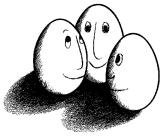
**Diplomarbeit**

# Analyse von Betriebssystem-Logdateien

Andrea Matuszewski



Diplomarbeit
am Fachbereich Informatik
der Technischen Universität Dortmund

Dortmund, 17. September 2009

**Betreuer:**

Prof. Dr. Katharina Morik
Dipl.-Inform. Christian Bockermann

## Danksagung

An dieser Stelle möchte ich mich zunächst bei Prof. Dr. Katharina Morik und Dipl.-Inform. Christian Bockermann für die Betreuung dieser Diplomarbeit und die Ermutigungen während dieser Zeit bedanken.
Ein besonderer Dank gilt zudem meinen Eltern, die mich in den Jahren meines Studiums immer unterstützt haben. Vielen Dank auch an meine Freunde und meine Schwester, die während dieser Diplomarbeit viel Verständnis gezeigt und Rücksicht genommen haben.
Zu guter Letzt möchte ich mich bei meinen Korrekturlesern für die investierte Mühe und die Zeit bedanken.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# 1. Einleitung

Die wachsende Nutzung von Computersystemen und die zunehmende Anzahl an Breitbandzugängen zum Internet ermöglicht eine immer schneller werdende Verbreitung von Viren, Würmern und anderen Schadprogrammen. Die Zahl der Anwender, deren System bereits zum Ziel eines Angriffs wurde, steigt. Laut heise Security [Hei08] "sind fast vier Millionen Deutsche schon einmal Opfer von Computer–Kriminalität geworden."

Die Verbreitung von Malware erfolgt mittlerweile auf vielfältige Art und Weise und die Anzahl der Schadprogramme steigt. Abbildung 1.1 zeigt die von IT–Sicherheitsunternehmen und verschiedenen Virenschutzprogrammen jährlich neu gefundenen Computerschädlinge. Trotz zum Teil unterschiedlicher Definitionen und Zählweisen ist die Vervielfachung der gefundenen Malware im Jahr 2007 im Vergleich zum Jahr 2006 deutlich erkennbar [BSI08]. Die Securitysoftware des Herstellers Kaspersky entdeckte laut [GGM08] im Jahr 2008 zwischen Januar und Juni 367.772 neue Schadprogramme. Das sind 2,9 Mal so viele wie in der zweiten Hälfte des Jahres 2007. Der Trend einer zunehmenden Anzahl von Schadprogrammen scheint sich demnach auch weiterhin fortzusetzen.

Durch die Vielzahl verschiedener Schadprogramme und deren kontinuierliche Veränderung und Weiterentwicklung ist es schwierig, einen Überblick über die aktuelle Bedrohung zu bewahren. Zur Ergreifung passender Gegenmaßnahmen muss jedoch bekannt sein, welchen Schaden eine Malware verursachen kann. Die vorliegende Diplomarbeit befasst sich daher mit der automatischen Erkennung von Schadprogramm–Familien. Diese Familien stellen Gruppen von Malware mit ähnlichen Verhaltensweisen dar.

Der folgende Abschnitt 1.1 liefert eine Motivation für den Einsatz von Systemen zur automatischen Angriffserkennung. Anschließend werden in Abschnitt 1.2 die Ziele dieser Arbeit erläutert, bevor in Abschnitt 1.3 der Aufbau der Arbeit beschrieben wird.

## 1.1. Motivation: Einsatz von Intrusion Detection Systemen

Mit dem Trend einer zunehmenden Anzahl von Schadprogrammen wächst auch der Bedarf an Methoden und Applikationen zum Schutz der Sicherheit. Die Wahrung der Vertraulichkeit, der Integrität und der Verfügbarkeit eines Computersystems ist entscheidend für seine Nutzung und Akzeptanz [Bis02]. Die immer größer werdende Menge der zu beobachtenden Daten und der zu analysierenden Informationen zwingt Netzwerkadministratoren und Sicherheitsanalysten dazu, unterstützende Programme einzusetzen, zu denen auch Intrusion Detection Systeme gehören.

*Intrusion Detection* bezeichnet den Prozess der Erkennung von Ereignissen, die Sicherheitsbestimmungen missachten oder zu einer Verletzung dieser führen könnten. Derart bösartiges Verhalten kann seinen Ursprung in Schadprogrammen (z.B. Viren, Würmern, Trojanern etc.), unautorisierten Zugriffsversuchen von Angreifern aus dem Internet oder Privilegienmissbrauch autorisierter Nutzer haben.

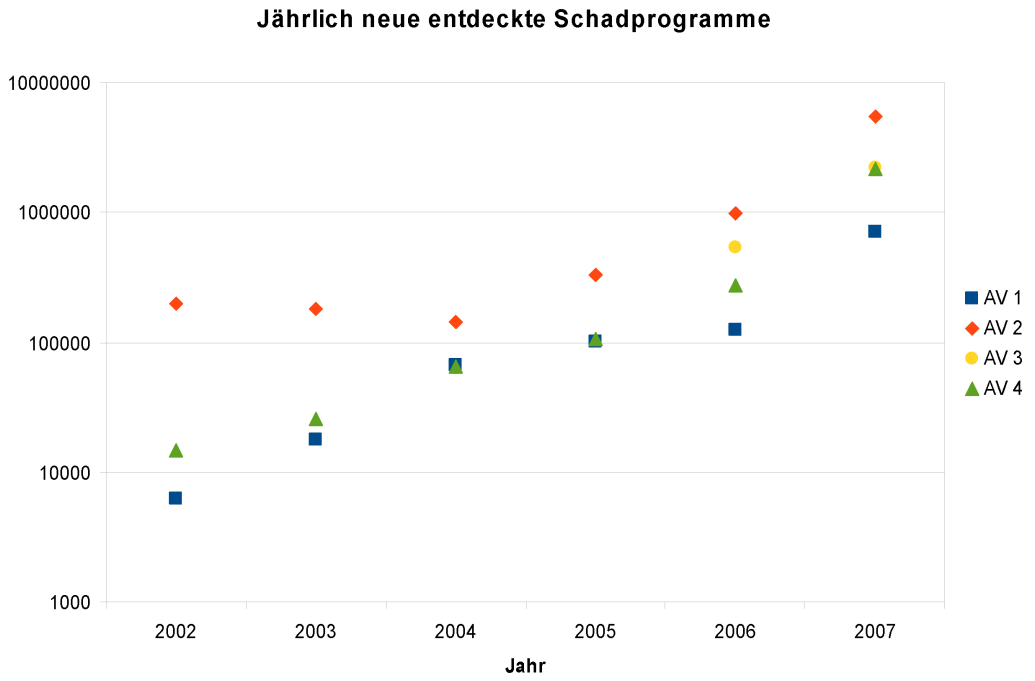



Abbildung 1.1.: Anzahl der jährlich neu gefundenen Schadprogramme verschiedener Quellen (aus [BSI08]).

Ein *Intrusion Detection System* (kurz IDS) ist eine Software zur Automatisierung des Intrusion Detection Prozesses. IDS-Module können auf einer Firewall aufsetzen. Eine Firewall analysiert den durch sie hindurch laufenden Datenverkehr. Sie entscheidet anhand festgelegter Regeln, ob bestimmte Netzwerkpakete zulässig sind oder nicht, um das Netzwerk vor unerlaubten Zugriffen zu schützen. Die Funktion einer Firewall besteht im Gegensatz zu einem Intrusion Detection System nicht darin, bereits stattgefundene Angriffe zu erkennen, sondern lediglich darin, Kommunikationsbeziehungen zu erlauben oder abzulehnen. Ein IDS hingegen bewertet ein stattgefundenes Ereignis und erzeugt bei der Erkennung eines Angriffs eine Warnung. Das IDS kann als eine Art Alarmanlage verstanden werden, die bereits vorhandene Sicherheitsstrukturen wie Firewalls ergänzt oder unterstützt. Das IDS protokolliert folglich Angriffe auf das Computersystem, die von einer konventionellen Firewall nicht erkannt werden können.

## 1.2. Ziele der Arbeit

Die meist genutzten Programme zur Erkennung von Angriffen auf Computersysteme sind Virenscanner. Derartige Software kann als Intrusion Detection System aufgefasst werden. Die bekannten Hersteller solcher Antivirussoftware realisieren in erster Linie signaturbasierte Systeme. Dabei erfolgt eine statische Binärcodeanalyse der ausführbaren Schadprogramme zur Erstellung von Signaturen. Die Muster zukünftiger Ereignisse können mit den Signaturen verglichen werden, so dass das erneute Auftreten eines Schadprogramms

erkannt wird.

Zur Signaturerstellung wird der Binärcode der Malware für gewöhnlich zunächst in Assembler–Code konvertiert [KAW94]. Ein typisch erscheinender, charakteristischer Teil des Codes wird extrahiert und die entsprechenden Bytes werden im Maschinencode identifiziert. Sie liefern die Signatur. Eine neue Signatur wird getestet, um sicher zu stellen, dass der extrahierte Code in keinem anderen gängigen ("gutartigen") Software–Produkt enthalten ist, da das IDS sonst Fehlalarme erzeugen würde. Getestete Signaturen werden in einer Datenbank gespeichert.

Probleme dieses Ansatzes bestehen in der häufig aufwendigen und teilweise manuell durchzuführenden Signaturerstellung und den stetig wachsenden Signaturdatenbanken. Zudem werden eigentlich bereits identifizierte Schadprogramme nicht mehr automatisch erkannt, sobald ihr Code geändert wird (Code–Morphing). Dabei kann die Funktionalität von der Code–Änderung unberührt bleiben, die Signaturen stimmen jedoch nicht mehr überein. Der Antivirussoftware–Hersteller Symantec[1] hat bereits bei dem Vergleich der Jahre 2006 und 2007 festgestellt, dass immer schneller Varianten bekannter Schädlinge gefunden werden [TFJ+08]. Das bedeutet für die Hersteller von Antivirussoftware, dass laufend eine Vielzahl neuer Signaturen erstellt werden muss.

Die vorliegende Diplomarbeit knüpft hier an und bewertet verschiedene Data Mining Techniken zur automatischen Klassifikation von Schadprogrammen und ihren Varianten. Es wird versucht, Malware aufgrund ähnlicher Verhaltensweisen Gruppen zuzuordnen. Als Datengrundlage zur Analyse dienen die Aufrufe von Betriebssystem–Funktionen, so genannte Systemcalls, die ein Schadprogramm während seiner Ausführung nutzt. Diese Aufrufe und die dabei übergebenen Parameter wurden protokolliert und stehen in der Form von Systemcall–Logdateien zur Verfügung. Das Ziel ist also die Extraktion charakteristischer Eigenschaften auf der Grundlage dieser Daten, um die verschiedenen Gruppen, die so genannten Schadprogramm–Familien, voneinander unterscheiden zu können. Durch die Anwendung von Klassifikationsverfahren sollen Modelle erstellt werden, die die Zuordnung von Varianten bereits bekannter Schadprogramme ermöglichen und automatisieren. So kann eine explizite Signaturerstellung und somit die Vorhaltung einer großen Signatur–Datenbank vermieden werden. Durch die Analyse von Schadprogrammen auf der Systemcall–Ebene ist Code–Morphing kein Problem mehr, da Programme zur Ausführung gleicher Funktionen auch ähnliche Systemfunktionen oder Abfolgen dieser nutzen müssen.

Eine "Familien–Zuordnung" von Schadprogrammen ist zudem wichtig, um den Überblick über die aktuelle Bedrohungen zu bewahren. Die Securitysoftware–Hersteller erstellen in regelmäßigen Abständen Reports über die Anzahl und die Art der gefundenen Schadprogramme, um aktuelle Risiken erkennen und adäquat agieren zu können. Herstellern beliebiger Software–Produkte helfen solche Schadprogramm–Reports dabei, Schwachstellen ihrer Programme zu identifizieren. Nur durch die Kenntnis der Schwachstellen sind sie dazu in der Lage Sicherheitslücken zu schließen und erfolgreiche Angriffe zukünftig zu vermeiden.

[1] http://www.symantec.com

## 1.3. Aufbau der Arbeit

Das folgende Kapitel 2 gibt zunächst eine Einführung in den Bereich der Intrusion Detection. Der Begriff der "Intrusion" wird definiert und es erfolgt eine Beschreibung verschiedener Kategorien von Intrusion Detection Systemen. Zwei grundlegende Herangehensweisen zur Erkennung von Schadprogrammen werden erläutert.

In Kapitel 3 werden einige Arbeiten zur Angriffserkennung und zur Klassifikation von Schadprogramm–Familien vorgestellt. Es wird ein Überblick über den Bereich des Data Minings und über die Nutzung von Systemcalls als Datengrundlage gegeben.

Kapitel 4 liefert die theoretischen Grundlagen einer Auswahl überwachter Lernverfahren, die in dieser Arbeit verwendet werden.

In Kapitel 5 werden die Systemcall–Logdateien beschrieben, die für die Analysen in dieser Diplomarbeit zur Verfügung stehen. Die notwendigen Konvertierungen der Logdateien werden erläutert und die durchgeführten Experimente werden dokumentiert.

Schließlich fasst Kapitel 6 die Ergebnisse der Arbeit zusammen. Es wird ein Ausblick auf mögliche Erweiterungen und Verbesserungen der vorgestellten Ansätze gegeben und ein Fazit gezogen.

# 2. Intrusion Detection

Intrusion-Detection-Systeme (kurz IDS) sollen Angriffe und Sicherheitsverletzungen zeitnah erkennen und dienen dazu, die Integrität und Verfügbarkeit von Computersystemen und Diensten zu erhöhen. Sie überwachen verschiedene Aktivitäten, um Schäden am System zu verhindern, zu begrenzen oder zu beheben. Laut [BSI02] ermöglicht die Erkennung unerwünschter Verhaltensweisen mit Hilfe eines IDS zudem in vielen Fällen die Verbesserung von Systemkonfigurationen, da Schwachstellen aufgedeckt werden können.

