecki, T. Holz, R. Hund, P. Trinius, and C. Willems. The InMAS Approach. In *1st European Workshop on Internet Early Warning and Network Intelligence (EWNI)*, 2010b.

M. Engelberth, F. Freiling, J. Göbel, C. Gorecki, T. Holz, R. Hund, P. Trinius, and C. Willems. Das Internet-Malware-Analyse-System (InMAS). *Datenschutz und Datensicherheit - DuD*, (1), 2011.

M. Ester and J. Sander. *Knowledge Discovery in Databases: Techniken und Anwendungen*. Springer, 2000.

N. Falliere, L. O. Murchu, and E. Chien. W32.Stuxnet Dossier, Version 1.4, Februar 2011. URL: *http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf*.

Faronics. Deep Freeze - Absolute System Integrity, 2010. URL: *http://www.faronics.com/en/Products/DeepFreeze/DeepFreezeCorporate.aspx*.

P. Ferrie. Anti-Unpacker Tricks 2 Part One, Dezember 2008.

P. Ferrie. Anti-Unpacker Tricks 2 Part Seven, Juni 2009.

S. Gauthronet and E. Drouard. Unsolicited Commercial Communications and Data Protection, April 2008. URL: *http://www.rigacci.org/docs/biblio/online/spam_garante/document/434683.pdf*.

A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th VLDB Conference*, 1999.

J. Govil. Examining the Criminology of Bot Zoo. In *Proceedings of the 6th International Conference on Information, Communications Signal Processing*, Singapore, Dezember 2007.

M. Gräßlin. Implementation of ProactiveSpam Fighting Techniques, 2010. URL: *https://pi1.informatik.uni-mannheim.de/filepool/theses/masterthesis-2010-graesslin.pdf*.

J. Göbel. Amun: Automatic Capturing of Malicious Software. In *Proceedings of 5th GI Conference "Sicherheit, Schutz und Zuverlässigkeit"*, 2010.

J. Göbel and P. Trinius. Towards Optimal Sensor Placement Strategies for Early Warning Systems. In *Proceedings of 5th GI Conference "Sicherheit, Schutz und Zuverlässigkeit"*, 2010.

J. Göbel, T. Holz, and P. Trinius. Towards Proactive Spam Filtering. In *Proceedings of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 38–48, 2009.

P. Hay. Spam Volumes Drop After Spamit Shakeup, October 2010. URL: *http://labs.m86security.com/2010/10/spam-volumes-drop-after-spamit-shakeup/*.

S. Hoernlimann and T. Meyer. Classical Machine Learning. In *Seminar on Embodied Models of Learning, Development and Memory*. University of Zurich, 2005.

A. Holovaty. Python library for extracting data from similarly formatted text strings., 2007. URL: *http://code.google.com/p/templatemaker/*.

T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling. Measurements and Mitigation of Peer-to-Peer-based Botnets: A Case Study on Storm Worm. In *Proceedings of LEET'08*, April 2008.

Honeynet Project. Know Your Enemy Lite: Proxy Threats – Port v666, 2008. URL: *http://honeynet.org/papers/proxy/index.html*.

J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy. Studying Spamming Botnets Using Botlab. In *Proceedings of NSDI'09*, 2009.

J. Jung and E. Sit. An Empirical Study of Spam Traffic and the Use of DNS Black Lists. In *Proceedings of the 4th ACM Conference on Internet Measurement*, 2004.

KDE-PIM. Akonadi - The PIM Storage Service, 2010. URL: *http://pim.kde.org/akonadi/*.

J. Kirk. Virus Encrypts Data, Demands Ransom, March 2006.

C. Kolbitsch, P. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and Efficient Malware Detection at the End Host. In *Proceedings of USENIX Security Symposium*, 2009.

J. Kolter and M. Maloof. Learning to Detect and Classify Malicious Executables in the Wild. *Journal of Machine Learning Research*, 8(Dec):2755–2790, 2006.

C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. On the Spam Campaign Trail. In *Proceedings of LEET'08*, 2008.

K. Lab. *Trojaner-Programme*. 2005. URL: *http://www.viruslist.com/de/viruses/encyclopedia?chapter=152540521*.

T. Lee and J. J. Mody. Behavioral Classification. In *Proceedings of Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*, April 2006.

M. Lesk, M. R. Stytz, and R. L. Trope. The New Front Line: Estonia under Cyberassault. In *IEEE Security & Privacy*, 2007.

C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of Conference on Computer and Communications Security (CCS)*, 2003.

R. W. Lo, K. N. Levitt, and R. A. Olsson. MCF: a malicious code filter. *Computers & Security*, 14(6):541–566, 1995.

M86 Security. Tracking Spam Botnets, December 2010a. URL: *http://www.m86security.com/labs/bot_statistics.asp*.

M86 Security. Security Labs Report: January - June 2010 Recap, July 2010b. URL: *http://www.m86security.com/documents/pdfs/security_labs/m86_security_labs_report_1H2010.pdf*.

M86 Security. Spam Statistics - Spam Sources by Country, December 2010c. URL: *http://www.m86security.com/labs/spam_statistics.asp*.

L. Martignoni, M. Christodeorescu, and S. Jha. OmniUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of Anual Computer Security Application Conference (ACSAC)*, pages 431–441, 2007.

I. I. McAfee. The Carbon Footprint of Email Spam Report, 2009. URL: *http://resources.mcafee.com/content/NACarbonFootprintSpam*.

A. Meinecke. Eigenschaften der Spambots mit dem meisten (Studienarbeit), 2011.

Microsoft. Windows Sysinternals., July 2006. URL: *http://technet.microsoft.com/de-de/sysinternals/*.

Microsoft. Microsoft Security Intelligence Report (SIR). Volume 7 (January – June 2009), Microsoft Corporation, 2009.

Microsoft. Malware – Infection Rates, 2010a. URL: *http://www.microsoft.com/security/sir/guide/default.aspx#section_5_2*.

Microsoft. Microsoft Security Intelligence Report (SIR). Volume 9, Microsoft Corporation, 2010b.

T. Mitchell. *Machine Learning*. Mcgraw-Hill International Edit, 1997.

D. Moore, C. Shannon, and J. Brown. Code-Red: a case study on the spread and victims of an internet worm. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, 2002.

A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Proceedings of Anual Computer Security Application Conference (ACSAC)*, 2007a.

A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of IEEE Symposium on Security and Privacy*, 2007b.

A. Mushtaq. Silent Rustock, October 2010. URL: *http://blog.fireeye.com/research/2010/10/silent-rustock.html*.

National Office for the Information Economy (NOIE). Final Report of the NOIE Review of the Spam Problem and How it can be countered, 2003. URL:

http://www.apcomms.org.uk/apig/archive/activities-2003/spam-public-enquiry/written-evidence-submitted-to-the-enquiry/australian_noie_appendix_spamreport.pdf.

G. Neale. Template Based Spam., May 2009. URL: *http://www.m86security.com/labs/i/Template-Based-Spam,trace.996~.asp*.

Norman. Norman Sandbox - Whitepaper., 2003. URL: *http://download.norman.no/whitepapers/whitepaper_Norman_SandBox.pdf*.

G. Ollman. Botnet Communication Topologies: Understanding the intricacies of botnet Command-and-Control, 2010. URL: *http://www.damballa.com/downloads/r_pubs/WP_Botnet_Communications_Primer.pdf*.

A. Pathak, Y. C. Hu, and Z. M. Mao. Peeking into Spammer Behavior from a Unique Vantage Point. In *Proceedings of LEET'08*, 2008.

R. Perdisci, A. Lanzi, and W. Lee. McBoost: Boosting Scalability in Malware Collection and Analysis Using Statistical Classification of Executables. In *Proceedings of Anual Computer Security Application Conference (ACSAC)*, pages 301–310, 2008.

A. Pitsillidis, K. Levchenko, C. Kreibich, C. Kanich, G. Voelker, V. Paxson, N. Weaver, and S. Savage. Botnet Judo: Fighting Spam with Itself. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, USA, March 2010.

R. Pointer. *Eggheads.org*. 2006. URL: *http://www.eggheads.org/*.

I. V. Popov, S. K. Debray, and G. R. Andrews. Binary Obfuscation Using Signals. In *Proceedings of USENIX Security Symposium*, pages 275–290, 2007.

M. D. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. In *Proceedings of 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

M. D. Preda, M. Christodorescu, S. Jha, and S. Debray. A semantics-based approach to malware detection. *ACM Trans. Program. Lang. Syst.*, 30(5), 2008.

N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost in the Browser Analysis of Web-based Malware. In *Proceedings of HotBots'07*, 2007.

K. Rieck. MALHEUR - Automatic Analysis of Malware Behavior, 2010. URL: *http://www.mlsec.org/malheur/*.

K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and Classification of Malware Behavior. In *Proceedings of Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 108–125, 2008.

K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic Analysis of Malware Behavior using Machine Learning. Technical Report Technical Report 18-2009, Berlin Institute of Technology, 2009.

K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic Analysis of Malware Behavior using Machine Learning. *Journal of Computer Security*, 19(4), 2011.

P. Royal, M. Halpin, D. Dagon, R. Edmonds, and W. Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. In *Proceedings of Anual Computer Security Application Conference (ACSAC)*, pages 289–300, 2006.

P. Rubin, D. MacKenzie, and S. Kemp. dd - convert and copy a file. URL: *http://linuxmanpages.com/man1/dd.1.php*.

F. Rötzer. *DoS-Angriffe auf Internetseiten der estnischen Regierung*. heise online, 2007. URL: *http://www.heise.de/tp/artikel/25/25218/1.html*.

M. Sahami, S. Dumais, D. Heckerman, and E. Horvitz. A Bayesian Approach to Filtering Junk E-Mail. In *Learning for Text Categorization*. AAAI Technical Report WS-98-05, 1998.

G. Schryen. *Anti-Spam Measures – Analysis and Design*. Springer, first edition, 2007.

M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data Mining Methods for Detection of New Malicious Executables. In *Proceedings of IEEE Symposium on Security and Privacy*, 2001.

U. Schöning. *Theoretische Informatik - kurz gefasst*. Spektrum, 2009.

Secunia. Secunia Half Year Report 2010, 2010. URL: *http://secunia.com/gfx/pdf/Secunia_Half_Year_Report_2010.pdf*.

M. Sharif, A. Lanzi, G. Jonathon, and W. Lee. Impeding Malware Analysis Under Conditional Code Obfuscation. In *Proceedings of Symposium on Network and Distributed System Security (NDSS)*, 2008.

M. Sharif, A. Lanzi, J. Giffin, and W. Lee. Automatic Reverse Engineering of Malware Emulators. In *Proceedings of IEEE Symposium on Security and Privacy*, 2009.

H. Sistemas. Virus Total. URL: *http://www.virustotal.com*.

D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. BitBlaze: A New Approach to Computer Security via Binary Analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, 2008.

L. Spitzner. *Honeypots, Tracking Hackers*. Addison-Wesley Longman, 2002.

H. Stern. A Survey of Modern Spam Tools. In *Proceedings of the 5th Confernece in Email and Anti-Spam*, 2008.

J. Stewart. Top Spam Botnets Exposed, April 2008. URL: *http://secureworks.com/research/threats/topbotnets/*.

J. Stewart. Spam Botnets to Watch in 2009, January 2009. URL: *http://secureworks.com/research/threats/botnets2009/*.

J. Stewart. Cracking Down on Botnets, February 2011. URL: *http://blogs.technet.com/b/microsoft_on_the_issues/archive/2010/02/24/cracking-down-on-botnets.aspx*.

S. J. Stolfo, K. Wang, and W.-J. Li. *Towards Stealthy Malware Detection*, volume 27 of *Advances in Information Security*, pages 231–249. Springer US, 2007.

Symantec. Internet security threat report. Volume XIV (January – December 2008), Symantec Corporation, 2009.

Symantec. Sate of Spam & Phishing - A Monthly Report., April 2010a. URL: *http://www.symantec.com/content/de/de/about/downloads/PressCenter/SpamPhishingReport_April2010.pdf*.

Symantec. Symantec Global Internet Security Threat Report. Volume XV – April 2010, Symantec Corporation, 2010b.

P. Szor. *The art of computer virus research and defense*. Symantec Press, 2005.

P.-N. Tan, M. Steinbach, and V. Kumar. *Introduction to Data Mining*. Addison Wesley, 2005.

The Spamhaus Project. The Definition of Spam, 2010. URL: *http://www.spamhaus.org/definition.html*.

ThreadExpert. Automated Threat Analysis, 2010. URL: *http://www.threatexpert.com/*.

P. Trinius. Omnivora: Automatisiertes Sammeln von Malware unter Windows, September 2007. Diplomarbeit.

P. Trinius. Malwareklassifikation und -clustering mit MIST. In *Proceedings of Third GI SIG SIDAR Graduate Workshop on Reactive Security (SPRING)*, 2008.

P. Trinius, T. Holz, J. Göbel, and F. C. Freiling. Visual Analysis of Malware Behavior Using Treemaps and Thread Graphs. In *Proceedings of the 6th International Workshop on Visualization for Cyber Security (VizSec'09)*, pages 33–38, 2009a.

P. Trinius, C. Willems, T. Holz, and K. Rieck. A Malware Instruction Set for Behavior-Based Analysis. Technical Report TR-2009-007, University of Mannheim, 2009b.

P. Trinius, C. Willems, T. Holz, and K. Rieck. A Malware Instruction Set for Behavior-Based Analysis. In *Proceedings of 5th GI Conference "Sicherheit, Schutz und Zuverlässigkeit"*, 2010.

N. Utakrit. A Review of Browser Extensions, a Man-in-the-Browser Phishing Techniques Targeting Bank Customers. In *7th Australian Information Security Management Conference*, 2009.

C. van Rijsbergen. *Information Retrieval*. Butterworths, 1979.

Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. T. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Proceedings of Symposium on Network and Distributed System Security (NDSS)*, 2006.

C. Willems. CWSandbox: Automatische Verhaltensanalyse von Malware. In *Proceedings of Second GI SIG SIDAR Graduate Workshop on Reactive Security (SPRING)*, 2007.

C. Willems, T. Holz, and F. Freiling. CWSandbox: Towards Automated Dynamic Binary Analysis. *IEEE Security and Privacy*, 5(2), March 2007.

C. Womser-Hacker. *Der PADOK-Retrievaltest: Zur Methode und Verwendung statistischer Verfahren bei der Bewertung von Information-Retrieval-Systemen*. G. Olms, 1989.

Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming Botnets: Signatures and Characteristics. In *Proceedings of SIGCOMM'08*, 2008.

M. Yigit. Aktuelle SpamBots (Studienarbeit), 2011.

Anhang

APPENDIX A

PLY

PLY ist eine reine PYTHON-Implementierung eines auf LEX und YACC basierenden Parsergenerators. Die in den Listings A.1 und A.2 dargestellten Skripte definieren die für den Lexer und Parser notwendigen Token und Grammatikregeln und erzeugen einen Parser für die durch diese definierte Grammatik. Zu untersuchende Sätze können dem Skript zur Laufzeit übergeben werden und werden anschließend lexikalisch und syntaktisch analysiert.

```
1  #!/usr/bin/env python
2  # encoding: utf-8
3
4  import ply.lex as lex
5  import ply.yacc as yacc
6
7  # Tokenliste
8  tokens = ('ARTIKEL',
9            'ADJEKTIV',
10           'SUBSTANTIV',
11           'PRAEDIKAT',)
12
13 # Regulaere Ausdruecke zur Definiton der Token
14 t_ARTIKEL       = r'der|die|das'
15 t_ADJEKTIV      = r'kleine|bissige|grosse'
16 t_SUBSTANTIV    = r'Hund|Katze'
17 t_PRAEDIKAT     = r'jagt'
18
19 #ignored characters
20 t_ignore = " \t"
21
22 def t_error(t):
23   print "Illegal character '%s'" % t.value[0]
24   t.lexer.skip(1)
```

Listing A.1: Der Parser für eine einfache natürliche Grammatik.

```
# Funktionen fuer die Grammatikregeln
def p_satz(p):
  '''satz : subjekt PRAEDIKAT objekt'''
  p[0] = p[1] + " " + p[1] + " " + p[3]
  pass

def p_subjekt(p):
  '''subjekt : ARTIKEL attribut SUBSTANTIV
             | attribut SUBSTANTIV'''
  if len(p) == 4:
    p[0] = p[1] + " " + p[2] + " " + p[3]
  else:
    p[0] = p[1] + " " + p[2]
  pass

def p_attribut(p):
  '''attribut : ADJEKTIV attribut
              | ADJEKTIV
              | '''
  if len(p) == 2:
    p[0] = p[1]
  elif len(p) == 3:
    p[0] = p[1] + " " + p[2]
  pass

def p_objekt(p):
  '''objekt : ARTIKEL attribut SUBSTANTIV
            | attribut SUBSTANTIV'''
  if len(p) == 4:
    p[0] = p[1] + " " + p[2] + " " + p[3]
  else:
    p[0] = p[1] + " " + p[2]
  pass

def p_error(t):
  print "Syntax error at '%s'" % t.value

# Build the lexer and parser
lex.lex()
yacc.yacc()

while 1:
  try:
    s = raw_input('sentence > ')
  except EOFError:
    break
  yacc.parse(s, debug=1)
```

Listing A.2: Der Parser für eine einfache natürliche Grammatik (Fortsetzung).

Für den Satz

der kleine bissige Hund jagt die grosse Katze

liefert das Skript die folgende Ausgabe:

```
sentence > der kleine bissige Hund jagt die grosse Katze
PLY: PARSE DEBUG START

State  : 0
Stack  : . LexToken(ARTIKEL,'der',1,0)
Action : Shift and goto state 3

State  : 3
Stack  : ARTIKEL . LexToken(ADJEKTIV,'kleine',1,4)
Action : Shift and goto state 1

State  : 1
Stack  : ARTIKEL ADJEKTIV . LexToken(ADJEKTIV,'bissige',1,11)
Action : Shift and goto state 1

State  : 1
Stack  : ARTIKEL ADJEKTIV ADJEKTIV . LexToken(SUBSTANTIV,'Hund',1,19)
Action : Reduce rule [attribut -> ADJEKTIV] with ['bissige'] and goto
    state 5
Result : <str @ 0x7f95c7ca27b0> ('bissige')

State  : 6
Stack  : ARTIKEL ADJEKTIV attribut . LexToken(SUBSTANTIV,'Hund',1,19)
Action : Reduce rule [attribut -> ADJEKTIV attribut] with ['kleine','
    bissige'] and goto state 4
Result : <str @ 0x7f95c7ca92d0> ('kleine bissige')

State  : 7
Stack  : ARTIKEL attribut . LexToken(SUBSTANTIV,'Hund',1,19)
Action : Shift and goto state 10

State  : 10
Stack  : ARTIKEL attribut SUBSTANTIV . LexToken(PRAEDIKAT,'jagt',1,24)
Action : Reduce rule [subjekt -> ARTIKEL attribut SUBSTANTIV] with ['der'
    ,<str @ 0x7f95c7ca92d0>,'Hund'] and goto state 2
Result : <str @ 0x7f95c7ca1af0> ('der kleine bissige Hund')

State  : 4
Stack  : subjekt . LexToken(PRAEDIKAT,'jagt',1,24)
Action : Shift and goto state 8

State  : 8
Stack  : subjekt PRAEDIKAT . LexToken(ARTIKEL,'die',1,29)
Action : Shift and goto state 12

State  : 12
Stack  : subjekt PRAEDIKAT ARTIKEL . LexToken(ADJEKTIV,'grosse',1,33)
Action : Shift and goto state 1

State  : 1
Stack  : subjekt PRAEDIKAT ARTIKEL ADJEKTIV . LexToken(SUBSTANTIV,'Katze'
    ,1,40)
Action : Reduce rule [attribut -> ADJEKTIV] with ['grosse'] and goto
    state 5
Result : <str @ 0x7f95c7ca27b0> ('grosse')
```

```
State  : 14
Stack  : subjekt PRAEDIKAT ARTIKEL attribut . LexToken(SUBSTANTIV,'Katze'
    ,1,40)
Action : Shift and goto state 16

State  : 16
Stack  : subjekt PRAEDIKAT ARTIKEL attribut SUBSTANTIV . $end
Action : Reduce rule [objekt -> ARTIKEL attribut SUBSTANTIV] with ['die',
    'grosse','Katze'] and goto state 7
Result : <str @ 0x7f95c7ca92d0> ('die grosse Katze')

State  : 11
Stack  : subjekt PRAEDIKAT objekt . $end
Action : Reduce rule [satz -> subjekt PRAEDIKAT objekt] with [<str @ 0
    x7f95c7ca1af0>,'jagt',<str @ 0x7f95c7ca92d0>] and goto state 1
Result : <str @ 0x7f95c7cab168> ('der kleine bissige Hund der kleine biss
     ...)

State  : 2
Stack  : satz . $end
Done   : Returning <str @ 0x7f95c7cab168> ('der kleine bissige Hund der
    kleine biss ...)
PLY: PARSE DEBUG END
```