Das vorliegende Kapitel dient der Einführung in den Bereich der Intrusion Detection Systeme. Dazu werden in Abschnitt 2.1 zunächst einige Begriffe erläutert, die in der Angriffserkennung allgemein gebräuchlich sind und im weiteren Verlauf dieser Arbeit verwendet werden.

Abschnitt 2.2 gibt einen kurzen Überblick über die grundlegenden Konzepte von Intrusion Detection Systemen. Es werden netzwerkbasierte und hostbasierte IDS vorgestellt, die zur Erkennung von Schadprogrammen jeweils an unterschiedlichen Stellen des zu schützenden Systems eingesetzt werden. Im Anschluss daran werden die signaturbasierte Angriffserkennung und die Anomalieerkennung erläutert.

## 2.1. Begrifflichkeiten im Bereich der Intrusion Detection

Die Definition einer Intrusion ist nicht ganz einfach, da unter diesem Begriff eine Vielzahl von Aktivitäten und Ereignissen zusammengefasst wird. Das Wort "Intrusion" kann im Deutschen übersetzt werden als "Angriff", "Einbruch", "Eindringen" oder "Verletzung". Die Definition der Intrusion Detection Sub Group (IDSG) des National Security Telecommunication Advisory Council (NSTAC) lautet wie folgt:

*"Eine Intrusion ist definiert als unautorisierter Zugriff auf, und/oder Aktivität in einem Informationssystem."* [IDS97]

Heady et. al. definieren eine Intrusion als

*"Jede Menge von Aktionen die versucht, die Integrität, Vertraulichkeit oder Verfügbarkeit eines Betriebsmittels zu beeinträchtigen."* [HLSM90]

In [Bis02] betrachtet Matt Bishop Integrität, Vertraulichkeit und Verfügbarkeit ebenfalls als die Basiskomponenten der Computersicherheit. *Integrität* beschreibt die Vertrauenswürdigkeit von Daten oder Ressourcen. Der Begriff wird üblicherweise im Zusammenhang mit der Verhinderung missbräuchlicher oder unautorisierter Änderungen verwendet und umfasst die Datenintegrität, die sich auf den *Inhalt* der Daten bezieht, und die Integrität der Datenherkunft, die sich auf die *Quelle* der Daten bezieht. *Vertraulichkeit* meint die Geheimhaltung sensibler Daten oder Ressourcen. *Verfügbarkeit* bezieht sich auf die Möglichkeit der Nutzung benötigter Informationen oder Ressourcen – ein System, das nicht zur Verfügung steht ist mindestens so schlecht wie gar kein System.

**Verschiedene Arten von Intrusionen** Integrität, Vertraulichkeit und Verfügbarkeit eines Computersystems können auf vielfältige Weise beeinträchtigt werden. Zur Unterscheidung verschiedenen Arten von Intrusionen wird in der Literatur häufig differenziert zwischen Intrusion, Misuse und Anomaly. Dabei wird unter einer *Intrusion* (Einbruch) ein Angriff oder Einbruch von außen verstanden. *Misuse* (Missbrauch) meint den Angriff oder Einbruch von innen und *Anomaly* (Anomalie) beschreibt einen ungewöhnlichen Zustand, der auf einen Einbruch (von innen oder von außen) hinweisen kann. Die Begriffe werden im Zusammenhang mit den in dieser Arbeit behandelten Schadprogrammen synonym verwendet. Die obigen Definitionen für eine Intrusion aus [IDS97] und aus [HLSM90] decken sowohl Einbrüche und Angriffe von außen als auch den Missbrauch des Systems durch autorisierte Nutzer, also Angriffe von innen ab.

**Verschiedene Schadprogramme** Zur Differenzierung unterschiedlicher Arten von Malware wird in der Literatur häufig grob unterschieden zwischen "Virus", "Wurm" und "Trojaner".

Unter einem klassischen *Virus* wird im Allgemeinen ein Programm verstanden, das sich ohne die Einwilligung oder das Wissen des Nutzers selbst replizieren und auf einem System verbreiten kann, indem es sich in bereits bestehende Programmroutinen integriert. Die meisten Viren haben die Absicht, möglichst viele Dateien eines Computers zu infizieren und damit gewisse Funktionen zu manipulieren. Andere Systeme können zum Beispiel über infizierte Datenträger oder E–Mails ebenfalls "angesteckt" werden.

Ein *Trojaner* ist ein Programm, dessen Ausführung für den Nutzer den Anschein einer erwünschten Funktion hat, tatsächlich aber den unautorisierten Zugang zum System des Nutzers ermöglicht. Solche Programme sind nicht selbstreplizierend und unterscheiden sich dahingehend von Viren. Trojaner ermöglichen die Ausführung von Remote–Operationen, für die allerdings Interaktionen eines Hackers nötig sind. Malware dieser Art verbirgt sich häufig in frei verfügbarer Software oder anderen Dienstprogrammen.

*Würmer* sind Schadprogramme, die sich über ein Netzwerk selbstständig verbreiten, indem sie Kopien von sich selbst an andere Knoten des Netzwerks verschicken. Im Gegensatz zu Viren sind sie nicht auf die Integration in bestehende Programmroutinen angewiesen. Meistens fügen Würmer dem *Netzwerk* Schaden zu, während Viren Dateien auf dem *Zielcomputer* beschädigen oder zerstören.

Unter dem Begriff "Malware" werden also im Allgemeinen alle unerwünschten und schädlichen Programme oder Codes zusammengefasst. Dabei ist eine klare Grenze zwischen den unterschiedlichen Schädlingen häufig nicht leicht zu ziehen, da viele Schadprogramme zugleich Virus, Wurm und/oder Trojaner sind.

In dieser Arbeit werden die Begriffe "Malware", "Schadprogramm" und "Angriff" synonym als zusammenfassende Ausdrücke verschiedener Arten bösartiger Programme verwendet und es wird keine detaillierte Differenzierung vorgenommen.

## 2.2. Kategorisierung von Intrusion Detection Systemen

*Intrusion Detection Systeme* (IDS) sind Programme zur Automatisierung des Intrusion Detection Prozesses. Die bereits zitierten IDSG definiert den Begriff "Intrusion Detection" als

*"De[n] Prozess der Erkennung eines versuchten Einbruchs, eines gerade stattfindenden Einbruchs oder eines in der Vergangenheit stattgefundenen Einbruchs."* [IDS97]

Im Allgemeinen teilt man IDS in zwei Gruppen ein und unterscheidet netzwerkbasierte IDS von hostbasierten IDS. Obwohl die in dieser Diplomarbeit vorgenommenen Analysen lediglich auf der Basis von Daten durchgeführt werden, die an einem Host aufgezeichnet wurden, werden in den beiden folgenden Abschnitten zunächst beide Gruppen erläutert. In Abschnitt 2.2.3 wird anschließend kurz auf die zwei grundlegenden Ansätze zur Erkennung von Schadprogrammen eingegangen.

### 2.2.1. Netzwerkbasierte Intrusion Detection Systeme

Netzwerkbasierte IDS (kurz NIDS) beobachten den Netzverkehr eines Rechners oder eines Netzwerksegments, um maliziöses Verhalten zu entdecken. Sie überwachen den Datenverkehr von Routern oder Switchen auf Paketebene und können diesen aufzeichnen und darstellen. Meistens wird eine Vielzahl von Hosts überwacht. Dabei wird typischerweise separate Hardware eingesetzt, so dass die Ressourcen der Hosts nicht genutzt werden müssen und deren Leistung somit nicht beeinträchtigt wird. Der Ausfall eines Hosts hat folglich keine Auswirkungen auf die Funktion des IDS.

Netzwerkbasierte IDS sind unter anderem dazu in der Lage, Bufferoverflow–Angriffe[2], ungültige Pakete, Änderungen von Applikationsprotokollen zur Nutzung unerwünschter Funktionen, Zugriffsversuche auf vertrauliche Daten wie Passwörter oder Kennwortdatenbanken, Angriffe auf die Firewall und Portscans[3] zu erkennen. Netzbasierte Angriffe wie dDoS–Attacken (distributed Denial of Service), die von einer größeren Anzahl anderer Systeme aus erfolgen, können identifiziert werden.

Bei der Erkennung eines Angriffs alarmiert das IDS die verantwortliche Person und ergreift eventuell bereits selbst erste Gegenmaßnahmen.

Die Konfiguration und die Pflege netzwerkbasierter IDS ist meistens sehr aufwendig. Liegen zudem Konfigurationsfehler anderer Schutzkomponenten oder Fehler in der Datenübertragung vor, werden häufig falsche Alarme erzeugt. Maliziöse Ereignisse in Daten, die *verschlüsselt* über das Netzwerk versendet werden, können nicht erkannt werden. Ferner kann das Verhalten des Angriffsziels nur begrenzt nachvollzogen werden, da nur die Auswirkungen erfasst werden, die im Netzverkehr sichtbar sind. Somit sind zum Beispiel Viren, mit denen das System über einen Datenträger infiziert wurde, für das NIDS nicht auffindbar. Ein weiterer Nachteil von NIDS ist die teilweise lückenhafte Überwachung des Netzverkehrs, die durch eine zu hohe Netzlast eines zu kontrollierenden Netzwerksegments entstehen kann.

[2] Bei einem Bufferoverflow–Angriff speichert ein Prozess Daten außerhalb des Pufferspeichers, der dafür eigentlich reserviert ist. Diese zusätzlich gespeicherten Daten überschreiben angrenzenden Speicher, der zum Beispiel Programmvariablen und Kontrollinformationen enthält. Daraus kann ein unberechenbares Programmverhalten resultieren, das zu einer Vielzahl unterschiedlicher Fehler und der Ermöglichung von Eingriffen in das System führen kann.

[3] Bei einem Portscan werden offene Ports eines Systems oder Netzwerks und die zur Verfügung stehenden Dienste ermittelt, um Angriffe an diese Ports zu richten.

### 2.2.2. Hostbasierte Intrusion Detection Systeme

Hostbasierte IDS (kurz HIDS) werden auf dem zu schützenden System betrieben und analysieren Daten auf Anwendungs– und Betriebssystemebene. Dazu gehören unter anderem Interaktionen von Anwendungen mit dem Betriebssystem, Prüfsummen, Dateigrößen, bestehende Logdaten und vom IDS zusätzlich erzeugte Logdaten.

Ein hostbasiertes IDS kann an das zugrunde liegende System angepasst werden. Es ist im Gegensatz zu netzwerkbasierten IDS insbesondere dazu in der Lage, den Missbrauch des Systems durch autorisierte Nutzer, also "Angriffe von innen", zu erkennen. Reaktionen des Systems können beobachtet und ausgewertet werden. Datei– und Applikationszugriffe, Anmeldungen am System und Zugriffsverletzungen können analysiert und einem Nutzer zugeordnet werden. Das bedeutet auch, dass das HIDS verschiedenste Malware–Arten wie Viren, Würmer und Trojaner erkennen kann, die verdächtige Aktionen auf dem Host durchführen. Zur Erhaltung der Integrität kann geprüft werden, ob sich Änderungen an Daten oder Programmen ergeben haben, die einem definierten Sollzustand widersprechen.

Bei der Erkennung eines Angriffs erzeugt auch dieses System einen Alarm oder ergreift sogar eigenständig Gegenmaßnahmen.

Hostbasierte IDS müssen im Gegensatz zu netzwerkbasierten IDS auf jedem zu überwachenden System installiert und an vorhandene Applikationen angepasst werden. Sie nutzen Ressourcen der Hosts, wodurch deren Leistung beeinträchtigt werden kann. Von mehreren Systemen ausgehende Angriffe wie dDoS–Attacken können von einem HIDS nicht als solche erkannt werden.

### 2.2.3. Signaturbasierte Angriffserkennung und Anomalieerkennung

Neben der Kategorisierung von Intrusion Detection Systemen nach hostbasierten und netzwerkbasierten Systemen erfolgt eine Differenzierung zwischen signaturbasierter und anomalie– beziehungsweise missbrauchserkennender Intrusion Detection.

Bei der *signaturbasierten Angriffserkennung* werden Muster (Signaturen) erstellt, die einer bekannten Bedrohung entsprechen. Das IDS verfügt in diesem Fall über eine Signaturdatenbank und erkennt bösartiges Verhalten in aktuellen Aktivitäten durch den Vergleich der Aktivitäten mit bekannten Signaturen (Angriffsmustern). Gibt es eine Übereinstimmung, wird die aktuelle Aktivität als Angriff gewertet.

Bei der *Anomalieerkennung* (auch als *Missbrauchserkennung* bezeichnet) wird davon ausgegangen, dass sich das normale Verhalten eines Systems von Angriffen unterscheidet. Daher wird zur Angriffserkennung zunächst "normales" Verhalten von Nutzern, Hosts, Netzwerkverbindungen, Anwendungen und anderen Prozessen analysiert und es werden Profile erstellt. Die Analyse erfolgt in der Regel durch den Einsatz statistischer Verfahren. Nachdem auf diese Art und Weise "normales" Verhalten definiert wurde, werden aktuelle Arbeitsmuster erneut durch die Nutzung statistischer Methoden mit den (statischen) normalen Arbeitsmustern verglichen und als "normal" oder "anomal" bewertet.

**Qualitätsvergleich der Erkennungsmethoden** Die Qualität von Intrusion Detection Systemen wird in erster Linie anhand der auftretenden Falsch– und Fehlmeldungen gemessen. Solche falschen oder fehlenden Alarme sind grundsätzlich ein großes Problem

von IDS, unabhängig von der verwendeten Analysemethode.

Als *falsch–positiv* werden Alarme bezeichnet, die keine tatsächliche Bedrohung darstellen. Kommen solche falschen Alarme selten vor, stellen sie keine gravierende Beeinträchtigung des IDS dar. Treten sie jedoch über einen längeren Zeitraum hinweg häufig auf, führt das dazu, dass (echte) Alarme nicht mehr ernst genommen werden. *Fehlende* Alarme, also nicht erkannte Angriffe, werden als *falsch–negativ* bezeichnet und bergen ein noch höheres Risiko als falsch–positiv Meldungen, da in diesem Fall eine Bedrohung gar nicht bemerkt wird.