Das Lernen neuer Wörter und Sätze erfolgt während der Fehlerbehandlungsroutine. Das heißt, erst wenn sich ein Wort oder Satz nicht mit Hilfe der bereits bekannten Token und Regeln ableiten lässt, wird die aktuelle Grammatik entsprechend erweitert. Dazu sind verschiedene Informationen, wie zum Beispiel der aktuelle Stand und der Stack des Parsers, notwendig, die von dem Modul standardmäßig nicht bereitgestellt werden. Der in Listing A.3 dargestellte Patch für das Modul *yacc.py* erweitert dieses um die notwendigen Codezeilen.

```
--- /usr/lib/pymodules/python2.6/ply/yacc.py    2009-09-02
    16:27:23.000000000 +0200
+++ yacc.py     2011-08-05 12:04:48.000000000 +0200
@@ -59,7 +59,7 @@
 # own risk!
 # ----------------------------------------------------------------------

-__version__    = "3.3"
+__version__    = "3.2"
 __tabversion__ = "3.2"       # Table version

 #-----------------------------------------------------------------------
@@ -211,7 +211,7 @@
          return getattr(self.slice[n],"lineno",0)

     def set_lineno(self,n,lineno):
-        self.slice[n].lineno = lineno
+        self.slice[n].lineno = n

     def linespan(self,n):
         startline = getattr(self.slice[n],"lineno",0)
@@ -242,6 +242,10 @@
         self.action      = lrtab.lr_action
         self.goto        = lrtab.lr_goto
         self.errorfunc   = errorf
+
+       self.lookahead = None
+       self.lookaheadstack = []
+       self.state = 1

     def errok(self):
         self.errorok     = 1
@@ -282,6 +286,10 @@
     def parsedebug(self,input=None,lexer=None,debug=None,tracking=0,
         tokenfunc=None):
         lookahead = None                 # Current lookahead symbol
         lookaheadstack = [ ]             # Stack of lookahead symbols
+
+       self.lookahead = lookahead
+       self.lookaheadstack = lookaheadstack
+
         actions = self.action            # Local reference to action
             table (to avoid lookup on self.)
         goto    = self.goto              # Local reference to goto table
              (to avoid lookup on self.)
         prod    = self.productions       # Local reference to production
              list (to avoid lookup on self.)
@@ -328,6 +336,7 @@
         sym.type = "$end"
         symstack.append(sym)
         state = 0
+       self.state = state
         while 1:
             # Get the next symbol on the input.  If a lookahead symbol
```

```
50              # is already set, we just use that. Otherwise, we'll pull
51  @@ -347,6 +356,10 @@
52                        lookahead = YaccSymbol()
53                        lookahead.type = "$end"
54
55  +               self.lookahead = lookahead
56  +               self.lookaheadstack = lookaheadstack
57  +
58  +
59              # --! DEBUG
60              debug.debug('Stack  : %s',
61                          ("%s . %s" % (" ".join([xx.type for xx in
                                symstack][1:]), str(lookahead))).lstrip())
62  @@ -361,6 +374,7 @@
63                        # shift a symbol on the stack
64                        statestack.append(t)
65                        state = t
66  +                    self.state = state
67
68                        # --! DEBUG
69                        debug.debug("Action : Shift and goto state %s", t)
70  @@ -424,6 +438,7 @@
71                                # --! DEBUG
72                                symstack.append(sym)
73                                state = goto[statestack[-1]][pname]
74  +                             self.state = state
75                                statestack.append(state)
76                            except SyntaxError:
77                                # If an error was set. Enter error recovery
                                    state
78  @@ -510,13 +525,14 @@
79                        if errtoken.type == "$end":
80                            errtoken = None                # End of file!
81                        if self.errorfunc:
82  -                         global errok,token,restart
83  +                         global errok, token, restart
84                            errok = self.errok        # Set some special
                                  functions available in error recovery
85                            token = get_token
86                            restart = self.restart
87                            if errtoken and not hasattr(errtoken,'lexer'):
88                                errtoken.lexer = lexer
89                            tok = self.errorfunc(errtoken)
90  +
91                            del errok, token, restart   # Delete special
                                  functions
92
93                            if self.errorok:
94  @@ -869,6 +885,10 @@
95        def parseopt_notrack(self,input=None,lexer=None,debug=0,tracking=0,
              tokenfunc=None):
96            lookahead = None                 # Current lookahead symbol
97            lookaheadstack = [ ]             # Stack of lookahead symbols
98  +
99  +       self.lookahead = lookahead
100 +       self.lookaheadstack = lookaheadstack
101 +
102           actions = self.action            # Local reference to action
                  table (to avoid lookup on self.)
103           goto    = self.goto              # Local reference to goto table
                   (to avoid lookup on self.)
104           prod    = self.productions       # Local reference to production
                 list (to avoid lookup on self.)
105 @@ -911,6 +931,7 @@
106           sym.type = '$end'
107           symstack.append(sym)
```

```
108            state = 0
109  +         self.state = state
110            while 1:
111                # Get the next symbol on the input.  If a lookahead symbol
112                # is already set, we just use that. Otherwise, we'll pull
113  @@ -925,6 +946,9 @@
114                        lookahead = YaccSymbol()
115                        lookahead.type = '$end'
116
117  +                 self.lookahead = lookahead
118  +                 self.lookaheadstack = lookaheadstack
119  +
120                # Check the action table
121                ltype = lookahead.type
122                t = actions[state].get(ltype)
123  @@ -934,6 +958,8 @@
124                        # shift a symbol on the stack
125                        statestack.append(t)
126                        state = t
127  +
128  +                     self.state = state
129
130                        symstack.append(lookahead)
131                        lookahead = None
132  @@ -971,6 +997,9 @@
133                                p.callable(pslice)
134                                symstack.append(sym)
135                                state = goto[statestack[-1]][pname]
136  +
137  +                             self.state = state
138  +
139                                statestack.append(state)
140                            except SyntaxError:
141                                # If an error was set. Enter error recovery
                                      state
```

Listing A.3: Patch für die Klasse *yacc.py* des PLY-Modules.

APPENDIX B

Antiviren-Scanner

Zur Evaluierung der MIST-Repräsentation wurde ein Referenzdatensatz von Schadprogrammen unterschiedlicher Familien benötigt. Dazu wurden rund 70.000 Schadprogramme aus dem Datenbestand des MwAnalysis-Projektes an den freien Analyseservice VIRUSTOTAL übertragen, wo die Dateien mit 33 verschiedenen Virenscannern untersucht wurden. Die Erkennungsraten der einzelnen Virenscanner auf diesen Schadprogrammen sind in Tabellen B.1 dargestellt.

Die zum Markieren des Refenzdatensatzes verwendeten Virenscanner sind grau hinterlegt und gehören zu den zehn „besten“ Virenscannern. Neben der Erkennungsrate wurde auch die von den Virenscannern gelieferte Klassifizierung der erkannten Schadprogramme herangezogen. Die Virenscanner sollten den Datensatz nach Möglichkeit in äquivalente Familien – nicht notwendigerweise mit der gleichen Bezeichnung – unterteilen. Diese beiden Kriterien führten zu der dargestellten Auswahl.

Die 70.000 Schadprogramme wurden mit den sechs Virenscannern in Klassen unterteilt. Entscheidend für die Zuordnung in eine Klasse war dabei, dass die Mehrzahl der Virenscanner das Schadprogramm derselben Klasse zuordnete. Anschließend wurden diejenigen Klassen mit weniger als 25 Mitgliedern aus dem Datensatz entfernt und alle übrigen Klassen wurden auf 300 Schadprogramme beschränkt. In Tabellen B.2 sind die Erkennungsraten der von VIRUSTOTAL eingesetzten Virenscanner für diesen Referenzdatensatz dargestellt.

	Anti-Virus Lösung	kein Ergebnis	nicht infiziert	ER in %
1	WEBWASHER-GATEWAY	21.559	5.065	90,10
2	IKARUS	2	8.186	88,74
3	ANTIVIR	9	10.434	85,65
4	GDATA	1.896	15.102	78,67
5	SOPHOS	9	16222	77,69
6	BITDEFENDER	23	16.540	77,24
7	F-SECURE	808	17.152	76,14
8	AVG	132	18.030	75,16
9	FORTINET	11	20.061	72,41
10	KASPERSKY	92	20.218	72,16
11	NORMAN	131	22.069	69,59
12	CAT-QUICKHEAL	7	22.305	69,32
13	F-PROT	15	22.741	68,72
14	MICROSOFT	16	24.239	66,66
15	DRWEB	73	24.536	66,22
16	VBA32	764	24.359	66,14
17	PANDA	8	24.628	66,12
18	ESAFE	38	25.247	65,26
19	SYMANTEC	5.881	24.001	64,09
20	AUTHENTIUM	39	26.151	64,01
21	MCAFEE	28	26.273	63,85
22	NOD32V2	23.305	18.673	62,20
23	TRENDMICRO	22.894	19.677	60,50
24	VIRUSBUSTER	12	29.059	60,03
25	CLAMAV	32	30.166	58,49
26	RISING	98	32.879	54,72
27	SUNBELT	136	35049	51,71
28	AHNLAB-V3	58	35.359	51,33
29	ETRUST-VET	20	38.746	46,70
30	THEHACKER	1.030	43.279	39,62
31	PREVX1	1.417	48.932	31,36
32	EWIDO	0	58.946	18,93
33	FILEADVISOR	70.115	2.505	3,43

Tabelle B.1: Performanz der Anti-Virus Lösungen auf einem Datensatz von 72.709 Schadprogrammen. ER bezeichnet dabei die Erkennungsrate der Virenscanner. Die sechs schlussendlich verwendeten Virenscanner sind grau hinterlegt.