Der Vorteil *signaturbasierter Angriffserkennung* ist die Generierung weniger falsch–positiv–Meldungen und die Effektivität dieser Methode. Allerdings bedürfen die Signatur–Datenbanken einer ständigen Aktualisierung, um falsch–negativ–Meldungen zu vermeiden und auch die Erkennung der neusten Angriffe zu ermöglichen. Das hat zur Folge, dass die Datenbanken stetig wachsen und neue, noch nicht gesehene Angriffe dennoch nicht erkannt werden, wenn die entsprechende Signatur noch nicht erstellt wurde. Viele signaturbasierte Systeme nutzen zur Behandlung dieses Problems eine Heuristik anstelle eines 1:1–Abgleichs. Dadurch wird die Analyse beschleunigt und es können auch Angriffe erkannt werden, für die in der Datenbank keine *exakte* Signatur existiert, die einer vorhandenen Signatur jedoch ähneln.

Der Vorteil der *Anomalieerkennung* ist die Möglichkeit, auch bislang unbekannte Angriffe entdecken zu können und keine Signaturdatenbank pflegen zu müssen. Ein Nachteil des Verfahrens ist die Tatsache, dass zunächst normale Muster gewonnen werden müssen. Dazu werden repräsentative Daten erhoben, die das Spektrum normalen Verhaltens möglichst vollständig abdecken. Da die Systemumgebungen häufig sehr heterogen und dynamisch sind, kommt es aufgrund zu statischer Profile normalen Verhaltens im Vergleich zu signaturbasierten Systemen wesentlich häufiger zu falsch–positiv–Meldungen.

# 3. Intrusion Detection und Data Mining

Sowohl im Bereich der Intrusion Detection als auch in vielen anderen Bereichen der Datenerfassung, –speicherung und –verteilung hat sich im Laufe der vergangenen Jahre aufgrund der großen Menge anfallender Daten die Notwendigkeit computergestützter Programme und Techniken zur Analyse der Daten ergeben. Die verwendeten Analyseverfahren werden unter den Begriffen *Knowledge Discovery* (KD) und *Data Mining* zusammengefasst und stellen einen immer weiter wachsenden Forschungs– und Anwendungsbereich dar. Sie bauen auf Verfahren und Theorien unterschiedlicher Fachbereiche auf. Dazu zählen unter anderem die Bereiche Statistik, Mathematik, Datenbanken, Mustererkennung, maschinelles Lernen und künstliche Intelligenz, Datenvisualisierung, Data Warehousing und Online Analytical Processing (OLAP), Optimierung und Hochleistungsrechnen.

Die Entwickler von Anti–Malware–Programmen nutzen mittlerweile zwar verbreitet automatische Tools zur Code–Analyse und zur Erstellung von Signaturen bekannter Schadprogramme, zur Erkennung eines veränderten Angriffs reicht dies jedoch nicht aus. Data Mining Techniken bieten hier die Möglichkeit, aufgrund vergangener Beobachtungen identifizierende, charakteristische Eigenschaften und Verhaltensweisen von Angriffen automatisch zu extrahieren und zu "lernen". So kann auch Malware erkannt werden, bei der der Code des Schadprogramms durch Code–Morphing verändert wurde. Solche Angriffe werden bei einem Signaturenvergleich nicht erkannt.

Durch Data Mining Techniken "gelernte" Malware–Charakteristika können darüber hinaus zum Beispiel zur Aktualisierung von Signaturdatenbanken oder zur Weiterentwicklung anderer Schutzmechanismen verwendet werden.

Der folgende Abschnitt 3.1 gibt zunächst einen kurzen Überblick über das Gebiet der Wissensentdeckung und des Data Minings. Als Datengrundlage zur Analyse von Angriffen dienen in der Intrusion Detection verbreitet die von Malware ausgeführten und aufgezeichneten Systemcalls, auf die in Abschnitt 3.2 näher eingegangen wird. In den Abschnitten 3.3 bis 3.6 werden verschiedene Ansätze zur Angriffserkennung durch die Nutzung von Data Mining Techniken und auf der Basis von Systemcall–Logdateien vorgestellt. Obwohl das Ziel dieser Diplomarbeit nicht die *Erkennung* von Angriffen sondern die Klassifikation von Schadprogrammen ist, gleichen sich die beiden Problemstellungen und die Erkenntnisse aus dem Bereich der Angriffserkennung werden in dieser Diplomarbeit genutzt. Arbeiten zur Klassifikation von Schadprogrammen werden in Abschnitt 3.7 vorgestellt.

## 3.1. KDD und Data Mining

*Knowledge Discovery in Databases (KDD)* beschreibt ursprünglich den Prozess, aus großen Rohdatenmengen in Datenbanken verständlichere und hilfreichere Daten zu extrahieren. Mittlerweile wird der Begriff KDD häufig allgemein definiert als der Prozess des

Abbildens systemnaher Daten in andere Formen, die kompakter, abstrakter und nützlicher sind (vgl. [FPsS96]). Der Prozess beinhaltet die Datenbereitstellung, –auswahl, –skalierung und –bereinigung, die Modellierung von Rauschen in vorhandenen Daten, das Erstellen von Stichproben, Dimensionsreduktionen, Daten– und Wissensvisualisierungen, die Einbeziehung von Vorwissen und die passende Interpretation und Erklärung der Ergebnisse.

*Data Mining* ist ein elementarer Schritt im KDD–Prozess. Unter diesem Begriff wird die Anwendung verschiedener Methoden und Algorithmen auf Datenbestände zur Erkennung von Mustern in den Daten zusammengefasst. Data Mining Techniken versuchen folglich mit Hilfe von Verfahren aus den einleitend genannten Fachbereichen Strukturen und Beziehungen in (großen) Datensätzen aufzuspüren, um daraus neue, nichttriviale Erkenntnisse in Bezug auf die Daten zu erlangen. Fayyad et. al. [FPsS96] definieren Data Mining als einen

*"... Schritt im KDD Prozess, der aus der Anwendung von Datenanalysen und der Entdeckung von Algorithmen zur Erzeugung einer bestimmten Aufzählung von Mustern (oder Modellen) über den Daten besteht."*

Ziele des Data Minings sind unter anderem eine adäquate Klassifikation von Daten, Wahrscheinlichkeitsmodellierungen, Vorhersagen und Schätzungen, Abhängigkeitsanalysen und Optimierungen. Diese Ziele sollen unter anderem mit Hilfe von maschinellen Lernverfahren erreicht werden.

*Maschinelle Lernverfahren* sind Methoden (Algorithmen) der künstlichen Intelligenz. "Lernen" beschreibt in diesem Zusammenhang die Generierung von Wissen aus Erfahrung. Es wird versucht, Gesetzmäßigkeiten in vorliegenden Trainingsdaten zu erkennen, aufgrund derer eine zuverlässige Zuordnung zukünftiger Eingaben zu Ausgaben erfolgen kann. Es werden folglich Modelle (Hypothesen, Funktionen) der Form

$$x \mapsto \mathcal{Y}$$

gesucht, die Objekten $x$ einer Eingabemenge $\mathcal{X}$ einen Ausgabewert einer Ausgabemenge $\mathcal{Y}$ zuordnen. Eingabe– und Ausgabeobjekte werden durch qualitative oder quantitative Merkmale beschrieben. Je nach Datengrundlage und Ziel des Verfahrens werden unüberwachte oder überwachte Lernverfahren verwendet. Zur Optimierung der Zuordnungsfunktion wird häufig ein Fehler minimiert.

### Unüberwachtes Lernen

Beim unüberwachten Lernen ist die Ausgabemenge $\mathcal{Y}$ nicht bekannt. Unüberwachte Lernverfahren versuchen die vorliegenden Daten zusammenzufassen und zu erklären. Es wird nach vorhandenen Mustern, Gruppen von Objekten, Assoziationen zwischen Objekten und Merkmalen oder statistischen Eigenschaften der Daten gesucht.

Eine beliebte Methode des unüberwachten Lernens ist das *Clustering*. Dabei werden Objekte aufgrund von Ähnlichkeits– oder Distanzmaßen wie räumlicher Nähe oder ähnlicher Strukturen in Gruppen, so genannten Clustern, zusammengefasst. Ziel ist die Minimierung der Distanzen innerhalb eines Clusters (Homogenität) und die gleichzeitige Maximierung der Distanzen zwischen verschiedenen Clustern (Heterogenität).

Eine weiteres populäres Verfahren ist das Suchen nach *Assoziationen* zwischen Merkmalen. Assoziationen werden häufig in Form von Regeln der Art "`WENN A DANN B`" dargestellt.

### Überwachtes Lernen

Beim überwachten Lernen ist neben der Eingabemenge $\mathcal{X}$ zu jedem Beispiel auch eine Ausgabe $y \in \mathcal{Y}$ gegeben. Es soll eine Funktion (Hypothese) gefunden werden, die die Zuordnung der gegebenen Beispiele $x \in \mathcal{X}$ zu ihrem vorgegebenen Zielwert (der *Klasse*, dem *Label*) $y \in \mathcal{Y}$ bestmöglich approximiert. Neue Beispiele der Domäne sollen mit Hilfe der erstellten Hypothese möglichst zuverlässig der richtigen Klasse zugeordnet werden können.

Bei den überwachten Lernverfahren wird unterschieden zwischen der *Klassifikation* und der *Regression*. Die Klassifikation sucht eine Funktionsapproximation mit *nominalen* Zielwerten. Die Regression sucht eine Funktionsapproximation mit *numerischen* Zielwerten.

Populäre *Klassifikationsverfahren* sind zum Beispiel die instanzbasierte Klassifikation, die Klassifikation durch Regeln, Entscheidungsbäume oder Graphen und die Stützvektormethode. Bei der *instanzbasierten Klassifikation* beruht die Klassenzuordnung auf der Ähnlichkeit eines Objektes zu den bereits bekannten Objekten. Bei der Klassifikation durch *Regeln* werden Abhängigkeiten zwischen Merkmalen der Objekte und der Klassenzugehörigkeiten der Objekte in Form gültiger Regeln gesucht. In einem *Graphen* werden Merkmale als Zufallsvariablen betrachtet und stellen die Knoten des Graphen dar. Die Kanten repräsentieren bedingte Wahrscheinlichkeiten. *Entscheidungsbäume* betrachten die Merkmale eines Objektes als Information und erstellen einen Baum, dessen Knoten, ähnlich den Graphen, die Merkmale repräsentieren. Die Merkmalsausprägungen bilden die Kanten und die Blätter die Klassen. Somit muss an jedem Knoten eine Entscheidung über die weitere Verzweigung im Baum getroffen werden. Die *Stützvektormethode* versucht eine (mehrdimensionale) separierende Hyperebene in den Merkmalsraum einzupassen, die die gegebenen Beispiele gemäß ihrer Klassenzugehörigkeiten bestmöglich voneinander trennt. Dazu müssen die Objekt durch Vektoren repräsentiert werden. Die Stützvektormethode kann sowohl zur Klassifikation als auch zur Regression verwendet werden.

Die Generalisierungsfähigkeit eines Klassifikationsverfahrens kann dank der beim überwachten Lernen gegebenen Ausgabewerte mit Hilfe eines Trainings– und eines Testdatensatzes überprüft werden. Der so genannte Trainings– oder Lerndatensatz besteht aus einem Teil der vorliegenden Daten und wird an den Lernalgorithmus übergeben. Daraus lernt dieser eine Hypothese $f(x) = \hat{y}$. Dabei bezeichnet $\hat{y}$ die durch die Hypothese vorhergesagte Klasse. Der Testdatensatz enthält die übrigen Daten, die mit Hilfe der gelernten Funktion klassifiziert werden, um anschließend $\hat{y}$ mit der tatsächlichen Klasse $y$ vergleichen zu können. Dieser zweite Teil der Daten dient also der Bewertung des aufgestellten Modells.

### Einordnung der Diplomarbeit

In dieser Diplomarbeit werden maschinelle Lernverfahren zur Klassifikation von *Schadprogramm–Familien* eingesetzt. Es handelt sich dabei um einen hostbasierten Ansatz.

Hostbasierte IDS wurden in Abschnitt 2.2.2 beschrieben.

Zur Erkennung verschiedener Arten von Malware werden die von schädlichen Programmen während ihrer Ausführung getätigten Systemaufrufe analysiert. Diese Aufrufe (die Systemcalls) wurden zu Analysezwecken aufgezeichnet und in Logdateien dokumentiert. Die Klassen der vorliegenden Beispiele sind bekannt. Daher werden Verfahren des *überwachten* Lernens genutzt. Einige Details der verwendeten Lernverfahren werden in Kapitel 4 erläutert. Die Beschreibung der zugrunde liegenden Daten folgt in Kapitel 5 und auf die Systemcalls als Datengrundlage wird in dem folgenden Abschnitt näher eingegangen. Die Erkenntnisse der im weiteren Verlauf des vorliegenden Kapitels beschriebenen Arbeiten werden für die Problemstellung der Schadprogramm–Klassifikation genutzt.

## 3.2. Systemcalls als Datengrundlage

Eine im Bereich der Intrusion Detection häufig verwendete Datengrundlage zur Analyse von Schadprogrammen bilden Systemcalls, die von der Malware während ihrer Ausführung aufgerufen werden.

*Systemcalls* (Systemfunktionen) sind vom Betriebssystem bereitgestellte Funktionen, die Zugriff auf den Befehlssatz der CPU und den gesamten Speicher des Systems haben. Zum Schutz des Systems haben Anwendungsprogramme keinen direkten Zugriff auf diese Ressourcen, sondern benutzen Systemcalls zur Abwicklung privilegierter Operationen. Privilegierte Operationen sind Vorgänge wie Speicher–Allozierung, Thread–Erzeugung, das Laden von Bibliotheken, File–Handling und Ähnliches. Zur genauen Spezifizierung auszuführender Operationen besitzen Systemcalls Parameter, denen Werte übergeben werden können. Die Menge der Systemcalls eines Betriebssystems ist begrenzt. Linux 2.0 besitzt zum Beispiel 164 unterschiedliche Systemcalls[4]. Populäre Systemcalls Unix–ähnlicher Systeme sind beispielsweise `open`, `read`, `write` und `close`.

Da auch Malware zur Ausführung von Operationen über Systemcalls mit dem zugrundeliegenden Betriebssystem interagieren muss, gilt die Intrusion Detection auf der Basis von Systemcalls als vielversprechender Ansatz. Zur Analyse von Malware werden daher verbreitet die von Schadprogrammen ausgeführten Systemcalls in der Reihenfolge ihres Auftretens aufgezeichnet und in der Form von Logdateien gespeichert. Die Erstellung der Logdateien (auch *Traces* oder *Berichte* genannt) kann mit Hilfe unterschiedlicher Tools erfolgen, die teilweise frei verfügbar sind. Unter Unix können Systemcall–Traces zum Beispiel mit Hilfe von `strace` oder `ktrace` erstellt werden. Für Windows–Systeme gibt es das kommerzielle Tool *CWSandbox*[5], das die Malware–Programme für eine begrenzte Zeit in einer simulierten Umgebung ausführt und die Systemcall–Aufrufe aufzeichnet.

Systemcall–Logdateien bieten eine Vielzahl an Möglichkeiten zur Analyse mit Data Mining Techniken und werden daher sowohl in den im Folgenden vorgestellten Arbeiten als auch in dieser Diplomarbeit genutzt. Dabei gibt es unterschiedliche Ansätze zur Extraktion von Merkmalen, anhand derer eine Klassifikation erfolgen kann. Es werden zum Beispiel Systemcall–Sequenzen, einzelne Systemcalls oder Systemcall–Mengen verwendet. Parameterwerte der Systemcalls werden bei der Analyse häufig nicht betrachtet.

[4] http://linux.die.net/man/2/intro
[5] http://www.cwsandbox.org

## 3.3. Intrusion Detection auf Systemcall–Sequenzen

In diesem Abschnitt werden zunächst einige Arbeiten vorgestellt, die Systemcall–Sequenzen zur Erkennung normaler und anomaler Verhaltensweisen eines Systems nutzen. Die Arbeit von Forrest et. al. in [FHSL96] gehörte zu den ersten Ansätzen, die Systemcalls als Datengrundlage zur Intrusion Detection nutzten.

### 3.3.1. Anomalieerkennung durch Sequenz–Vergleiche

Stephanie Forrest et.al. präsentierten 1996 in [FHSL96] einen Ansatz zur Anomalieerkennung in privilegierten UNIX–Programmen. Jedes Programm legt implizit eine Menge von Systemcall–Sequenzen fest, die es erzeugen kann. Daher lag die Vermutung einer äußerst konsistenten lokalen Ordnung der Systemcalls innerhalb eines Programms nah. Dies führte zu der Idee, eine Datenbasis aus Sequenzen normalen Verhaltens für jeden zu überwachenden Prozess zu erstellen. Dazu werden lediglich die Systemcalls betrachtet. Parameter der Systemcalls und übergebene Werte, zeitliche Informationen und Anweisungssequenzen zwischen den Systemcall–Aufrufen werden nicht berücksichtigt. Es wird lediglich bewertet, inwieweit bereits mit Hilfe dieses einfachen Ansatzes Programme voneinander unterschieden und Angriffe erkannt werden können.

Zur Erstellung eines Profils normalen Verhaltens wird ein gleitendes Fenster der Größe $k + 1$ über die aufgezeichneten Systemcall–Traces eines Prozesses geschoben. Die in dem Fenster vorkommenden Systemcall–Sequenzen bilden die Datenbank normaler Folgen. Sequenzen neuer Traces werden dann mit demselben Verfahren erstellt und mit den Daten in der Datenbank des normalen Verhaltens verglichen, um abweichende Sequenzen zu ermitteln. Die Anzahl der Anomalien wird zum einen als absolute Zahl und zum anderen als prozentualer Anteil aller *möglichen* Sequenz–Abweichungen eines Traces dokumentiert, um auch die Länge der Traces zu berücksichtigen.

Die für unterschiedliche Programme und Angriffe durchgeführten Experimente mit den Fenstergrößen 5, 6 und 11 zeigten, dass kurze Systemcall–Sequenzen, die von den jeweiligen Programmen während ihrer normalen Ausführung aufgerufen werden, innerhalb eines Programms tatsächlich sehr konsistent sind. Sequenzen verschiedener Programme unterscheiden sich voneinander.

Da der Code der meisten Programme statisch ist und die Systemcalls somit an festen Stellen im Code auftreten wird von Forrest et. al. vermutet, dass diese Konsistenz der Sequenzen innerhalb eines Programms auch für in [FHSL96] nicht getestete Programme gilt. Darüber hinaus kommt diese Arbeit zu dem Schluss, dass jede Intrusion einige unübliche Sequenzen ausführt.

**Probleme des Ansatzes** Ein Problem des Ansatzes besteht in der Erstellung der Datenbasis. Hierfür muss entschieden werden, wie viel normales Verhalten aufgezeichnet wird und ob dazu das Verhalten eines Nutzers beobachtet wird oder ob künstliche Programmausführungen verwendet werden. Wird die Datenbasis zu groß, wird das Verfahren ineffizient. Enthält die Datenbank *alle möglichen* Muster, wäre sie für die Anomalieerkennung nutzlos. Da sich das "normale" Verhalten im Laufe der Zeit mit hoher Wahrscheinlichkeit verändert, ist eine kontinuierliche Aktualisierung der Datenbasis erforderlich. Einige

Angriffsformen wie zum Beispiel Race Condition Attacks[6] und Angriffe unter Nutzung fremder Berechtigungen können mit diesem Verfahren nicht erkannt werden.

### 3.3.2. Angriffserkennung durch Sequenz–Regeln

Wenke Lee und Salvatore J. Stolfo verwenden in [LS98] die Systemcall–Traces aus dem im vorangegangenen Abschnitt 3.3.1 beschriebenen Ansatz [FHSL96]. Auch sie wollen Angriffe von normalen Programmausführungen unterscheiden. Dazu nutzen sie ebenfalls gleitende Fenster, um charakteristische Systemcall–Sequenzen der Länge $n$ zu finden. Die Systemcalls einer $n$–Sequenz werden mit $\{p_1, \ldots, p_n\}$ bezeichnet. Im Gegensatz zu [FHSL96] erfolgt keine Datenbank–Erstellung, sondern es werden Regeln der Form

"`WENN Systemcall` $p_i$`=a UND Systemcall` $p_j$`=b UND ...DANN normal/anomal`"

mit $i, j \in \{1, \ldots, n\}$ gesucht. Dazu wird der Regellerner RIPPER [Coh95] genutzt. Da RIPPER Regeln für die Klasse liefert, für die weniger Beispiele in der Lernmenge enthalten sind, können durch Variation der Lernmenge Regeln für normales Verhalten ebenso wie Regeln für maliziöses Verhalten erstellt werden.

Zur Klassifikation eines Traces wird ein Fenster der Größe $2l + 1$ mit einem Shift von $l$ über dieses Trace–Beispiel geschoben. Werden in einer Region der Länge $2l + 1$ mehr als $l$ anomale Sequenzen gefunden, gilt die Region als "anomal". Übersteigt der Prozentsatz anomaler Regionen eines Systemcall–Traces einen definierten Schwellwert, wird das Trace als Angriff klassifiziert.

Diese Vorgehensweise ist im Vergleich zu dem in [FHSL96] vorgestellten Verfahren durch die Berücksichtigung anomaler Traces während der Lernphase weniger fehleranfällig. Die Experimente ergaben, dass *Regeln für normales Verhalten* zur Erkennung von Anomalien geeignet sind. Der Anteil anomaler Regionen ist bei einem Angriff wesentlich höher als bei einer normalen Programmausführung, so dass Systemcall–Traces anhand dieser Werte gut klassifiziert werden können.

*Regeln für maliziöses Verhalten* eignen sich gut zur Erkennung *bekannter* Angriffe, jedoch weniger gut zur Erkennung von Intrusionen, die zum Zeitpunkt des Trainings noch nicht vorlagen.

**Vorhersage normaler Systemcalls** Bei einem weiteren Ansatz in [LS98] werden lediglich normale Sequenzen betrachtet und Korrelationen der Form

"`WENN Systemcall` $p_i$`=a UND Systemcall` $p_j$`=b UND ...DANN Systemcall` $p_k$`=c`"

mit $i, j, k \in \{1, \ldots, n\}$ zwischen den Systemcalls einer Sequenz gesucht. Bei der Klassifikation eines Beispiels wird für jede gefundene Regel die Konfidenz berechnet durch

$$conf(Regel) = \frac{|\text{Seq, in denen die Regel gilt}|}{|\text{Seq, in denen die Regel gilt}| + |\text{Seq, in denen die Regel nicht gilt}|}$$

[6] Unter *Race Condition Attacks* werden die Fälle verstanden, in denen das Ergebnis eines Prozesses von der Reihenfolge der Ankunft, dem Scheduling oder der Ausführung spezieller konkurrierender Anweisungen abhängt.

mit "Seq" einer Sequenz. Wird eine Regel durch eine Sequenz des Traces verletzt, wird der "Wert" des Traces um $100 \cdot conf(verletzte\ Regel)$ erhöht. Der Durchschnittswert bezüglich der Anzahl an Sequenzen des Traces wird benutzt um zu entscheiden, ob ein Angriff vorliegt oder nicht. Die in [LS98] durchgeführten Experimente ergaben, dass die Diskrepanz normaler und maliziöser Traces bei der Vorhersage des 11–ten Systemcalls und bei der Vorhersage des mittleren Systemcalls einer Sequenz der Länge 11 groß genug ist, um diese Traces voneinander zu unterscheiden. Vorhersagen für andere Systemcall–Positionen in Sequenzen waren schlechter, da in diesen Fällen wahrscheinlich keine stabilen Muster vorliegen.

### 3.3.3. Bestimmung der optimalen Sequenzlänge

Die in den vorangegangenen Abschnitten 3.3.1 und 3.3.2 vorgestellten Ansätze [FHSL96] und [LS98] nutzen eine fest vorgegebene Fenstergröße. In [ESL01] weisen Eskin, Lee und Stolfo auf den Tradeoff zwischen kurzen und langen Sequenzen hin. Wird angenommen, dass alle Sequenzen mit derselben Wahrscheinlichkeit auftreten und es $|\Sigma|$ unterschiedliche Systemcalls gibt kann ebenfalls angenommen werden, dass jede Sequenz der Länge $n$ mit einer Wahrscheinlichkeit von $\frac{1}{|\Sigma|^n}$ auftritt[7]. Lange Sequenzen werden in den Daten demnach wesentlich seltener vorkommen als kurze, sind intuitiv aber genauer als kurze Sequenzen.

**Feste, optimale Fenstergröße** In [ESL01] wird zunächst ein informationstheoretisches Rahmenkonstrukt zur Auswahl einer optimalen Fenstergröße vorgestellt. Um zu schätzen, wie gut sich verschiedene Fenstergrößen eignen, wird mit Hilfe der bedingten Entropie die Regularität der Daten bei unterschiedlichen Fenstergrößen $n$ ermittelt. Die Entropie ist ein Maß für die Unordnung der Daten bezüglich ihrer Klassenzugehörigkeiten. Dieses Maß wird in Abschnitt 4.3 (Definition 4.2) genauer erläutert. Es wird also die Fenstergröße gewählt, bei der die Entropie der Beispiele (die Unordnung) am niedrigsten ist.

**Kontextabhängige Fenstergröße** In einem zweiten Ansatz in [ESL01] wird die Fenstergröße kontextabhängig gewählt, da Prozessausführungen von vielen Faktoren wie zum Beispiel unterschiedlichen Eingaben oder dem Systemzustand abhängig sind. Um Sequenzen gruppieren und vergleichen zu können, die sich nur in einzelnen Systemcalls unterscheiden, werden Platzhalter (Wildcards) in die Sequenzen eingefügt. Zur Vorhersage eines Systemcalls werden vorangegangene Subsequenzen einer Systemcallsequenz ermittelt. Eskin et. al. nutzen *Sparse Markov Transducers* (SMTs) zur Modellierung dieses Ansatzes. SMTs stellen eine Erweiterung wahrscheinlichkeitstheoretischer Suffixbäume dar. Der Pfad von der Wurzel eines solchen Baumes zu einem Blatt repräsentiert eine Systemcallsequenz. Für Details der Modellierung sei an dieser Stelle auf [ESL01] verwiesen.

Die in [ESL01] durchgeführten Experimente auf den DARPA 1999 Daten und den UNM–Daten zeigten, dass sich die optimalen Fenstergrößen für verschiedene Prozesse unterscheiden. In den meisten Fällen konnten mit dem kontextabhängigen Modell bessere Ergebnisse erzielt werden als mit einer festen (optimalen) Fenstergröße.

[7] $n$ entspricht in [FHSL96] der Fenstergröße $k + 1$.

### 3.3.4. Aussagekraft von Systemcall–Sequenzen

Zusammenfassend bestätigen die Ergebnisse der Arbeit von Lee und Stolfo die Ergebnisse aus [FHSL96]. Beide Arbeiten kommen zu dem Schluss, dass das normale Verhalten eines Programms auf der Grundlage von Systemcall–Sequenzen fester Länge ermittelt werden kann. Auf diese Weise modelliertes normales Verhalten kann anschließend durch Sequenz–Vergleiche zur Erkennung von Anomalien genutzt werden. Mit Hilfe von Sequenz–Regeln wurde das Programmverhalten in [LS98] generalisiert. Mit dem entsprechenden Modell wurden im Vergleich zu [FHSL96] bessere Vorhersagen getroffen. Eine weitere Performanzsteigerung dieser Ansätze kann laut [ESL01] erreicht werden, wenn die Fenstergröße zur Erstellung der Sequenzen kontextabhängig gewählt wird.

Durch die Angriffserkennung auf der Grundlage von *Teil*–Sequenzen eines Traces besteht die Möglichkeit, Angriffe bereits *während* ihrer Ausführung zu erkennen. Nämlich genau dann, wenn eine genügend große Anzahl von Sequenzen bereits als maliziös klassifiziert wurde.