	Anti-Virus Lösung	kein Ergebnis	nicht infiziert	ER in %
1	SOPHOS	0	13	99,58
2	IKARUS	1	44	98,59
3	WEBWASHER-GATEWAY	608	137	94,57
4	ANTIVIR	0	298	90,48
5	BITDEFENDER	0	460	85,31
6	GDATA	2	482	84,60
7	AVG	11	489	84,33
8	KASPERSKY	2	594	81,02
9	FORTINET	0	615	80,36
10	ESAFE	1	648	79,30
11	VIRUSBUSTER	0	659	78,95
12	MICROSOFT	0	698	77,71
13	F-SECURE	46	726	76,47
14	AVAST	0	742	76,30
15	SYMANTEC	220	793	72,76
16	F-PROT	0	867	72,31
17	TRENDMICRO	1669	417	71,48
18	NORMAN	4	903	71,12
19	DRWEB	3	992	68,29
20	RISING	5	996	68,14
21	MCAFEE	3	1022	67,33
22	AUTHENTIUM	1	1041	66,74
23	PANDA	0	1111	64,52
24	CLAMAV	0	1143	63,49
25	CAT-QUICKHEAL	0	1149	63,30
26	VBA32	39	1153	62,71
27	NOD32V2	676	923	62,40
28	ETRUST-VET	0	1357	56,66
29	THEHACKER	42	1619	47,59
30	AHNLAB-V3	0	1835	41,39
31	SUNBELT	8	2110	32,44
32	PREVX1	73	2101	31,29
33	EWIDO	0	2209	29,45
34	FILEADVISOR	3004	127	0,0

Tabelle B.2: Performanz der Anti-Virus Lösungen auf dem Referenzdatensatz mit 3.131 Schadprogrammen. ER bezeichnet dabei die Erkennungsrate der Virenscanner. Die sechs schlussendlich verwendeten Virenscanner sind grau hinterlegt.

APPENDIX C

MIST-Lexer, -Parser und -Filter

Das zum Speichern der 2-Gramme aufgebaute Datawarehouse ist in Abbildung C.1 dargestellt. Das Schema besteht aus vier Dimensions-Tabellen (*cluster*, *analysis*, *samples* und *mistword*) und drei Fakten-Tabellen (*ngram*, *cluster_ngram*, *analysis_cluster* und *analysis_ngram*). In den Dimensions-Tabellen werden die eigentlichen Daten, wie Clusternamen und Analysereporte, abgespeichert. Die Fakten-Tabellen stellen lediglich eine Verbindung zwischen diesen Informationen her. Da in den Fakten-Tabllen nur die Identifikationsnummern (*ID*s) der Dimensions-Tabellen enthalten sind, werden in diesem Schema keine redundanten Daten gespeichert. Wird in dem Schema ein neues Cluster definiert, lassen sich über die den Cluster zugewiesenen Analyse-IDs sehr komfortabel die das Cluster charakterisierenden 2-Gramme berechnen.

In den Listings C.1 sowie C.2 und C.3 ist der Quellcode der PYTHON- Klassen *MISTLexer* und *MISTParser* dargestellt. Diese stellen die erweiterte Grundgrammatik für MIST bereit und werden für die einzelnen Schadprogrammfamilien lediglich um die zu suchenden 2-Gramme und die entsprechenden Regeln erweitert. Der Quellcode des dazu entwickelten *Parsergenerators* ist in den Listings C.4 und C.5 dargestellt.

Abbildung C.1: Aufbau des Datawarehouse zur Speicherung der n-Gramme. Sowohl den Analysen als auch den Clustern werden die entsprechenden n-Gramme direkt zugeordnet.

```
#!/usr/bin/env python
# encoding: utf-8

"""
MISTLexer.py

Created by Philipp Trinius on 2010-03-09.
Copyright (c) 2010 University of Mannheim. All rights reserved.
"""

import sys
import os
import ply.lex as lex

class MISTLexer(object):

  def t_error(self, t):
    raise TypeError("Illegal charachter '%s'" % (t.value[0]))
    pass

  def __init__(self):
    self.tokens = ['INFO', 'HEXNUMBER', 'SEPARATOR', 'NEWLINE']

    # Tokens definition
    self.t_INFO = r'\#\ process\ [0-9a-f]{8}\ [0-9a-f]{8}
                    \ [0-9a-f]{8}\ [0-9a-f]{8}
                    \ thread\ [0-9a-f]{4}\ \#\n'
    self.t_HEXNUMBER = r'[0-9a-f]+'
    self.t_SEPARATOR = r'\|'
    self.t_NEWLINE = r'\n'

    #Ignored characters
    self.t_ignore = ' \t'

    #nGrams 2 find
    self.ngrams = []
    self.check_ngrams = []

  def reset(self):
    self.check_ngrams = list(self.ngrams)

  # Build the lexer
  def build(self, **kwargs):
    self.lexer = lex.lex(module=self, **kwargs)

  def escape(self, string):
    return string.replace(" ", "\ ").replace("|", "\|")

  def test(self, data, debug=False):
    try:
      self.lexer.input(data)
      while True:
        tok = self.lexer.token()
        if not tok:
          break
      print "passed."
      return True
    except TypeError,e:
      print "failed (%s)" % (e)
      return False
```

Listing C.1: Die PYTHON-Klasse *MISTLexer*.

```
#!/usr/bin/env python
# encoding: utf-8
"""
MISTParser.py

Created by Philipp Trinius on 2010-03-09.
Copyright (c) 2010 University of Mannheim. All rights reserved.
"""

import sys
import os
from ply.yacc import yacc

class MISTParser(object):

  # Parsing rules
  def p_mistreport(self, p):
    '''mistreport : mistthreads'''
    if len(self.lexer.check_ngrams) != 0:
      raise
    pass

  def p_mistthreads(self, p):
    '''mistthreads : mistthread
                   | mistthreads mistthread'''
    pass

  def p_mistthread(self, p):
    '''mistthread : INFO mistwords
                  | empty'''
    pass

  def p_mistwords(self, p):
    '''mistwords : mistword
                 | mistwords mistword'''
    pass

  def p_mistword(self, p):
    '''mistword : apicall SEPARATOR NEWLINE
                | apicall SEPARATOR attributes NEWLINE
                | INFO
                | empty'''
    pass

  def p_apicall(self,p):
    '''apicall : section operation'''
    pass

  def p_attributes(self, p):
    '''attributes : attribute
                  | attributes attribute'''
    pass

  def p_section(self, p):
    '''section : HEXNUMBER'''
    pass

  def p_operation(self, p):
    '''operation : HEXNUMBER'''
    pass

  def p_attribute(self, p):
    '''attribute : HEXNUMBER'''
    pass
```

Listing C.2: Die PYTHON-Klasse *MISTParser*.

```
1    def p_empty(self, p):
2      '''empty :
3               | NEWLINE'''
4      pass
5
6    def p_ngram(self, p):
7      '''ngram : empty'''
8      pass
9
10   def p_error(self, p):
11     raise TypeError("Pattern not found")
12
13   # Build the parser
14   def build(self, lexer, startpoint='mistreport'):
15     self.lexer = lexer
16     self.tokens = lexer.return_tokens()
17     self.parser = yacc(module=self, start=startpoint)
18
19   def escape(self, string):
20     return string.replace(" ", "\ ").replace("|", "\|")
21
22   def test(self, data, debug=False):
23     self.lexer.reset()
24     try:
25       self.parser.parse(data)
26       print "passed"
27       return True
28     except TypeError, e:
29       print "failed! (%s)" % (e)
30       return False
```

Listing C.3: Die PYTHON-Klasse *MISTParser* (Fortsetzung).

```
#!/usr/bin/env python
# encoding: utf-8
"""
MISTLexYacc_ParserGenerator.py
Created by Philipp Trinius on 2010-03-09.
Copyright (c) 2010 University of Mannheim. All rights reserved.
"""

import sys
import os
from pgsqlObj import pgsqlObj

def escape(string):
  return string.replace(" ", "\ ").replace("|", "\|")

def generate_Pattern(db_ngrams):
  pattern = []
  ngrams = ''
  for ngram in db_ngrams:
    parts = ngram.split("\n")
    if pattern.count(parts[0]) == 0:
      pattern.append(parts[0])
    if pattern.count(parts[1]) == 0:
      pattern.append(parts[1])
    ngrams += "self.ngrams.append(('" + escape(parts[0]) + "\\n', '" +
        escape(parts[1]) + "\\n'))\n\t\t"
  i = 0
  token_names = ''
  token_definitions = ''
  token_list = ''
  for pat in pattern:
    token_names += "self.tokens.append('PATTERN_"+ str(i) + "')\n\t\t"
    token_definitions += "self.t_PATTERN_"+ str(i) + " = r'"+ escape(pat)
        +"\\n'\n\t\t"
    token_list += " PATTERN_"+ str(i) + "\n\t\t\t\t|"
    i += 1
  return token_names, token_definitions, ngrams, token_list


def write_lexer(classname, token_names, token_definitions, ngram_list):
  ffile = open("MISTLexer_Template.py", "r")
  lexer = ffile.read()
  ffile.close()
  lexer = lexer.replace("$$MALWARE_FAMILY$$", classname)
  lexer = lexer.replace("$$TOKEN_NAMES$$", token_names)
  lexer = lexer.replace("$$TOKEN_DEFINITIONS$$", token_definitions)
  lexer = lexer.replace("$$NGRAMS_LIST$$", ngram_list)
  ffile = open("filter/" + classname + "Lexer.py", "w")
  ffile.write(lexer)
  ffile.close()

def write_parser(classname, token_list):
  ffile = open("MISTParser_Template.py", "r")
  parser = ffile.read()
  ffile.close()
  parser = parser.replace("$$MALWARE_FAMILY$$", classname)
  parser = parser.replace("$$TOKEN_LIST$$", token_list)
  ffile = open("filter/" + classname + "Parser.py", "w")
  ffile.write(parser)
  ffile.close()
```

Listing C.4: Das PYTHON-Skript *MISTLexYacc_ParserGenerator*.

```
1  def write_filter(classname):
2    ffile = open("MISTFilter_Template.py", "r")
3    ffilter = ffile.read()
4    ffile.close()
5    ffilter = ffilter.replace("$$MALWARE_FAMILY$$", classname)
6    ffile = open("filter/" + classname + "Filter.py", "w")
7    ffile.write(ffilter)
8    ffile.close()
9
10 if __name__ == '__main__':
11   cwsDB = pgsqlObj(db="database", dbuser="dbuser", dbpass="dbpw", dbhost=
         "localhost", dbport=5432, dblogDir="log")
12   cluster = {}
13   print "Read Cluster from database ...",
14   sys.stdout.flush()
15   (count, db_results) = cwsDB.query("SELECT * FROM mist_ngrams.cluster")
16   if count > 0 and db_results != -1:
17     for db_result in db_results:
18       cid = db_result["id"]
19       cluster[cid] = db_result["name"]
20   print " done."
21
22   for cid in cluster:
23     pattern = []
24     print "Read ngrams of cluster " +str(cluster[cid])+ " from database."
25     (count, db_results) = cwsDB.query("SELECT mistword1_id, mistword2_id
           FROM mist_ngrams.ngram JOIN mist_ngrams.cluster_ngram ON
           ngram_id = id WHERE cluster_id = %i" % (int(cid)))
26     print count
27     if count > 0 and db_results != -1:
28       for db_result in db_results:
29         mist1_id = db_result["mistword1_id"]
30         mist2_id = db_result["mistword2_id"]
31         print ".",
32         (count, db_results) = cwsDB.query("SELECT mist FROM mist_ngrams.
               mistword WHERE id = %i" % (int(mist1_id)))
33         if count > 0 and db_results != -1:
34           mist1 = db_results[0]["mist"]
35         (count, db_results) = cwsDB.query("SELECT mist FROM mist_ngrams.
               mistword WHERE id = %i" % (int(mist2_id)))
36         if count > 0 and db_results != -1:
37           mist2 = db_results[0]["mist"]
38         pattern.append(mist1 + "\n" + mist2)
39     print " done."
40
41     print "Generate Parser",
42     (token_names, token_definitions, ngrams, token_list) =
           generate_Pattern(pattern)
43     write_lexer(cluster[cid], token_names, token_definitions, ngrams)
44     write_parser(cluster[cid], token_list)
45     write_filter(cluster[cid])
46     print " done."
```

Listing C.5: Das PYTHON-Skript *MISTLexYacc_ParserGenerator* (Fortsetzung).