Eine grundsätzliche Voraussetzung der vorgestellten Ansätze [FHSL96], [LS98] und [ESL01] ist die annähernde Vollständigkeit der Trainingsdaten bezüglich des "normalen" Verhaltens eines Programms oder Nutzers. Ist diese Voraussetzung nicht erfüllt, werden unter Umständen neue Sequenzen, die lediglich bisher ungesehenes, normales Verhalten darstellen, als Angriff klassifiziert.

## 3.4. Intrusion Detection auf Systemcallvektoren

Systemcall–Logdateien werden zu Analysezwecken verbreitet als Textdokumente betrachtet. Diese Dateninterpretation ermöglicht die Übertragung des kompletten Spektrums weit entwickelter Methoden der Textanalyse auf den Bereich der Intrusion Detection. Zu den anwendbaren Verfahren gehört neben der Merkmalsextraktion eine Vielzahl an Lernverfahren.

Bei der Analyse von Texten werden die vorliegenden Dokumente in eine Vektordarstellung transformiert. Für diesen Schritt wird das *Vector Space Model* genutzt, das im Folgenden beschrieben und auf Systemcall–Logdateien übertragen wird.

#### Das Vector–Space–Modell

Gemäß dem in [SWY75] vorgestellten *Vector Space Model* wird ein Dokument $d_i$ als $m$–dimensionaler *Termvektor* $d_i = (d_{i1}, d_{i2}, \ldots, d_{im})$ betrachtet, in dem $d_{ij}$ das Gewicht des $j$–ten Terms angibt. Sind derartige Indexvektoren für Dokumente gegeben, können darauf verschiedene Berechnungen durchgeführt werden. Zum Beispiel kann ein Ähnlichkeitskoeffizient $s(d_i, d_j)$ berechnet werden, der den Ähnlichkeitsgrad der Dokumente $d_i$ und $d_j$ bezüglich ihrer Terme und Termgewichte widerspiegelt. Durch die Normalisierung der Vektoren auf die Länge 1 kann die relative Distanz zwischen Vektoren bestimmt werden.

Der erste Schritt bei der Analyse von Texten besteht folglich darin, die Wörter oder Terme eines Dokuments zu extrahieren. Diese bilden dann die Menge der Merkmale und liefern die Komponenten für die Vektordarstellung. Die *Merkmalsextraktion* hängt dabei von der Definition der Schriftzeichen ab, die als Teil eines Wortes betrachtet werden

sollen. Neben den Groß– und Kleinbuchstaben A bis Z können Textdokumente Bindestriche, Zahlen, Unterstriche, Sonderzeichen etc. enthalten. In den meisten Fällen wird ein *Wort* als die längste zusammenhängende Sequenz von Buchstaben definiert. Wörter werden demnach extrahiert, indem bei Leerzeichen, Satzzeichen (Komma, Punkt, Semikolon etc.), Sonderzeichen und Zahlen davon ausgegangen wird, dass an diesen Stellen ein neues Wort beginnt. Diese "trennenden Zeichen" werden folglich nicht als Text betrachtet. Bei der Textanalyse besteht die Menge der Merkmale folglich aus einzelnen Wörtern und ein Dokument wird in der Form eines *Wortvektors* der Merkmalsmenge repräsentiert (auch als *Bag of words* oder *Dokumentenvektoren* bezeichnet). Bei dieser Darstellung von Dokumenten wird somit vereinfachend angenommen, dass die Reihenfolge der Wörter vernachlässigt werden kann.

Die Komponentenwerte $d_{ij}$ der Vektoren können auf unterschiedliche Weise berechnet werden.

**Definition 3.1 (Termfrequenz)** *Die* Termfrequenz $TF(w_i, d)$ *bezeichnet die Häufigkeit des Wortes $w_i$ in einem Dokument $d$.*

In diesem Fall setzt sich der Wortvektor eines Dokuments folglich aus den Häufigkeiten der einzelnen Wörter im Dokument zusammen. Eine weitere Möglichkeit ist die Definition der Komponenten als binäres Wortvorkommen.

**Definition 3.2 (Binäres Vorkommen)** *Das* Binäre Vorkommen *(Binary Occurrence)* $BO(w_i, d) \in \{0, 1\}$ *ist 1, wenn das Wort $w_i$ in einem Dokument $d$ vorkommt und sonst 0. Es gilt also*

$$BO(w_i, d) = \begin{cases} 1 & \textit{wenn } w_i \in d \\ 0 & \textit{sonst.} \end{cases}$$

$TF$– und $BO$–Wortvektoren beziehen sich auf einzelne Dokumente. Bei der Analyse einer Dokumentenkollektion ist es häufig sinnvoll, Eigenschaften der Kollektion zu berücksichtigen. Dabei kann die Dokumentenfrequenz von Interesse sein.

**Definition 3.3 (Dokumentenfrequenz)** *Die* Dokumentenfrequenz $DF(w_i, D)$ *gibt an, in wie vielen Dokumenten einer Dokumentenkollektion $D$ das Wort $w_i$ vorkommt.*

Die Berechnung der Komponenten eines Wortvektors durch die Multiplikation der Termfrequenz $TF$ mit einem Faktor, der invers zur Dokumentenfrequenz $DF$ des Terms ist, führt in der Textklassifikation zu guten Ergebnissen.

**Definition 3.4 (Inverse Dokumentenfrequenz)** *Bezeichne $|D|$ die Anzahl der in der Kollektion enthaltenen Dokumente. Dann ist die* inverse Dokumentenfrequenz *definiert als*

$$IDF(w_i, D) = log \frac{|D|}{DF(w_i, D)}.$$

Durch die Multiplikation $TF(w_i, d) \cdot IDF(w_i, D)$ wird den Termen, die in einzelnen Dokumenten sehr häufig vorkommen, aber in der Kollektion als Ganzes eher selten sind, ein hohes Gewichte zugeordnet. Es wird angenommen, dass gerade diese Terme Indikatoren für die jeweils vorliegende Klasse darstellen.

Eine ebenfalls bewährte Repräsentation, die eine Erweiterung der IDF–Darstellung definiert, ist die TFIDF–Darstellung. Sie setzt ein einzelnes Worte in Beziehung zu der Gewichtung aller Wörter in der Kollektion und gibt ein Ranking der Wörter an.

**Definition 3.5 (TFIDF–Repräsentation)**

$$TFIDF(w_i, D) = \frac{TF(w_i, d)IDF(w_i, D)}{\sqrt{\Sigma_j [TF(w_j, d)IDF(w_j, D)]^2}}$$

**Anwendung des Vector Space Modells auf Systemcall–Logdateien** Zur Anwendung der Verfahren zur Textanalyse auf Systemcall–Logdateien wird eine Logdatei als Dokument $d$ betrachtet. Die Systemcalls stellen die Wörter $w_i$ dar, so dass sich ein *Wort*vektor in diesem Fall aus *Systemcalls* zusammensetzt und im Folgenden als *Systemcallvektor* bezeichnet wird.

Sei also zum Beispiel ein Trace $d$ einer Dokumentenkollektion $D$ aus Systemcall–Logdateien gegeben durch

$d$=`loadDll loadDll openKey queryValue openFile`

und $\mathcal{F}$ bezeichne die geordnete Menge aller Systemcalls $\{s_1, s_2, \ldots, s_m\} \in D$ mit $m = |\mathcal{F}|$. In der gesamten Dokumentenkollektion $D$ sei $m = 6$ und die Merkmalsmenge $\mathcal{F}$ sei gegeben durch

$s_1$=`loadDll`,
$s_2$=`openFile`,
$s_3$=`openKey`,
$s_4$=`queryValue`,
$s_5$=`sleep`,
$s_6$=`writeValue`.

Bei der Berechnung der Vektorkomponenten durch die Termfrequenz $TF$ wird das Trace $d$ abgebildet auf den *Systemcallvektor*

$$\vec{x} = \begin{pmatrix} 2 \\ 1 \\ 1 \\ 1 \\ 0 \\ 0 \end{pmatrix} \begin{matrix} \texttt{loadDll} \\ \texttt{openFile} \\ \texttt{openKey} \\ \texttt{queryValue} \\ \texttt{sleep} \\ \texttt{writeValue}. \end{matrix}$$

Das *binäre Vorkommen* eines Systemcalls nach Definition 3.2 ist 1 wenn der Systemcall in dem Trace $d$ vorkommt und ansonsten 0. Die *Dokumentenfrequenz* aus Definition 3.3 gibt an, in wie vielen Traces einer Kollektion von Systemcall–Logdateien der Systemcall vorkommt.

Im Folgenden werden verschiedene Ansätze vorgestellt, in denen Lernverfahren auf Systemcallvektoren angewendet werden. Es ist zu beachten, dass die Reihenfolge der Systemcalls, anders als bei den in Abschnitt 3.3 vorgestellten Ansätzen, hier vernachlässigt wird.

### 3.4.1. Anomalieerkennung auf Systemcallvektoren mit $k$NN

Yihua Liao und V. Rao Vemuri stellten 2002 in [LV02] einen Ansatz zur Klassifikation "normaler" und "anomaler" Verhaltensweisen von Programmen unter Nutzung des *k–Nearest–Neighbor*–Klassifikationsverfahrens vor. Ein Prozess wird dabei wie in dem vorangegangenen Abschnitt beschrieben, als Dokument betrachtet und als Systemcallvektor dargestellt. Statt Sequenzen, wie in den Arbeiten aus Abschnitt 3.3, werden folglich die *Termgewichte der einzelnen Systemcalls* zur Charakterisierung des Programmverhaltens verwendet. Dabei werden zusätzliche Informationen wie übergebene Parameter oder Rückgabewerte ignoriert. Die Komponenten der Vektoren werden als $TF$–Werte (Definition 3.1) oder $TFIDF$–Werte (Definition 3.5) der Systemcalls berechnet.

Bei dem $k$–Nearest–Neighbor Verfahren ($k$NN) wird ein Beispiel aufgrund seiner Distanz zu den Trainingsbeispielen normaler Prozesse als normal oder anomal klassifiziert. Als Distanzmaß dient in [LV02] das wie folgt definierte *Cosinus–Maß*:

$$sim(\vec{x}, \vec{x}_i) = \frac{\Sigma_{s_j \in (\vec{x} \cap \vec{x}_i)} x_j \cdot x_{ij}}{||\vec{x}||_2 \cdot ||\vec{x}_i||_2}$$

Dabei bezeichnet $\vec{x}$ den Testprozess, $\vec{x}_i$ den i–te Trainingsprozess, $s_j$ einen Systemcall, der in $\vec{x}$ und $\vec{x}_i$ vorkommt, $x_j$ den Wert des Systemcalls $s_j$ in $\vec{x}$, $x_{ij}$ den Wert des Systemcalls $s_j$ im Prozess $\vec{x}_i$, $||\vec{x}||_2 = \sqrt{x_1^2 + x_2^2 + x_3^2 + \ldots}$ die Norm von $\vec{x}$ und $||\vec{x}_i||_2$ die Norm von $\vec{x}_i$.

**Einige Anwendungsdetails** Das Verfahren wird auf die BSM–Daten des DARPA 1998 Datensatzes (MIT LL)[8] angewendet. Dieser Datensatz beinhaltet eine Vielzahl von Angriffs–Sitzungen, eingebettet in normales Hintergrundgeschehen. In den vorliegenden Experimenten werden die einzelnen Prozesse der Sitzungen extrahiert und jeweils in die Vektordarstellung transformiert. Die normalen Prozessabläufe werden zur Charakterisierung normalen Verhaltens genutzt. Die Daten enthalten 50 verschiedene Systemcalls, so dass für jeden Prozess ein Vektor dieser Größe erzeugt wird.

Zur Klassifizierung eines neuen Prozesses $\vec{x}$ wird auch dieser zunächst in die Vektordarstellung transformiert. Anschließend wird die Cosinus–Ähnlichkeit von $\vec{x}$ zu allen Beispielen der normalen Datenmenge berechnet. Ist das Ähnlichkeitsmaß (die Distanz) eines Trainingsbeispiels zu dem zu klassifizierenden Prozess 1, wird der Prozess sofort als normal betrachtet. Anderenfalls wird die durchschnittliche Ähnlichkeit der $k$ nächsten Nachbarn berechnet und mit einem definierten Schwellwert verglichen. Liegt der Durchschnittswert über diesem Schwellwert, wird der Prozess als normal klassifiziert, anderenfalls als anomal.

Wird ein Prozess als maliziös klassifiziert, wird die Sitzung, mit der dieser Prozess verknüpft ist, als Angriffs–Sitzung gelabelt. Die Genauigkeit des Verfahrens wird als Anteil der erkannten Angriffe berechnet. Dabei zählt jeder erkannte *Angriff* als ein richtig klassifiziertes Beispiel, auch wenn er mehrere Prozesse enthält. Die Wahrscheinlichkeit der Erzeugung eines Alarms, ohne dass tatsächlich ein Angriff vorliegt (auch als Falsch–Positiv–Meldung ($FP$) bezeichnet) resultiert aus dem Anteil falsch klassifizierter *Prozesse*, anstelle von falsch klassifizierten *Sitzungen*.

[8] http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html

Bei der Verwendung der $TFIDF$–Vektordarstellung und einem Wert von $k$=10 für das $k$NN–Verfahren erreicht die Erkennungsrate bei der passenden Wahl eines Schwellwerts schnell 100% bei einem kleinen Anteil falscher Alarme von 0,44%. Bei der Verwendung der $TF$–Vektordarstellung und einem Wert von $k$=15 gilt Ähnliches, allerdings mit einem Anteil falscher positiver Alarme von 0,87%.

Die $TFIDF$–Vektordarstellung scheint sich hier zur Unterscheidung der Klassen also besser zu eignen. Der Schwellwert und der $FP$–Anteil sind niedriger als bei der Verwendung der $TF$–Vektordarstellung.