APPENDIX D

Filtern mit regulären Ausdrücken

Die folgenden Tabellen zeigen die Details zu der in Abschnitt 4.3.2 vorgestellten Evaluierung der auf regulären Ausdrücken basierenden Spam-Filter. Der Filteransatz wurde auf insgesamt 437 Kampagnen des STORM-Botnets angewandt. Dabei wurden jeweils eine, fünf, fünfzig, fünfhundert und fünftausend Nachrichten zur Filtergenerierung herangezogen. Die Tabellen D.1 bis D.12 zeigt die mit den resultierenden Filtern errichten Filterergebnisse, auf den fünftausend aus der jeweiligen Kampagne gesammelten Nachrichten.

Kampagne	**1**	**5**	**50**	**500**	**5000**
6032737	10/4990	2072/2928	4743/257	5000/0	5000/0
6032786	11/4989	892/4108	4508/492	5000/0	5000/0
6032982	11/4989	1881/3119	4756/244	5000/0	5000/0
6033164	20/4980	817/4183	4436/564	5000/0	5000/0
6033271	27/4973	972/4028	4767/233	5000/0	5000/0
6033534	76/4924	3610/1390	4284/716	5000/0	5000/0
6033961	87/4913	431/4569	5000/0	5000/0	5000/0
6034245	88/4912	1063/3937	4276/724	5000/0	5000/0
6034383	65/4935	2447/2553	4823/177	5000/0	5000/0
6034666	17/4983	2125/2875	4777/223	5000/0	5000/0
6034688	70/4930	1320/3680	4559/441	5000/0	5000/0
6035054	92/4908	1243/3757	4493/507	5000/0	5000/0
6035146	9/4991	3058/1942	4583/417	5000/0	5000/0
6035380	75/4925	3516/1484	4813/187	5000/0	5000/0
6035498	86/4914	1701/3299	4497/503	5000/0	5000/0
6036057	12/4988	560/4440	4257/743	5000/0	5000/0
6036127	31/4969	1503/3497	4919/81	5000/0	5000/0

Tabelle D.1: Vollständige Filterergebnisse des STOM-Botnetz.

Kampagne	0.02%	0.1%	1%	10%	100%
6036353	71/4929	1222/3778	4489/511	5000/0	5000/0
6036393	83/4917	1918/3082	4577/423	5000/0	5000/0
6036590	26/4974	1342/3658	4758/242	5000/0	5000/0
6036638	79/4921	1192/3808	4297/703	5000/0	5000/0
6036681	23/4977	1041/3959	4099/901	5000/0	5000/0
6037283	7/4993	1700/3300	4651/349	5000/0	5000/0
6037675	6/4994	1160/3840	4569/431	5000/0	5000/0
6037891	89/4911	901/4099	4825/175	5000/0	5000/0
6038074	33/4967	2112/2888	4490/510	5000/0	5000/0
6038096	90/4910	656/4344	4692/308	5000/0	5000/0
6038158	86/4914	811/4189	4749/251	5000/0	5000/0
6038412	17/4983	1750/3250	4829/171	5000/0	5000/0
6038424	103/4897	1236/3764	4855/145	5000/0	5000/0
6038431	27/4973	1004/3996	4080/920	5000/0	5000/0
6038553	89/4911	2238/2762	4480/520	5000/0	5000/0
6038725	69/4931	3321/1679	4841/159	5000/0	5000/0
6039083	85/4915	815/4185	4363/637	5000/0	5000/0
6039580	7/4993	2367/2633	4830/170	5000/0	5000/0
6040034	17/4983	1953/3047	4498/502	5000/0	5000/0
6040230	89/4911	2326/2674	4747/253	5000/0	5000/0
6040577	8/4992	404/4596	4183/817	5000/0	5000/0
6041266	27/4973	1101/3899	4836/164	5000/0	5000/0
6041285	35/4965	1380/3620	4032/968	5000/0	5000/0
6041367	28/4972	908/4092	4921/79	5000/0	5000/0
6041636	72/4928	2606/2394	4269/731	5000/0	5000/0
6041846	27/4973	2198/2802	4572/428	5000/0	5000/0
6041889	73/4927	1082/3918	4392/608	5000/0	5000/0
6042180	18/4982	690/4310	4354/646	5000/0	5000/0
6042202	20/4980	570/4430	4740/260	5000/0	5000/0
6042994	25/4975	1539/3461	4479/521	5000/0	5000/0
6043509	77/4923	2082/2918	4922/78	5000/0	5000/0
6043744	105/4895	2281/2719	4354/646	5000/0	5000/0
6043857	91/4909	1185/3815	4529/471	5000/0	5000/0
6043998	20/4980	1412/3588	4571/429	5000/0	5000/0
6044026	81/4919	732/4268	4895/105	5000/0	5000/0
6044444	25/4975	1584/3416	4773/227	5000/0	5000/0
6044759	86/4914	1558/3442	4334/666	5000/0	5000/0
6045129	26/4974	931/4069	4733/267	5000/0	5000/0
6045501	27/4973	1486/3514	4492/508	5000/0	5000/0
6045914	82/4918	2045/2955	4273/727	5000/0	5000/0

Tabelle D.2: Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung).

Kampagne	0.02%	0.1%	1%	10%	100%
6046341	94/4906	1023/3977	4510/490	5000/0	5000/0
6046510	86/4914	742/4258	5000/0	5000/0	5000/0
6046700	7/4993	841/4159	3893/1107	5000/0	5000/0
6046848	28/4972	368/4632	4506/494	5000/0	5000/0
6046987	7/4993	1362/3638	4845/155	5000/0	5000/0
6047058	103/4897	2013/2987	4235/765	5000/0	5000/0
6047254	81/4919	502/4498	4734/266	5000/0	5000/0
6047261	92/4908	1659/3341	4911/89	5000/0	5000/0
6047429	73/4927	721/4279	4503/497	5000/0	5000/0
6047753	79/4921	847/4153	4846/154	5000/0	5000/0
6047899	78/4922	2911/2089	4666/334	5000/0	5000/0
6048032	93/4907	2682/2318	4829/171	5000/0	5000/0
6048088	86/4914	2401/2599	4423/577	5000/0	5000/0
6048193	93/4907	2427/2573	4836/164	5000/0	5000/0
6048238	95/4905	1682/3318	4789/211	5000/0	5000/0
6048370	8/4992	1686/3314	4651/349	5000/0	5000/0
6048372	91/4909	996/4004	4582/418	5000/0	5000/0
6048491	24/4976	708/4292	4835/165	5000/0	5000/0
6048637	87/4913	2378/2622	4570/430	5000/0	5000/0
6048749	22/4978	3061/1939	4838/162	5000/0	5000/0
6048840	9/4991	1043/3957	4362/638	5000/0	5000/0
6048896	81/4919	2402/2598	4824/176	5000/0	5000/0
6049008	80/4920	1103/3897	4325/675	5000/0	5000/0
6049050	73/4927	828/4172	4420/580	5000/0	5000/0
6049330	78/4922	1942/3058	4350/650	5000/0	5000/0
6049505	53/4947	2091/2909	4416/584	5000/0	5000/0
6049552	93/4907	1252/3748	4387/613	5000/0	5000/0
6049608	86/4914	1824/3176	4423/577	5000/0	5000/0
6049709	7/4993	1776/3224	4599/401	5000/0	5000/0
6049884	89/4911	2026/2974	4570/430	5000/0	5000/0
6049952	6/4994	938/4062	4580/420	5000/0	5000/0
6050107	10/4990	823/4177	4760/240	5000/0	5000/0
6050235	30/4970	1698/3302	4434/566	5000/0	5000/0
6050385	25/4975	366/4634	4519/481	5000/0	5000/0
6051179	7/4993	1346/3654	4859/141	5000/0	5000/0
6051811	88/4912	3126/1874	4657/343	5000/0	5000/0
6052024	30/4970	966/4034	4343/657	5000/0	5000/0
6052076	4/4996	1221/3779	4793/207	5000/0	5000/0
6052566	76/4924	1334/3666	4324/676	5000/0	5000/0
6052734	84/4916	691/4309	4807/193	5000/0	5000/0

Tabelle D.3: Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung).

Kampagne	0.02%	0.1%	1%	10%	100%
6052775	33/4967	1537/3463	4503/497	5000/0	5000/0
6053142	89/4911	1068/3932	4844/156	5000/0	5000/0
6053244	19/4981	2113/2887	4922/78	5000/0	5000/0
6053502	88/4912	1004/3996	4568/432	5000/0	5000/0
6053597	80/4920	1090/3910	4825/175	5000/0	5000/0
6053611	12/4988	1938/3062	4419/581	5000/0	5000/0
6053891	33/4967	1321/3679	4925/75	5000/0	5000/0
6053970	6/4994	1052/3948	4932/68	5000/0	5000/0
6054059	83/4917	1456/3544	4836/164	5000/0	5000/0
6054318	82/4918	1247/3753	4940/60	5000/0	5000/0
6054590	86/4914	947/4053	4856/144	5000/0	5000/0
6054941	32/4968	797/4203	4608/392	5000/0	5000/0
6055168	88/4912	1856/3144	4817/183	5000/0	5000/0
6055187	98/4902	2234/2766	5000/0	5000/0	5000/0
6055432	87/4913	1353/3647	4588/412	5000/0	5000/0
6055587	93/4907	1439/3561	4298/702	4926/74	5000/0
6055916	74/4926	1717/3283	4523/477	5000/0	5000/0
6055987	81/4919	1832/3168	4826/174	5000/0	5000/0
6056676	86/4914	961/4039	4734/266	5000/0	5000/0
6056809	21/4979	1742/3258	4917/83	5000/0	5000/0
6056908	102/4898	1290/3710	3983/1017	5000/0	5000/0
6056980	28/4972	631/4369	4832/168	5000/0	5000/0
6057002	87/4913	2606/2394	4600/400	5000/0	5000/0
6057033	16/4984	755/4245	4804/196	5000/0	5000/0
6057149	75/4925	1148/3852	4436/564	5000/0	5000/0
6057845	88/4912	2681/2319	4674/326	5000/0	5000/0
6058549	112/4888	2153/2847	4422/578	5000/0	5000/0
6058759	92/4908	3186/1814	4700/300	5000/0	5000/0
6059287	78/4922	891/4109	4667/333	5000/0	5000/0
6059769	85/4915	2483/2517	4588/412	5000/0	5000/0
6059826	32/4968	2064/2936	4334/666	5000/0	5000/0
6060126	27/4973	2720/2280	4739/261	5000/0	5000/0
6060155	86/4914	902/4098	4844/156	5000/0	5000/0
6061237	80/4920	1032/3968	4921/79	5000/0	5000/0
6061345	22/4978	1250/3750	4412/588	4935/65	5000/0
6061457	3/4997	1567/3433	4263/737	5000/0	5000/0
6061541	7/4993	1065/3935	4341/659	5000/0	5000/0
6061660	9/4991	800/4200	4673/327	5000/0	5000/0
6062077	18/4982	1524/3476	4611/389	5000/0	5000/0
6062090	24/4976	2055/2945	4729/271	5000/0	5000/0

Tabelle D.4: Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung).

Kampagne	0.02%	0.1%	1%	10%	100%
6062136	26/4974	759/4241	4559/441	5000/0	5000/0
6062623	21/4979	2143/2857	4732/268	5000/0	5000/0
6063110	9/4991	1636/3364	4578/422	5000/0	5000/0
6063125	6/4994	2206/2794	4756/244	5000/0	5000/0
6063253	12/4988	3208/1792	4668/332	5000/0	5000/0
6063688	26/4974	1730/3270	4467/533	5000/0	5000/0
6063863	28/4972	814/4186	4665/335	5000/0	5000/0
6064046	30/4970	1013/3987	4673/327	5000/0	5000/0
6064075	89/4911	1933/3067	3981/1019	5000/0	5000/0
6064403	10/4990	2670/2330	4656/344	5000/0	5000/0
6064487	8/4992	2891/2109	4592/408	5000/0	5000/0
6064594	94/4906	2973/2027	4654/346	5000/0	5000/0
6064612	31/4969	1490/3510	4670/330	5000/0	5000/0
6065886	84/4916	2014/2986	4682/318	5000/0	5000/0
6065994	24/4976	2013/2987	4215/785	5000/0	5000/0
6066149	81/4919	1789/3211	4442/558	5000/0	5000/0
6066250	92/4908	1858/3142	4106/894	5000/0	5000/0
6066855	33/4967	1112/3888	4753/247	5000/0	5000/0
6067091	13/4987	1329/3671	4904/96	5000/0	5000/0
6067118	24/4976	2096/2904	4355/645	5000/0	5000/0
6067174	74/4926	1131/3869	4664/336	5000/0	5000/0
6067276	12/4988	907/4093	4614/386	5000/0	5000/0
6067385	59/4941	2140/2860	4319/681	5000/0	5000/0
6067433	97/4903	1500/3500	4844/156	5000/0	5000/0
6067858	29/4971	1872/3128	4826/174	5000/0	5000/0
6068187	30/4970	1865/3135	4924/76	5000/0	5000/0
6068211	21/4979	1970/3030	4188/812	5000/0	5000/0
6068315	79/4921	2768/2232	4486/514	5000/0	5000/0
6068480	88/4912	2498/2502	4843/157	5000/0	5000/0
6068933	100/4900	1408/3592	4570/430	5000/0	5000/0
6069104	95/4905	2755/2245	4414/586	5000/0	5000/0
6069194	94/4906	833/4167	4496/504	5000/0	5000/0
6069438	7/4993	1109/3891	4363/637	5000/0	5000/0
6069449	97/4903	2717/2283	4490/510	5000/0	5000/0
6069501	24/4976	900/4100	4429/571	5000/0	5000/0
6070324	108/4892	2363/2637	4693/307	5000/0	5000/0
6070348	92/4908	1550/3450	4732/268	5000/0	5000/0
6070887	32/4968	1134/3866	4572/428	5000/0	5000/0
6071272	28/4972	765/4235	4759/241	5000/0	5000/0
6071348	26/4974	2030/2970	4131/869	5000/0	5000/0

Tabelle D.5: Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung).

Kampagne	0.02%	0.1%	1%	10%	100%
6071513	9/4991	2137/2863	4831/169	5000/0	5000/0
6071662	98/4902	2613/2387	4829/171	5000/0	5000/0
6071699	87/4913	799/4201	4914/86	5000/0	5000/0
6071748	78/4922	1999/3001	5000/0	5000/0	5000/0
6071989	84/4916	1658/3342	4744/256	5000/0	5000/0
6071998	23/4977	2703/2297	4919/81	5000/0	5000/0
6072497	97/4903	1442/3558	4764/236	5000/0	5000/0
6072609	92/4908	1071/3929	4537/463	5000/0	5000/0
6072819	73/4927	1645/3355	4311/689	5000/0	5000/0
6073119	27/4973	1253/3747	4573/427	5000/0	5000/0
6073145	94/4906	1136/3864	4334/666	5000/0	5000/0
6073169	9/4991	3750/1250	4568/432	5000/0	5000/0
6073346	12/4988	620/4380	4830/170	5000/0	5000/0
6073524	27/4973	1072/3928	4704/296	5000/0	5000/0
6073681	25/4975	1548/3452	4593/407	5000/0	5000/0
6073893	79/4921	730/4270	4158/842	5000/0	5000/0
6074088	84/4916	1329/3671	4829/171	5000/0	5000/0
6074296	23/4977	3130/1870	4755/245	5000/0	5000/0
6074653	75/4925	744/4256	4746/254	5000/0	5000/0
6074793	92/4908	2670/2330	4846/154	5000/0	5000/0
6075256	29/4971	2276/2724	4772/228	5000/0	5000/0
6075332	87/4913	2598/2402	4926/74	5000/0	5000/0
6075450	24/4976	781/4219	4741/259	5000/0	5000/0
6075501	95/4905	1082/3918	4325/675	5000/0	5000/0
6075595	8/4992	2534/2466	4675/325	5000/0	5000/0
6075773	92/4908	1546/3454	4743/257	5000/0	5000/0
6076067	15/4985	1600/3400	4823/177	5000/0	5000/0
6076187	9/4991	1393/3607	4514/486	5000/0	5000/0
6076417	29/4971	1758/3242	4400/600	5000/0	5000/0
6076529	91/4909	3843/1157	4779/221	5000/0	5000/0
6076634	32/4968	2325/2675	4669/331	5000/0	5000/0
6077156	16/4984	1992/3008	4616/384	5000/0	5000/0
6077183	87/4913	752/4248	4759/241	5000/0	5000/0
6077194	21/4979	1546/3454	4344/656	5000/0	5000/0
6077236	8/4992	1919/3081	4843/157	5000/0	5000/0
6077292	76/4924	2389/2611	4446/554	5000/0	5000/0
6077313	96/4904	1334/3666	4842/158	5000/0	5000/0
6077411	25/4975	961/4039	4737/263	5000/0	5000/0
6077727	10/4990	1498/3502	4482/518	5000/0	5000/0
6077855	38/4962	1726/3274	4607/393	5000/0	5000/0

Tabelle D.6: Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung).

Kampagne	0.02%	0.1%	1%	10%	100%
6077859	77/4923	328/4672	4699/301	5000/0	5000/0
6077952	35/4965	817/4183	4781/219	5000/0	5000/0
6077975	77/4923	1848/3152	4747/253	5000/0	5000/0
6078048	78/4922	492/4508	4325/675	5000/0	5000/0
6078088	4/4996	2679/2321	4674/326	5000/0	5000/0
6078405	30/4970	1577/3423	4744/256	5000/0	5000/0
6078601	7/4993	388/4612	4586/414	5000/0	5000/0
6078790	26/4974	2621/2379	4687/313	5000/0	5000/0
6078856	83/4917	2428/2572	4494/506	5000/0	5000/0
6079259	28/4972	1537/3463	4639/361	5000/0	5000/0
6079469	75/4925	2581/2419	4818/182	5000/0	5000/0
6079658	77/4923	1344/3656	4758/242	5000/0	5000/0
6079728	15/4985	1044/3956	4665/335	5000/0	5000/0
6079735	73/4927	2745/2255	4750/250	5000/0	5000/0
6080093	84/4916	2741/2259	4572/428	5000/0	5000/0
6080141	93/4907	2138/2862	4750/250	5000/0	5000/0
6080289	90/4910	1834/3166	4252/748	5000/0	5000/0
6080357	10/4990	1088/3912	4223/777	5000/0	5000/0
6080573	32/4968	2175/2825	4573/427	5000/0	5000/0
6080642	8/4992	2570/2430	4673/327	5000/0	5000/0
6081017	9/4991	1617/3383	4594/406	5000/0	5000/0
6081125	87/4913	1523/3477	4770/230	5000/0	5000/0
6081423	84/4916	1587/3413	4378/622	5000/0	5000/0
6081584	17/4983	785/4215	4520/480	5000/0	5000/0
6081671	84/4916	2059/2941	4820/180	5000/0	5000/0
6081740	83/4917	2434/2566	4742/258	5000/0	5000/0
6082048	88/4912	3304/1696	4096/904	5000/0	5000/0
6082071	19/4981	955/4045	4829/171	5000/0	5000/0
6082494	22/4978	1731/3269	4740/260	5000/0	5000/0
6082510	80/4920	587/4413	4610/390	5000/0	5000/0
6082678	85/4915	1607/3393	4918/82	5000/0	5000/0
6082707	28/4972	1527/3473	4766/234	5000/0	5000/0
6082842	83/4917	684/4316	4846/154	5000/0	5000/0
6082875	25/4975	1510/3490	4448/552	5000/0	5000/0
6082979	95/4905	248/4752	4355/645	5000/0	5000/0
6083597	68/4932	1864/3136	4597/403	5000/0	5000/0
6083700	79/4921	1177/3823	4754/246	5000/0	5000/0
6083785	90/4910	2607/2393	4511/489	5000/0	5000/0
6083974	67/4933	2298/2702	4924/76	5000/0	5000/0
6084081	34/4966	1431/3569	4488/512	5000/0	5000/0

Tabelle D.7: Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung).