**Kombination des $k$NN–Verfahrens mit einer Signaturprüfung** Die Laufzeit der $k$NN–Methode beträgt $O(N)$ bei N Prozessen in der Trainingsmenge, da für jeden zu klassifizierenden Prozess die Cosinus–Ähnlichkeit zu allen $N$ Prozessen der Trainingsmenge berechnet werden muss. Zur effizienteren Angriffserkennung wird das erstellte Modell daher in [LV02] erweitert und mit einer Signaturprüfung kombiniert. Dazu wird eine kleine Beispielmenge maliziöser Prozesse gebildet, die die meisten der im ganzen Datensatz vorhandenen Angriffstypen abdeckt. Jeder neue Testprozess wird nun zunächst mit den maliziösen Beispielen verglichen. Liegt ein perfektes Matching vor, wird der neue Prozess als Angriff gelabelt und die Berechnungen der Ähnlichkeiten zu den übrigen Prozessen der Trainingsmenge entfallen. Anderenfalls wird die oben beschriebene Angriffserkennungs–Prozedur durchgeführt.

Dieses verbesserte Verfahren erkennt alle *bekannten* Angriffe und die meisten neuen Angriffe. Dabei erhöht sich der Anteil falscher Alarme gegenüber dem ersten Ansatz bei der $TFIDF$–Vektordarstellung auf 0,59%, bei der $TF$–Vektordarstellung bleibt der $FP$–Anteil bei 0,87%.

**Probleme des Ansatzes** Ein grundlegendes Problem dieses Ansatzes ist die lange Laufzeit des $k$NN–Verfahrens. Einerseits müssen die Trainingsdaten vollständig sein, damit keine falschen Alarme erzeugt werden weil das vorliegende normale Verhalten in den Trainingsdaten nicht erfasst wurde. Andererseits erhöht sich die Laufzeit des Verfahrens mit jedem in der Trainingsmenge enthaltenen Beispiel. Wird der $k$NN–Algorithmus mit einer Signaturprüfung kombiniert, kann die Laufzeit in einigen Fällen gesenkt werden. Allerdings muss dazu eine möglichst kleine, vollständige Trainingsmenge maliziöser Beispiele erstellt werden. Dieser Schritt kann sehr aufwendig sein und weitere Datenanalysen erfordern.

### 3.4.2. Intrusion Detection auf gewichteten Systemcallvektoren

In [YZF06] wird ein Ansatz zur Anomalieerkennung vorgestellt, der die Stützvektormethode nutzt und eine Merkmalsgewichtung durchführt. Dabei bilden ebenfalls Systemcallvektoren die Datengrundlage, allerdings wird in dieser Arbeit keine Aussage über die Komponentenberechnung der Vektoren getroffen. Die Stützvektormethode (SVM) passt eine separierende Hyperebene in den Merkmalsraum ein, die die gegebenen Beispiele gemäß ihrer Klassenzugehörigkeiten bestmöglich voneinander trennt. Die Beispiele, die der Hyperebene am nächsten liegen bestimmen deren Lage und werden als *Stützvektoren* bezeichnet. Bei nicht linear trennbaren Daten kann die Transformation der Daten in einen

anderen Merkmalsraum eine lineare Trennung dennoch ermöglichen. Eine Funktion, die diese Transformation, eine Skalarproduktberechnung und die Rücktransformation in den Ursprungsraum ersetzt, wird *Kernfunktion* genannt. Eine detaillierte Beschreibung der Stützvektormethode und die genaue Definition von Kernfunktionen folgen in Abschnitt 4.4.

Yao, Zhao und Fan führen in [YZF06] einen Vorverarbeitungsschritt zur Gewichtung von Systemcalls durch. Der berechnete Gewichtsvektor $\vec{w}$ wird in die Kernfunktion integriert.

**Gewichtung durch Rough Sets** Zur Erstellung des Gewichtsvektors wird die *Rough Sets Theorie* verwendet. Mit Rough Sets werden Beispiele in Äquivalenzklassen eingeteilt. Eine Äquivalenzklasse ist eine Zusammenfassung von Beispielen, die bestimmte Gemeinsamkeiten bezüglich ihrer Merkmalswerte aufweisen.

Formal sind Paare $\mathcal{X} = (U, \mathcal{F} \cup \mathcal{L})$ gegeben, mit $U$ einer Menge von Objekten (dem Universum), $\mathcal{F}$ einer nicht–leeren, endlichen Menge von Attributen, die in diesem Fall durch die Systemcalls gegeben ist, und $\mathcal{L}$ dem Klassenattribut, so dass $f : U \rightarrow V_f$ für jedes Attribut $f \in \{\mathcal{F} \cup \mathcal{L}\}$ gilt. Die Menge $V$ ist die Menge aller Attributwerte und $V_f$ ist die Wertemenge von $f$.

Für eine beliebige Teilmenge $A \subseteq \mathcal{F}$ ist eine Äquivalenzrelation definiert als

$$IND_{\mathcal{X}}(A) = \{(x_i, x_j) \in U^2 | \, \forall f \in A : \; f(x_i) = f(x_j)\}.$$

Eine Äquivalenzrelation induziert folglich eine Partitionierung des Universums. Die Äquivalenzklassen werden mit $[x]_A$ notiert.

Bei vielen Problemen treten unscharfe Entscheidungen auf, bei denen die Werte der Attribute und die resultierende Klasse nicht eindeutig voneinander getrennt werden können. Beispielsweise könnten $x_1$ und $x_2$ aus $U$ die in Tabelle 3.1 angegebenen Merkmalswerte aufweisen. Obwohl die Attributwerte $f_1$ und $f_2$ übereinstimmen, divergieren die Label. Bei derartigen Problemen werden *Rough Sets* gebildet, die eine untere und eine obere

| | $f_1$ `loadDll` | $f_2$ `createFile` | $f_L$ Label |
|---|---|---|---|
| $x_1$ | 5 | 2 | Allaple |
| $x_2$ | 5 | 2 | Virut |

Tabelle 3.1.: Beispiel einer unscharfen Entscheidung

*Approximation* der Entscheidung liefern.

Sei also $A \subseteq \mathcal{F}$ und $X \subseteq U$. $X$ wird angenähert, indem die untere und die obere Approximation der Menge $X$ gebildet wird. Bezogen auf das Beispiel aus Tabelle 3.1 sei $X = \{x | f_L(x) = \text{Allaple}\}$. Es können Beispiele aus der Menge $U \setminus \{x_1, x_2\}$ angegeben werden, die *mit Sicherheit* zur Klasse "Allaple" gehören und die untere Grenze definieren:

$$LOWER(A, X) = \{x \mid [x]_A \subseteq X\}$$

Darüber hinaus können Beispiele angegeben werden, die *möglicherweise* zu der Klasse "Allaple" gehören und die obere Grenze bilden:

$$UPPER(A, X) = \{x \mid [x]_A \cap X \neq \emptyset\}$$

Die "Grenzregion" zwischen den Klassen ist unscharf und nicht leer.

Da bei einer Vielzahl gegebener Attribute häufig einige redundant sind, wird mit Hilfe der Rough Set Theorie ein minimales Redukt gesucht. Ein *minimales Redukt* bezeichnet die kleinste Teilmenge der Attribute $\mathcal{F}^* \subseteq \mathcal{F}$, so dass $IND_{\mathcal{X}}(\mathcal{F}^*) = IND_{\mathcal{X}}(\mathcal{F})$ gilt.

Das Problem der Berechnung eines minimalen Redukts ist NP-schwer, allerdings gibt es gute Heuristiken (siehe u.a. [PTL00], [Paw98] und weitere Arbeiten von Zdzislaw Pawlak). Weitere grundlegende Informationen zu Rough Sets können zum Beispiel in [Paw91] oder [Slo92] gefunden werden.

**SVM mit Kerngewichten** In [YZF06] werden die Gewichte aller Merkmale zunächst mit 0 initialisiert. Ist ein Merkmal in einem Redukt enthalten, wird sein Gewicht um $\frac{1}{m}$ erhöht, wobei $m$ die Anzahl aller in dem Redukt enthaltenen Merkmale bezeichnet. Somit bleiben die Gewichte der Merkmale, die nicht in dem Redukte vorhanden sind 0. Diese Merkmale werden vom Lernalgorithmus nicht berücksichtigt.

Zur Evaluation der vorgestellten Methode gewichteter Kernfunktionen wird das Verfahren in [YZF06] auf einem Teil der UNM Daten getestet, die bereits in [ESL01] (Abschnitt 3.3.3) genutzt wurden. Die Anzahl der Merkmale dieses Datensatzes wird von 467 auf 9 reduziert. Zusätzlich wird der KDD–Datensatzes verwendet, bei dem die Anzahl der Merkmale von 41 auf 16 reduziert wird. Die SVM wird mit einem RBF–Kern (siehe auch Gleichung (4.15)) und dem Kernparameter $\gamma = 10^{-6}$ angewendet. Die Genauigkeit des Klassifikators steigt im Vergleich zur Anwendung des Verfahrens ohne eine Gewichtung der Merkmale. Der Anteil nicht erkannter Angriffe sinkt und die Laufzeit verringert sich.

### 3.4.3. Aussagekraft von Systemcallvektoren

Gegenüber der in Abschnitt 3.3 beschriebenen Angriffserkennung haben die hier vorgestellten Ansätze [YZF06] und [LV02] den Vorteil, dass keine separaten Profile für verschiedene Programme erstellt werden müssen und der Speicherplatzbedarf somit wesentlich geringer ist. Die Berechnungen zur Klassifikation neuen Programmverhaltens werden stark reduziert.

In [LV02] werden mit dem $k$NN–Verfahren auf Systemcallvektoren gute Ergebnisse erzielt. Allerdings sind die Berechnungen zur Klassifikation eines Prozesses mit $O(N)$ bei $N$ Prozessen in der Trainingsmenge immer noch hoch. Durch die Kombination des Verfahrens mit einer Signaturprüfung kann die Laufzeit zwar gesenkt werden, allerdings ist die Erstellung der Signaturen schwierig.

In [YZF06] wird die Stützvektormethode zur Klassifikation von Beispielen in der Vektordarstellung genutzt. Auch dieses Verfahren liefert gute Ergebnisse, die laut Yao et. al. noch verbessert werden können, wenn die Systemcalls mit Hilfe der Rough Set Theorie gewichtet werden.

Die den vorgestellten Ansätzen zugrunde liegende Annahme besteht darin, dass die Komponentenwerte der Systemcallvektoren bei normaler Programmausführung konsistent sind und dass unübliche Systemcalls oder unübliche Termgewichte von Systemcalls auftreten, wenn ein Angriff stattfindet. Trifft dieses vorausgesetzte Verhalten auf einen Angriff *nicht* zu, wird dieser nicht erkannt.

Dennoch wurde gezeigt, dass sich die Darstellung von Systemcall–Logdateien als *Systemcallvektoren* neben den Systemcall–Sequenzen aus Abschnitt 3.3 ebenfalls gut zur Erkennung von Angriffen eignet, obwohl dabei die Reihenfolge der Systemcalls vernachlässigt wird.

## 3.5. Intrusion Detection auf Systemcall–Mengen

Ein Nachteil der Betrachtung zusammenhängender *Systemcall–Sequenzen* fester Länge als Merkmale zur Erkennung von Angriffen besteht in der Größe der Merkmalsmenge. Diese wächst exponentiell mit der Länge der Sequenzen. Demgegenüber stellt bei der Betrachtung von Logdateien als *Systemcallvektoren* ein Systemcall ein Merkmal dar und die Merkmalsmenge ist durch die begrenzte Anzahl der Systemcalls eines Betriebssystems beschränkt. Allerdings wird der Aufrufzusammenhang der Systemcalls nicht berücksichtigt. Einen Kompromiss beider Betrachtungsweisen stellen *Systemcall–Mengen* dar. Eine Systemcall–Menge ist eine geordnete Systemcall–Sequenz. Die Anzahl der Merkmale wird im Vergleich zu der Nutzung von Sequenzen reduziert und der Aufrufzusammenhang wird im Gegensatz zu der Analyse von Systemcallvektoren dennoch berücksichtigt. Die in dem folgenden Abschnitt 3.5.1 vorgestellte Arbeit von Kang, Fuller und Honavar nutzt daher Systemcall–Mengen zur Erkennung von Angriffen.

### 3.5.1. Bewertung verschiedener maschineller Lernverfahren

Zur Darstellung von Logdateien durch Systemcall–Mengen werden in [KFH05] Teilsequenzen $X'$ der in einem Trace $Z_i$ dokumentierten Systemcall–Folgen extrahiert. Die $k$–te *Teilsequenz* $X'$ wird definiert als

$$X'_{ik} = x_1 x_2 x_3 \dots x_l$$

mit $l$ einer Konstante, die die Länge beschreibt. Die Komponenten $x_n$ der Teilsequenz entsprechen jeweils einem Systemcall aus der geordneten Menge $\mathcal{F}$ aller in den Logdateien vorhandenen Systemcalls $s_1, \dots, s_m$. Eine Teilsequenz wird nun in eine Mengendarstellung abgebildet, indem sie als Textdokument betrachtet und durch das in Abschnitt 3.4 beschriebene Verfahren der Textanalyse als Vektor dargestellt wird. Die Teilsequenz wird folglich durch eine geordnete Liste $X_{ik} = (d_{ik1}, d_{ik2}, \dots, d_{ikm})$ repräsentiert. Darin wird mit $d_{ikj}$ die Anzahl der Vorkommen des Systemcalls $s_j$ in der Teilsequenz $X'_{ik}$ notiert. Es gilt also $d_{ikj} = TF(s_j, X'_{ik})$. Die Mengen $X_{ik}$ werden mit der Klasse der Sequenz $Z_i$ gelabelt, aus der sie extrahiert wurden.

**Verschiedene Lernverfahren** Wie in einigen bereits vorgestellten Ansätzen werden in [KFH05] Experimente auf den UNM–Daten und den DARPA 1998–Daten (MIT LL)[9] durchgeführt. Im UNM–Datensatz enthält jedes Trace die Systemcalls eines Programms und stellt in den meisten Fällen nur einen Prozess dar. Ist mehr als ein Prozess in einem Trace enthalten, wird eine Systemcall–Folge für jeden einzelnen Prozess extrahiert.