Kampagne	0.02%	0.1%	1%	10%	100%
6084218	90/4910	1212/3788	4660/340	5000/0	5000/0
6084308	7/4993	489/4511	4828/172	5000/0	5000/0
6084644	30/4970	743/4257	4829/171	5000/0	5000/0
6085219	20/4980	1246/3754	4596/404	5000/0	5000/0
6085400	5/4995	1769/3231	4660/340	5000/0	5000/0
6085535	84/4916	2590/2410	4295/705	5000/0	5000/0
6085771	5/4995	1030/3970	4352/648	5000/0	5000/0
6085971	22/4978	1435/3565	5000/0	5000/0	5000/0
6085982	9/4991	2779/2221	4689/311	5000/0	5000/0
6086935	13/4987	3478/1522	4434/566	0/0	5000/0
6087879	5/4995	663/4337	4487/513	5000/0	5000/0
6087912	26/4974	370/4630	4505/495	5000/0	5000/0
6088142	92/4908	1300/3700	4628/372	5000/0	5000/0
6088352	7/4993	2010/2990	4753/247	5000/0	5000/0
6088669	30/4970	494/4506	4919/81	5000/0	5000/0
6088995	68/4932	1623/3377	4446/554	5000/0	5000/0
6089000	84/4916	1019/3981	4592/408	5000/0	5000/0
6089241	7/4993	1819/3181	4670/330	5000/0	5000/0
6089514	106/4894	2170/2830	4822/178	5000/0	5000/0
6089559	28/4972	1944/3056	4525/475	5000/0	5000/0
6089682	93/4907	2146/2854	4668/332	5000/0	5000/0
6089767	27/4973	1841/3159	4667/333	5000/0	5000/0
6089962	25/4975	1489/3511	4497/503	5000/0	5000/0
6090110	72/4928	837/4163	4676/324	5000/0	5000/0
6090145	106/4894	2429/2571	4809/191	5000/0	5000/0
6090467	94/4906	947/4053	4530/470	5000/0	5000/0
6091236	21/4979	2729/2271	4594/406	5000/0	5000/0
6091603	73/4927	412/4588	4719/281	5000/0	5000/0
6091831	70/4930	1713/3287	4572/428	5000/0	5000/0
6092572	87/4913	2063/2937	4732/268	5000/0	5000/0
6092648	62/4938	2095/2905	4578/422	5000/0	5000/0
6093237	98/4902	746/4254	4683/317	5000/0	5000/0
6093273	91/4909	1743/3257	4566/434	5000/0	5000/0
6093472	9/4991	1909/3091	4680/320	5000/0	5000/0
6093487	9/4991	1504/3496	4769/231	5000/0	5000/0
6093550	86/4914	4181/819	4760/240	5000/0	5000/0
6093649	83/4917	905/4095	4786/214	5000/0	5000/0
6093887	21/4979	1706/3294	4913/87	5000/0	5000/0
6094027	79/4921	1408/3592	4628/372	5000/0	5000/0
6094215	92/4908	1936/3064	4459/541	5000/0	5000/0

Tabelle D.8: Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung).

Kampagne	0.02%	0.1%	1%	10%	100%
6094411	5/4995	3003/1997	4905/95	5000/0	5000/0
6094433	85/4915	3397/1603	4000/1000	5000/0	5000/0
6095198	5/4995	2528/2472	4526/474	5000/0	5000/0
6095665	9/4991	1621/3379	4612/388	5000/0	5000/0
6095691	85/4915	3204/1796	4735/265	5000/0	5000/0
6095898	65/4935	557/4443	4576/424	5000/0	5000/0
6095999	26/4974	1876/3124	4760/240	5000/0	5000/0
6096286	10/4990	1097/3903	4817/183	5000/0	5000/0
6096379	78/4922	834/4166	4781/219	5000/0	5000/0
6096386	91/4909	2647/2353	4438/562	5000/0	5000/0
6096807	90/4910	3380/1620	4659/341	5000/0	5000/0
6096863	73/4927	2070/2930	4830/170	5000/0	5000/0
6096899	25/4975	1455/3545	4461/539	5000/0	5000/0
6097256	92/4908	408/4592	4116/884	5000/0	5000/0
6097977	79/4921	487/4513	4304/696	5000/0	5000/0
6098488	87/4913	2056/2944	4510/490	5000/0	5000/0
6098580	92/4908	2295/2705	4595/405	5000/0	5000/0
6098979	86/4914	2319/2681	4768/232	5000/0	5000/0
6099232	85/4915	177/4823	4348/652	5000/0	5000/0
6099314	81/4919	1466/3534	5000/0	5000/0	5000/0
6100024	79/4921	1171/3829	4595/405	5000/0	5000/0
6100213	34/4966	1536/3464	4538/462	5000/0	5000/0
6100543	7/4993	2317/2683	4838/162	5000/0	5000/0
6101075	98/4902	2403/2597	4612/388	5000/0	5000/0
6101364	79/4921	906/4094	4830/170	5000/0	5000/0
6101530	22/4978	2291/2709	4935/65	5000/0	5000/0
6101683	11/4989	698/4302	4720/280	5000/0	5000/0
6101735	34/4966	764/4236	4925/75	5000/0	5000/0
6101888	7/4993	385/4615	4335/665	5000/0	5000/0
6102043	91/4909	969/4031	4553/447	5000/0	5000/0
6102061	93/4907	1654/3346	4864/136	5000/0	5000/0
6102446	93/4907	2830/2170	4660/340	5000/0	5000/0
6102537	91/4909	2884/2116	4831/169	5000/0	5000/0
6102590	75/4925	954/4046	4766/234	5000/0	5000/0
6102978	11/4989	2178/2822	5000/0	5000/0	5000/0
6103061	9/4991	181/4819	3977/1023	5000/0	5000/0
6103076	22/4978	1193/3807	4275/725	5000/0	5000/0
6103318	24/4976	1998/3002	4330/670	5000/0	5000/0
6103619	5/4995	1380/3620	4907/93	5000/0	5000/0
6103698	103/4897	745/4255	4831/169	5000/0	5000/0

Tabelle D.9: Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung).

Kampagne	0.02%	0.1%	1%	10%	100%
6103757	16/4984	1719/3281	4273/727	5000/0	5000/0
6103957	6/4994	2389/2611	4842/158	5000/0	5000/0
6104836	73/4927	2135/2865	4248/752	5000/0	5000/0
6104924	82/4918	3132/1868	4586/414	5000/0	5000/0
6104987	35/4965	2943/2057	4909/91	5000/0	5000/0
6105082	37/4963	1788/3212	4444/556	5000/0	5000/0
6105131	15/4985	3024/1976	5000/0	5000/0	5000/0
6105987	38/4962	1027/3973	4825/175	5000/0	5000/0
6106314	24/4976	513/4487	4907/93	5000/0	5000/0
6106842	92/4908	535/4465	4386/614	5000/0	5000/0
6106983	93/4907	1029/3971	4600/400	5000/0	5000/0
6107022	27/4973	787/4213	4336/664	5000/0	5000/0
6107793	21/4979	1315/3685	4413/587	5000/0	5000/0
6107856	84/4916	2190/2810	4428/572	5000/0	5000/0
6107946	33/4967	2122/2878	4677/323	5000/0	5000/0
6108786	15/4985	978/4022	4597/403	5000/0	5000/0
6108788	12/4988	735/4265	4589/411	5000/0	5000/0
6108823	82/4918	1633/3367	4365/635	5000/0	5000/0
6108913	7/4993	1533/3467	4521/479	5000/0	5000/0
6108962	25/4975	1196/3804	4663/337	5000/0	5000/0
6109276	74/4926	955/4045	4747/253	5000/0	5000/0
6110255	8/4992	2396/2604	4492/508	5000/0	5000/0
6110285	8/4992	1695/3305	4568/432	5000/0	5000/0
6110295	22/4978	293/4707	4514/486	5000/0	5000/0
6110332	22/4978	2165/2835	5000/0	5000/0	5000/0
6110394	8/4992	1032/3968	4907/93	5000/0	5000/0
6110581	92/4908	4165/835	4773/227	5000/0	5000/0
6110775	77/4923	1303/3697	4909/91	5000/0	5000/0
6111642	91/4909	2425/2575	4595/405	5000/0	5000/0
6111672	101/4899	1924/3076	4467/533	5000/0	5000/0
6112069	12/4988	3063/1937	4251/749	5000/0	5000/0
6112163	81/4919	2625/2375	4831/169	5000/0	5000/0
6112363	6/4994	2454/2546	4902/98	5000/0	5000/0
6112565	30/4970	1224/3776	4332/668	5000/0	5000/0
6112850	88/4912	677/4323	4845/155	5000/0	5000/0
6112995	33/4967	398/4602	4765/235	5000/0	5000/0
6113583	77/4923	1461/3539	4127/873	5000/0	5000/0
6113660	18/4982	1975/3025	4903/97	5000/0	5000/0
6113743	5/4995	2022/2978	4569/431	5000/0	5000/0
6114231	69/4931	3259/1741	4340/660	5000/0	5000/0

Tabelle D.10: Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung).