[9] Die in [KFH05] verwendete, bereinigte Version dieses Datensatzes steht unter `http://www.cs.iastate.edu/~dkkang/IDS_Bag/` zur Verfügung.

Neben dem bereits in [LS98] (Abschnitt 3.3.2) verwendeten Regellerner RIPPER und der in 3.4.2 benutzten SVM werden eine Logistische Regression, ein Baumlerner und ein Naive Bayes Multinomial–Klassifizierer getestet. Die Logistische Regression versucht Relationen zwischen der binären Klassenzugehörigkeit von Beispielen und der Menge ihrer Merkmale zu finden. Bei der Klassifikation durch Regeln wird nach Abhängigkeiten zwischen den Merkmalen der Objekte und der Klassenzugehörigkeiten der Objekte in der Form gültiger Regeln gesucht. Die SVM passt eine separierende Hyperebene in den Merkmalsraum ein, die die gegebenen Beispiele gemäß ihrer Klassenzugehörigkeiten bestmöglich voneinander trennt. Das Verfahren wird hier mit einem linearen Kern (siehe Gleichung (4.12)) und Sequential Minimal Optimization (SMO, [Pla98]) angewendet. Bei dem C4.5 Baumlerner werden die Merkmale eines Klassifizierungsproblems als Information betrachtet und bilden die Knoten des Baumes. Die Merkmalsausprägungen bilden die Kanten und die Blätter die Klassen. Es wird somit in jedem Knoten des Baumes ein Merkmal abgefragt und für die möglichen Werte werden Entscheidungsalternativen angegeben. Der Naive Bayes Multinomial–Klassifizierer ordnet ein Objekt derjenigen Klasse zu, zu der es mit der größten Wahrscheinlichkeit gehört. Eine detaillierte Beschreibung dieser Lernverfahren folgt in Kapitel 4.

Mit Ausnahme des Naive Bayes Multinomial–Lerners liegt die Genauigkeit aller Verfahren bei fast 100% und es werden sehr wenige falsche Alarme erzeugt. Kleinere Schwankungen der Modell–Performanzen sind abhängig von den Daten. Der Bayes–Klassifikator liefert bei einigen Daten eine sehr niedrige Erkennungsrate.

### 3.5.2. Aussagekraft von Systemcall–Mengen

Wie einleitend bereits erläutert ermöglichen Systemcall–Mengenvektoren gegenüber der Betrachtung einzelner Systemcalls bei der Darstellung der Logdateien als Systemcallvektoren die Einbeziehung des Aufrufzusammenhangs. Die Merkmalsmenge ist im Vergleich zu der Nutzung von Systemcall–Sequenzen wesentlich kleiner. Da eine Menge aus einem *Teil* des Traces besteht, ist es wie bei der Betrachtung von Sequenzen möglich, Malware mit Hilfe eines einfachen Zählprogramms bereits *vor* der Terminierung eines Prozesses zu erkennen. Diese Darstellung bietet folglich einen guten Kompromiss zwischen den Darstellungen der vorangegangenen Abschnitte. Die Vorteile der Sequenz–Darstellung werden genutzt, während die Nachteile der Betrachtung einzelner Systemcalls behoben werden.

Die in [KFH05] durchgeführten Experimente bestätigen diese Einschätzung. Maschinelle Lernverfahren wie der Regellerner RIPPER, die SVM, der Baumlerner und die Logistische Regression liefern erstaunlich gute Ergebnisse bei der Angriffserkennung auf dieser Datengrundlage. Lediglich die Performanz des Naive Bayes–Klassifikators ist nicht ganz so gut. Systemcall–Mengen sind demnach ebenfalls eine adäquate Repräsentation von Systemcall–Logdateien zur Erkennung von Angriffen.

Dem vorgestellten Ansatz liegt die Annahme zugrunde, dass die Systemcall–Mengen bei normalen Programmausführungen konsistent sind und unübliche Mengen auftreten, wenn ein Angriff stattfindet. Trifft dieses Verhalten auf einen Angriff *nicht* zu, wird dieser nicht erkannt.

## 3.6. Intrusion Detection auf Systemcalls und Parameterwerten

In den in diesem Abschnitt vorgestellten Arbeiten werden Parameterwerte bei der Anomalieerkennung auf der Basis von Systemcall–Logdateien berücksichtigt. Die bisher vorgestellten Ansätze, die Systemcall–Sequenzen, Systemcallvektoren oder Mengenvektoren betrachtet haben, sind nicht dazu in der Lage Angriffe zu erkennen, die legitime Systemcall–Folgen nutzen. Bei einer *Mimicry Attacke* wird jedoch zum Beispiel maliziöser Code eingespeist, indem legitime Systemcall–Sequenzen imitiert aber bösartige Aktivitäten durchführt werden. Das maliziöse Verhalten ist also ausschließlich anhand der *Parameterwerte* der Systemcalls feststellbar und somit von den bisher vorgestellten Systemen nicht wahrnehmbar. Somit müssen zur Erkennung dieser Angriffe die aufgezeichneten Systemcall–Argumente mit in die Analyse einbezogen werden.

**Das Tool *LibAnomaly*** Die in den folgenden Abschnitten vorgestellten Arbeiten [Zan06] und [MVKV06] basieren auf dem Tool *LibAnomaly*[10]. Dieses Tool wurde von der Raliable Software Group der Univerity of California, Santa Barbara entwickelte [KMVV03] und implementiert verschiedene Modelle zur Klassifikation von Systemcall–Argumenten als Rahmenkonstrukt zur Erstellung von Anomalieerkennungs–Systemen. Zur Bewertung der "Normalität" eines Systemcall–Arguments werden verschiedene Eigenschaften der Argumente betrachtet. Dazu gehört die Länge eines String–Parameters, die Zeichenverteilung in String–Argumenten, die Struktur eines Arguments in der Form seiner Grammatik und die Gültigkeit übergebener Werte falls diese lediglich aus einer begrenzten Wertemenge stammen dürfen.

### 3.6.1. Parameterwerte und der Aufrufzusammenhang von Systemcalls

Stefano Zanero stellte 2006 in seiner Doktorarbeit [Zan06] einen Ansatz vor, der die Zuverlässigkeit der in *LibAnomaly* implementierten Modelle erhöht und Korrelationen zwischen Systemcall–Sequenzen berücksichtigt. "Normales" Verhalten einer Anwendung wird auf der Basis des Datenaustauschs eines Nutzers mit einem Terminal modelliert.

**Klassifikation von Parameterwerten** Da auch die Parameterwerte eines "normalen" Systemcalls sehr verschieden sein können, werden die in der Trainingsmenge vorhandenen Werte gruppiert. Es werden bottom–up so lange ähnliche Elemente eines Systemcalls zu Clustern zusammengefasst, bis eine definierte Anzahl an Clustern vorliegt oder die Distanzen zwischen den Cluster–Elementen einen vorgegebenen Schwellwert erreichen. Dieses Verfahren wird nur für eine Teilmenge aller Systemcalls durchgeführt. Die Angriffserkennung wird anschließend auch nur auf den Systemcalls dieser Teilmenge durchgeführt.

Zanero definiert Distanzmaße für die vier möglichen Parametertypen, die in der Teilmenge vorkommen können. Als Parametertypen werden Pfadnamen und Dateinamen, diskrete numerische Werte, Ausführungsargumente und User– beziehungsweise Gruppen–IDs genannt. Als Basiswerte für mögliche Distanzmaße werden die Zeichenverteilung, die

[10] http://www.cs.ucsb.edu/~seclab/projects/libanomaly/index.html

Pfadtiefe, die Pfadlänge, die Dateierweiterung, die Anzahl der Vorkommen eines Wertes und die Differenz von Flags und Modi angegeben.

Die Parameterwerte eines zu klassifizierenden Systemcalls werden mit Cluster–Zugehörigkeitswahrscheinlichkeiten versehen. Für einen Parameterwert wird das Cluster gewählt, zu dem dieser mit der größten Wahrscheinlichkeit gehört. Aufgrund des enormen Rechenzeit– und Speicherplatzbedarfs des vorgestellten Verfahrens wird eine Heuristik zur Reduzierung der benötigten Schritte eingeführt.

Die Parameterwerte der Systemcalls des DARPA–Datensatzes (MIT LL) konnten den Clustern mit einer Genauigkeit von annähernd 100% zugeordnet werden.

**Berücksichtigung des Ausführungskontextes** Um bei der Klassifikation eines Systemcalls neben der "Normalität" seiner Argumente auch den Ausführungskontext jedes Systemcalls berücksichtigen zu können, wird ein Markov Modell der Ordnung 1 erstellt. Das Modell soll den Programmfluss darstellen. Ein Markov Modell kann als die einfachste Form eines dynamischen Bayes–Netzes verstanden werden. Bayes–Netzen werden in Abschnitt 4.5 näher erläutert. Das Markov Modell besteht aus einer endlichen Menge von $n$ Zuständen $S = \{s_1, s_2, \ldots, s_n\}$, von denen jeder mit einer üblicherweise mehrdimensionalen Wahrscheinlichkeitsverteilung assoziiert wird. Übergänge zwischen den Zuständen werden durch Transitionswahrscheinlichkeiten bestimmt. Bei einem Markov Modell der Ordnung $K$ hängt die Wahrscheinlichkeit des aktuellen Zustands von den $K$ vorhergehenden Zuständen ab.

Im vorliegenden Fall repräsentieren die Zustände des Modells die Systemcalls bzw. die Cluster der Systemcalls, die während des Clustering–Prozesses entdeckt wurden. Die Transitionswahrscheinlichkeiten reflektieren die Wahrscheinlichkeiten der Übergänge von der einen zur anderen Systemcall–Gruppe. Liegt die aus den Cluster–Zugehörigkeitswerten und dem Wert des Markov Modells zusammengesetzte Wahrscheinlichkeit unter einem festgelegten Anomalie–Schwellwert, wird der entsprechende Prozess als Anomalie gekennzeichnet.

Durch die Berücksichtigung des Aufrufzusammenhangs werden viele Angriffe erkannt, die durch die alleinige Analyse der Parameterwerte keinen Alarm ausgelöst hätten.

### 3.6.2. Kombination von Parametermodellen mit einem Bayes–Netz

Darren Mutz et. al. verfolgen in [MVKV06] einen ähnlichen Ansatz wie Zanero [Zan06]. Dabei werden ebenfalls auf der Basis der *LibAnomaly*–Implementierungen charakteristische Modelle legitimer Systemcall–Argumente verschiedener Applikationen gelernt. Es wird ein hostbasiertes Intrusion Detection System entwickelt, das laufende Anwendungen überwacht, um maliziöses Verhalten zu entdecken. Statt die finale Klassifikation eines Systemcalls durch die Kombination der Wahrscheinlichkeiten einzelner Parametermodelle und die Anwendung eines Schwellwerts vorzunehmen wie in [Zan06], wird ein Bayes–Netz eingesetzt. Im Gegensatz zu [Zan06] wird der Ausführungskontext der Systemcalls *nicht* berücksichtigt.

**Aufbau des Bayes–Netzes zur Kombination der Modelle** Basierend auf den Anomalie–Werten $\{as_i | i = 1 \ldots k\}$ der $k$ Parametermodelle und möglicherweise zusätzlich

vorhandenen Informationen $I$ muss ein Systemcall abschließend durch eine Funktion

$$C_s(as_1, as_2, \ldots, as_k, I) = \{normal, anomal\}$$

klassifiziert werden. Dazu wird ein Bayes–Netz anstelle eines Schwellwerts verwendet. Auf Bayes–Netze wird in Abschnitt 4.5 näher eingegangen.

Das hier verwendete Netz besteht aus einem Wurzelknoten (Hypothesenknoten), der eine Variable mit den Zuständen "normal" und "anomal" repräsentiert. Für jedes Parametermodell wird ein weiterer Knoten hinzugefügt, um die Ausgaben $\{as_i | i = 1 \ldots k\}$ der $k$ Modelle einzubeziehen. Der Wurzelknoten wird mit allen anderen Knoten verbunden. Je nach Domäne können zusätzliche Informationen und Einflussfaktoren, wie zum Beispiel Abhängigkeiten der Modelle untereinander oder Gewichtungen der Modell–Ausgaben, hinzugefügt werden. Das IDS erzeugt einen Alarm, wenn die am Wurzelknoten berechnete Anomalie–Wahrscheinlichkeit hoch genug ist. Dies kann wieder als Schwellwert betrachtet werden, gibt aber im Gegensatz zur echten Schwellwert–Nutzung direkt die Wahrscheinlichkeit einer Anomalie an.

Das vorgestellte Verfahren wird aus Performanzgründen, ähnlich wie in [Zan06], nur auf eine Teilmenge aller Systemcalls angewendet. Die vorhandenen Parametertypen entsprechen denen in [Zan06].

Für ein erstes Experiment wurde ebenfalls wie in [Zan06] der DARPA 1999 MIT LL Datensatz verwendet. Das implementierte IDS erkannte alle Angriffe und erzeugte keine falschen Alarme. Bei einem Test mit Programmen in denen keine Angriffe enthalten waren, wurden einige falsche Alarme erzeugt, deren Anteil allerdings unter 0,5% aller evaluierten Systemcalls lag. Bei einem Vergleich mit anderen Ansätzen, wie zum Beispiel [FHSL96] und [KFH05], wurde festgestellt, dass das in dieser Arbeit vorgestellte System bei der Erkennung aller Angriffe die wenigsten falschen Alarme erzeugt. Die endgültige Klassifikation von Systemcalls mit Hilfe eines Bayes–Netzes lieferte durchweg bessere Ergebnisse als die Verwendung eines Schwellwerts.

Auch bei der Durchführung des Trainings auf einem System im normalen Betrieb und einem Server bewies das implementierte IDS in der Betriebsphase eine zuverlässige Angriffserkennung bei Erzeugung weniger falscher Alarme. Die Anzahl der durch das IDS zu analysierenden Systemcalls pro Sekunde ist laut [MVKV06] zu gering, um bemerkbare Performanz–Einbußen hervorzurufen.