Kampagne	0.02%	0.1%	1%	10%	100%
6114292	77/4923	1611/3389	4769/231	5000/0	5000/0
6114323	80/4920	791/4209	4195/805	5000/0	5000/0
6114492	7/4993	1241/3759	4508/492	5000/0	5000/0
6114547	29/4971	1518/3482	4592/408	5000/0	5000/0
6115201	79/4921	502/4498	4835/165	5000/0	5000/0
6115654	81/4919	1149/3851	4538/462	5000/0	5000/0
6116030	66/4934	2556/2444	4579/421	5000/0	5000/0
6116627	17/4983	1762/3238	4486/514	5000/0	5000/0
6116769	86/4914	1414/3586	4613/387	5000/0	5000/0
6116991	84/4916	1403/3597	4747/253	5000/0	5000/0
6117066	76/4924	3001/1999	4757/243	5000/0	5000/0
6117131	79/4921	1896/3104	4677/323	5000/0	5000/0
6117318	6/4994	1171/3829	4491/509	5000/0	5000/0
6117615	23/4977	1112/3888	4575/425	5000/0	5000/0
6117763	17/4983	652/4348	4450/550	5000/0	5000/0
6117893	19/4981	1027/3973	4916/84	5000/0	5000/0
6117986	30/4970	756/4244	4759/241	5000/0	5000/0
6118008	70/4930	462/4538	4902/98	5000/0	5000/0
6118666	109/4891	2901/2099	4759/241	5000/0	5000/0
6119045	88/4912	1714/3286	4420/580	5000/0	5000/0
6119156	26/4974	1098/3902	4309/691	5000/0	5000/0
6119219	88/4912	3000/2000	4552/448	5000/0	5000/0
6119492	102/4898	2067/2933	4453/547	5000/0	5000/0
6119624	72/4928	2062/2938	4687/313	5000/0	5000/0
6120878	86/4914	676/4324	4763/237	5000/0	5000/0
6120921	11/4989	568/4432	4496/504	5000/0	5000/0
6120957	79/4921	666/4334	4922/78	5000/0	5000/0
6121049	42/4958	732/4268	4581/419	5000/0	5000/0
6121501	85/4915	425/4575	4395/605	5000/0	5000/0
6121586	37/4963	1058/3942	4671/329	5000/0	5000/0
6121699	82/4918	2938/2062	4926/74	5000/0	5000/0
6121781	25/4975	1480/3520	4759/241	5000/0	5000/0
6121857	10/4990	1774/3226	4919/81	5000/0	5000/0
6121913	6/4994	1431/3569	4303/697	5000/0	5000/0
6133213	31/4969	924/4076	4894/106	5000/0	5000/0
6133222	109/4891	2253/2747	5000/0	5000/0	5000/0
6133511	33/4967	430/4570	5000/0	5000/0	5000/0
6134014	97/4903	891/4109	5000/0	5000/0	5000/0
6155327	109/4891	1205/3795	5000/0	5000/0	5000/0
6155376	93/4907	2878/2122	4840/160	5000/0	5000/0

Tabelle D.11: Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung).

Kampagne	0.02%	0.1%	1%	10%	100%
6155404	26/4974	1786/3214	4900/100	5000/0	5000/0
6155586	26/4974	1572/3428	4790/210	5000/0	5000/0
6155754	102/4898	2967/2033	4802/198	5000/0	5000/0
6156007	9/4991	1458/3542	4512/488	5000/0	5000/0
6156132	89/4911	2056/2944	4382/618	5000/0	5000/0
6156140	85/4915	1098/3902	4890/110	5000/0	5000/0
6156153	105/4895	695/4305	4701/299	5000/0	5000/0
6156251	10/4990	1042/3958	4684/316	5000/0	5000/0
6156300	5/4995	1470/3530	4499/501	5000/0	5000/0
6156880	99/4901	3472/1528	4597/403	5000/0	5000/0
6156930	108/4892	1680/3320	4716/284	5000/0	5000/0
6157084	7/4993	1341/3659	4730/270	5000/0	5000/0
6157120	27/4973	4066/934	4407/593	5000/0	5000/0
6157277	5/4995	1077/3923	4615/385	5000/0	5000/0
6157287	22/4978	1186/3814	4819/181	5000/0	5000/0
6157483	8/4992	403/4597	4426/574	5000/0	5000/0
6157658	9/4991	1405/3595	4910/90	5000/0	5000/0
6158078	89/4911	2899/2101	4527/473	5000/0	5000/0
6158687	117/4883	2584/2416	4696/304	5000/0	5000/0
6159492	30/4970	1049/3951	4597/403	5000/0	5000/0

Tabelle D.12: Vollständige Filterergebnisse des STOM-Botnetz (Fortsetzung).

APPENDIX E

Filtern mit kontextfreien Grammatiken

Für das GHEG-Botnetz wurden basierend auf 0,001 Prozent, 0,01 Prozent, 0,1 Prozent, einem Prozent, 10 Prozent und 99,6 Prozent der Nachrichten jeweils 10 kontextfreie Filter erzeugt, mit denen anschließend alle 255.622 aus dem Botnetz abgefangenen Nachrichten gefiltert wurden. Die Anzahl pro Filter gelernter Wörter und Sätze und die Filterergebnisse sind in den Tabellen E.1 bis E.6 dargestellt.

Für das SRIZBI-Botnetz wurden kontextfreie Filter auf 0,001 Prozent, 0,01 Prozent, 0,1 Prozent, einem Prozent und 10 Prozent der Nachrichten erzeugt. Diese wurden anschließend auf den eine Millionen Nachrichten umfassenden Datensatz angewendet. Tabelle E.7 zeigt die Filterergebnisse der einzelnen Filter, wobei jeweils die Anzahl von dem Filter erkannter und nicht erkannter Nachrichten angegeben ist.

	Filterdaten		**Filterergebnisse**	
Trainingslauf	*# Wörter*	*# Sätze*	*enthalten*	*nicht enthalten*
1	155	39	228.576	27.046
2	153	41	228.723	26.899
3	153	40	241.999	13.623
4	153	39	241.999	13.623
5	153	41	228.576	27.046
6	147	39	241.999	13.623
7	150	39	241.999	13.623
8	150	40	241.999	13.623
9	156	41	242.182	13.440
10	147	39	241.999	13.623
∅	151,7	39,8	238.005,1 (93,108%)	17.616,9 (6,892%)

Tabelle E.1: Filterdaten und -Ergebnisse für das GHEG-Botnetz bei einer Filtergenerierung auf 3 zufällig ausgewählten Spam-Nachrichtten.

	Filterdaten		Filterergebnisse	
Trainingslauf	*# Wörter*	*# Sätze*	*enthalten*	*nicht enthalten*
1	192	58	242.182	13.440
2	177	51	242.182	13.440
3	190	60	255.605	17
4	193	57	228.576	27.046
5	199	63	255.605	17
6	186	55	241.999	13.623
7	196	62	255.605	17
8	189	59	255.605	17
9	187	60	255.605	17
10	189	59	255.605	17
∅	189,8	58,4	248.856,9 (97,353%)	6.765,1 (2,647%)

Tabelle E.2: Filterdaten und -Ergebnisse für das GHEG-Botnetz bei einer Filtergenerierung auf 26 zufällig ausgewählten Spam-Nachrichtten.

	Filterdaten		Filterergebnisse	
Trainingslauf	*# Wörter*	*# Sätze*	*enthalten*	*nicht enthalten*
1	202	65	255.605	17
2	202	65	255.605	17
3	202	65	255.605	17
4	202	65	255.605	17
5	202	65	255.605	17
6	202	65	255.605	17
7	202	65	255.605	17
8	202	65	255.605	17
9	202	65	255.605	17
10	202	65	255.605	17
∅	202	65	255.605 (99,993%)	17 (0,007%)

Tabelle E.3: Filterdaten und -Ergebnisse für das GHEG-Botnetz bei einer Filtergenerierung auf 256 zufällig ausgewählten Spam-Nachrichtten.

	Filterdaten		Filterergebnisse	
Trainingslauf	*# Wörter*	*# Sätze*	*enthalten*	*nicht enthalten*
1	202	65	255.605	17
2	202	65	255.605	17
3	205	68	255.606	16
4	202	65	255.605	17
5	202	65	255.605	17
6	202	65	255.605	17
7	202	65	255.605	17
8	187	57	242.329	13.293
9	184	58	242.329	13.293
10	195	61	241.999	13.623
⌀	198,3	63,4	251.589,3 (98,422%)	4.032,7 (1,578%)

Tabelle E.4: Filterdaten und -Ergebnisse für das GHEG-Botnetz bei einer Filtergenerierung auf 2.556 zufällig ausgewählten Spam-Nachrichtten.

	Filterdaten		Filterergebnisse	
Trainingslauf	*# Wörter*	*# Sätze*	*enthalten*	*nicht enthalten*
1	202	65	255.605	17
2	202	65	255.605	17
3	204	67	255.605	17
4	202	65	255.605	17
5	202	65	255.605	17
6	202	65	255.605	17
7	203	66	255.605	17
⌀	202,4	65,4	255.605 (99,993%)	17 (0,007%)

Tabelle E.5: Filterdaten und -Ergebnisse für das GHEG-Botnetz bei einer Filtergenerierung auf 25.459 zufällig ausgewählten Spam-Nachrichtten.

	Filterdaten		Filterergebnisse	
Trainingslauf	*# Wörter*	*# Sätze*	*enthalten*	*nicht enthalten*
1	208	71	255.606	16
2	208	71	255.606	16
3	208	71	255.606	16
4	208	71	255.606	16
5	208	71	255.606	16
6	208	71	255.606	16
7	208	71	255.606	16
8	208	71	255.606	16
9	208	71	255.606	16
10	208	71	255.606	16
⌀	208	71	255.606 (99,994%)	16 (0,006%)

Tabelle E.6: Filterdaten und -Ergebnisse für das GHEG-Botnetz bei einer Filtergenerierung auf 254.591 zufällig ausgewählten Spam-Nachrichtten.

	Erkennungsrate in Prozent				
	0,001%	*0,01%*	*0,1%*	*1,0%*	*10%*
Filter 1	300.212 / 699.788	335.694 / 664.306	335.694 / 664.306	335.694 / 664.306	335.694 / 664.306
Filter 2	209.604 / 790.396	235.624 / 764.376	235.624 / 764.376	235.624 / 764.376	235.624 / 764.376
Filter 3	14.856 / 985.144	178.100 / 821.900	178.100 / 821.900	178.100 / 821.900	178.100 / 821.900
Filter 4	35.043 / 964.957	30.577 / 969.423	35.043 / 964.957	35.043 / 964.957	35.043 / 964.957
Filter 5	7.677 / 992.323	-	6.714 / 993.286	7.677 / 992.323	7.677 / 992.323
Filter 6	-	113.810 / 886.190	113.810 / 886.190	113.810 / 886.190	113.810 / 886.190
Filter 7	-	30.974 / 969.026	30.974 / 969.026	30.974 / 969.026	30.974 / 969.026
Filter 8	-	20.394 / 979.606	20.394 / 979.606	20.394 / 979.606	20.394 / 979.606
Filter 9	-	17.424 / 982.576	17.424 / 982.576	17.424 / 982.576	17.424 / 982.576
Filter 10	-	-	15.136 / 984.864	15.136 / 984.864	15.136 / 984.864
Filter 11	-	10.124 / 989.876	10.124 / 989.876	10.124 / 989.876	10.124 / 989.876
Filter 12	-	-	11.948 / 988.052	20.384 / 979.616	47.296 / 952.704
Filter 13	-	-	12.929 / 987.071	-	12.308 / 987.692

Tabelle E.7: Erkennungsrate der für den SRIZBI Spamdatensatz generierte Filter. Als 100 Prozent Bezugsgröße wurden die Filterergebnisse der aus 10% der Nachrichten erstellten Filter herangezogen. (* Da das Filterergebnis des auf einem Prozent der Nachrichten generierten Filters besser ist, wurde dieser als 100 Prozent Bezugsgröße verwendet.)