### 3.6.3. Aussagekraft von Parameterwerten

Das in [Zan06] präsentierte Modell liefert eine zuverlässige Klassifizierung normaler und anomaler Verhaltensweisen auf der Grundlage von Systemcall–Argumenten. Zudem können auf die vorgestellte Art und Weise detaillierte Angaben darüber gemacht werden, *welche* Systemcalls nicht dem normalen Programmablauf entsprechen.

In [MVKV06] wird anstelle eines Schwellwerts zur Verknüpfung der Parametermodell–Ausgaben ein Bayes–Netz verwendet. Das Verfahren liefert bessere Performanzen als einige ähnliche Arbeiten.

Parameterwerte von Systemcalls enthalten demnach zusätzliche, wichtige Informationen zur Erkennung von Angriffen. Allerdings ist die Analyse dieser Werte sehr aufwendig und speicherplatzintensiv.

In den Arbeiten von Zanero und Mutz et. al wird vorausgesetzt, dass sich Angriffe in den Argumenten der Systemcalls bemerkbar machen. Außerdem wird angenommen, dass sich die Parameterwerte während der Ausführung eines Angriffs wesentlich von den Parameterwerten während der normalen Programmausführung unterscheiden. Sind diese Voraussetzungen nicht gegeben, werden die Angriffe nicht erkannt.

Wie bei allen anomalieerkennenden IDS–Ansätzen gilt auch hier, dass das normale Verhalten der Applikation in den Trainingsdaten möglichst vollständig abgedeckt werden sollte, da sonst viele falsche Alarme erzeugt werden. Zudem dürfen die Trainingsdaten keine Anomalien enthalten, da entsprechende Angriffe sonst nicht erkannt werden. Die Modelle müssen kontinuierlich an geänderte Modalitäten in der Systemnutzung angepasst werden.

## 3.7. Klassifikation von Schadprogramm–Familien

Wie bei den bisher vorgestellten Arbeiten ist das Ziel der meisten Ansätze im Bereich der Intrusion Detection das Lösen des Zweiklassenproblems der Klassifizierung "normaler" und "anomaler" Traces. Die Vielfalt und die Menge unterschiedlicher Schadprogramme beeinträchtigt die Effektivität dieser klassischen Systeme jedoch stark. Das enorme Aufkommen immer neuer Schadprogramme erfordert eine ständige, schnelle Aktualisierung von Antivirus–Produkten und die Bewahrung eines Überblicks über die Entwicklung. Um der Erkennung durch Antivirus–Produkte zu entgehen, werden immer mehr Schadprogramm–Varianten produziert. Einige Schadprogramm–Familien weisen zehntausende bekannte Varianten auf, die jedoch häufig ähnliche Muster enthalten. Diese Muster lassen auf die Herkunft und das Ziel schließen. Die in diesem Abschnitt vorgestellten Ansätzen [KAT07] und [RHW+08] versuchen die Muster zu nutzen, um Schadprogramm–Familien zu klassifizieren und Verhaltensweisen zu lernen. Es werden maschinelle Lernverfahren eingesetzt, um das vorliegende *Mehrklassenproblem* zu lösen. Die Problemstellung in dieser Diplomarbeit ist dieselbe und das Vorgehen ist ähnlich.

### 3.7.1. Reduzierung der Trainingsmenge durch Clustering

Khan, Awad und Thuraisingham veröffentlichten 2007 in [KAT07] mit CT-SVM (Clustering Tree based on SVM) einen Ansatz zur Klassifikation netzwerkorientierter Angriffe mit der Stützvektormethode (SVM). Die SVM wurde bereits in den Ansätzen von Yao et. al. [YZF06] und Kang et. al. [KFH05] zur Anomalieerkennung auf Systemcall–Traces eingesetzt. Diese Arbeiten wurden in den Abschnitten 3.4.2 und 3.5.1 beschriebenen. Die SVM passt eine separierende Hyperebene in den Merkmalsraum ein, die die gegebenen Beispiele gemäß ihrer Klassenzugehörigkeiten bestmöglich voneinander trennt. Eine detaillierte Beschreibung der Stützvektormethode folgt in Abschnitt 4.4.

**Reduzierung der Laufzeit** Die zu klassifizierenden Beispiele liegen auch in [KAT07] wieder in der Form von Vektoren vor. Zur Reduzierung der Anzahl an Beispielen, die der SVM als Trainingsmenge übergeben werden und somit zur Reduzierung der Rechenzeit, wird ein Clusteringverfahren angewendet. Für jede in den Daten vorhandene Klasse wird mit Hilfe des so genannten DGSOT–Algorithmus (Dynamic Growing Self-Organizing

Tree) ein hierarchischer Cluster–Baum gebildet, um die Beispiele zu finden, die die Stützvektoren der SVM bilden und somit die Klassengrenzen definieren. Es wird iterativ top-down in einzelnen Schritten vorgegangen. Jeder Knoten eines Cluster–Baumes repräsentiert ein Cluster aus Trainingsbeispielen. Der Referenzvektor eines Clusters ist der Datenmittelpunkt der in diesem Cluster enthaltenen Beispiele. Die SVM wird auf den aktuellen Blättern der Cluster–Bäume trainiert. Die Stützvektoren dieses Lernschritts bestimmen die Blätter, die durch das Clusteringverfahren weiter expandiert werden. Für jeden Stützvektor–Knoten werden Kinder hinzugefügt und die Beispiele des Knotens werden auf die Kinder verteilt. Anschließend erfolgt ein erneutes SVM–Training und die Prozedur wird mit den Stützvektoren dieses Modells wiederholt. So wird der Baum nur an den Klassengrenzen weiter expandiert und die optimale Trennung wird sukzessive angenähert. Von den Grenzen entfernte Knoten werden nicht weiter betrachtet. Das Verfahren terminiert, wenn eine vorgegebene Anzahl an Stützvektoren gegeben ist.

Wie bereits in [LV02], [KFH05], [Zan06] und [MVKV06] im Rahmen der Anomalieerkennung wird das Modell auf die DARPA 1998 Daten des MIT LL angewendet. Dieser Datensatz enthält 4 Angriffskategorien. Es liegen also diese 4 Klassen und die Klasse der "normalen" Programmausführungen vor, so dass ein 5–Klassen–Problem zu lösen ist. Die nominalen Werte der 41 Merkmale werden zunächst in eine Bitvektor–Darstellung transformiert. Khan et. al. wenden die LIBSVM für SVSM Implementierung an. Sie nutzen die $\nu$-SVM [SSWB98] mit einem RBF–Kern (siehe Gleichung (4.15)). Der Parameter $\nu$ ermöglicht die Beeinflussung der Anzahl an Stützvektoren und wurde auf 0,001 gesetzt. Zur Lösung des Mehrklassenproblems wird ein one–against–one Schema implementiert, da diese Herangehensweise die Rechenzeit gegenüber dem one–against–all Schema reduzierte. Die durchschnittliche Trainingszeit des Verfahrens verkürzte sich durch die Anwendung des vorgestellten Clusterings im Vergleich zur Anwendung der reinen SVM um etwa 24%. Dabei verbesserte sich die Genauigkeit des Verfahrens um ca. 12% und lag somit bei 69,8%.

### 3.7.2. Verschiedene Systemcall–Detaillierungsgrade zur Klassifikation

Rieck, Holz, Willems, Düssel und Laskov [RHW+08] haben 2008 ebenfalls Schadprogramm–Familien klassifiziert. Zur Erstellung eines Datensatzes wurden hier Binärdateien von Schadprogrammen gesammelt und in dem kommerziellen Tool *CWSandbox*[11] ausgeführt. Dieses Tool erstellt Logdateien, in denen die während der Ausführung aufgerufenen Systemcalls aufgezeichnet werden. Die Schadprogramme und ihre entsprechenden Logdateien wurden mit Hilfe der Anti–Virus–Maschine *Avira AntiVir*[12] gelabelt und die Traces der 14 Schadprogramm–Familien mit den häufigsten Vorkommen wurden ausgewählt. Der entstandene Datensatz enthält 10.072 Beispiele.

**Vektordarstellung der Daten** Eine *CWSandbox*–Logdatei wird auch hier wieder als Textdokument betrachtet. Jeder in dem Bericht enthaltene Systemcall stellt einen String dar, der den Systemcallnamen und eine Liste der Parameter und Werte enthält. Die Strings werden als Merkmale betrachtet. Die so entstehende Merkmalsmenge $\mathcal{F}$ kann

[11] http://www.cwsandbox.org
[12] http://www.free-av.de

sehr speziell werden, da kleine Abweichungen in unterschiedlichen Strings und somit in unterschiedlichen Vektorraum–Dimensionen resultieren. Laut [RHW+08] sind unterschiedliche Strings in den Traces häufig durch eine Verschleierungstechnik begründet. Daher wird jeder Systemcall durch mehrere Strings mit unterschiedlichen Detaillierungsgraden dargestellt. Zum Beispiel werden für den Systemcall "`copy_file`" die folgenden Merkmale erzeugt:

$$\texttt{copy\_file} \longrightarrow \begin{cases} \texttt{copy\_file\_1}(\texttt{srcfile} = \texttt{A}, \texttt{dstfile} = \texttt{B}) \\ \texttt{copy\_file\_2}(\texttt{srcfile} = \texttt{A}) \\ \texttt{copy\_file\_3}() \end{cases}$$

Zur Transformation dieser Textdarstellung in die Vektordarstellung wird wieder das in Abschnitt 3.4 beschriebene Vector Space Modell genutzt. Für jeden String $s \in \mathcal{F}$ und jeden Bericht $x \in \mathcal{X}$ mit $\mathcal{X}$ der Menge aller Berichte wird die Häufigkeit $f(x, s)$ als Anzahl der Vorkommen von $s$ in $x$ bestimmt. Es wird also die in Definition 3.1 beschriebene Termfrequenz $TF$ verwendet. Formal wird eine Funktion $\phi$ implementiert, die einen Bericht in einen $|\mathcal{F}|$–dimensionalen Vektorraum abbildet, indem die Häufigkeiten aller Strings in $\mathcal{F}$ betrachtet werden:

$$\begin{aligned} \phi : \mathcal{X} &\to \mathbb{R}^{|\mathcal{F}|}, \\ x &\mapsto (f(x, s))_{s \in \mathcal{F}}. \end{aligned}$$

**Klassifikation mit der SVM** Als Klassifikationsverfahren wird in [RHW+08] die *Shogun*–Implementierung der SVM benutzt. Die Datenmenge wird zufällig in 3 Partitionen unterteilt, so dass eine Trainings–, eine Validierungs– und eine Testmenge entstehen. Auf der Trainingsmenge wird durch ein one–against–all Schema ein individuelles SVM–Modell für jede der 14 Schadprogramm–Familien gelernt. Es wird jeweils das Modell ausgewählt, das auf dem Validierungsdatensatz die höchste Genauigkeit liefert. Zur Kombination der verschiedenen Einzelmodelle wird die *Maximale Distanz* verwendet. Das bedeutet, dass einem neuen Bericht die Klasse zugeordnet wird, deren Klassifizierer den höchsten Wert liefert, also das Label der am besten trennenden Hyperebene. Die Güte des gesamten Modells wird mittels des Testdatensatzes evaluiert. Das Verfahren wird inklusive der zufälligen Partitionierung des Datensatzes 5 Mal wiederholt und die Ergebnisse jedes Durchlaufs werden gemittelt.

Die Genauigkeit pro Schadprogramm–Familie liegt bei diesem Ansatz zwischen 70% und 100%. Es werden durchschnittlich 88% der Beispiele der richtigen Schadprogramm–Familie zugewiesen.

Zur Bewertung der *Vorhersagegenauigkeit* des Modells wird der Testdatensatz durch Beispiele ersetzt, die bei der ersten Anfrage bei *Avira Antivir* nicht gelabelt werden konnten, bei einem erneuten Scan nach 4 Wochen aber erkannt wurden. Der resultierende Datensatz enthält 3.139 Beispiele der 14 gelernten Schadprogramm–Familien. Das Klassifikationsverfahren wird auch hier 5 Mal durchgeführt. Die durchschnittliche Genauigkeit pro Schadprogramm–Familien liegt bei diesem Experiment zwischen 20% und 100%. Es werden durchschnittliche 69% der Beispiele der richtigen Schadprogramm–Familie zugeordnet.

**Modellerweiterung zur Identifizierung unbekannter Verhaltensweisen** Der bis hierhin vorgestellte Ansatz wird noch erweitert, um auch *unbekanntes Verhalten* identifizieren zu können. Unbekanntes Verhalten kann in normalem Systemverhalten oder in Berichten nicht trainierter Schadprogramm–Familien begründet sein.

Zur Differenzierung von bekannten und unbekannten Beispielen wird anstelle der maximalen Distanz zur Bestimmung der vorliegenden Familie eine Wahrscheinlichkeitsschätzung für jede Familie verwendet.

Die bereits beschriebenen Experimente werden zunächst mit diesem erweiterten Klassifikationsverfahren wiederholt. Dabei beträgt die Genauigkeit des Gesamtmodells nur noch 76%. Allerdings sinkt die Anzahl der Beispiele, die einer falschen Klasse zugeordnet werden, da diese Beispiele nun als unbekannt klassifiziert werden. Bei der Klassifikation vorher nicht berücksichtigter Schadprogramm–Familien wird der Großteil der Beispiele als unbekannt gelabelt. Bei der Klassifikation von 498 gutartigen Berichten werden alle Beispiele als unbekannt identifiziert.

**Erklärung der Modelle** Da jede Komponente $w_i$ des Normalenvektors $\vec{w}$ den Beitrag eines Merkmals zur Unterscheidung der Klassen ausdrückt, können diese Werte zur Erklärung und Nachvollziehbarkeit der erstellten Modelle herangezogen werden. Dimensionen mit hohen Werten weisen auf einen starken Einfluss auf die Trennung hin. In [RHW+08] werden die Komponenten $w_i$ daher nach ihrer Größe sortiert, um einen Überblick über das spezifische Verhalten einzelner Schadprogramm–Familien und über die Ähnlichkeiten der Familien untereinander zu erhalten. Des Weiteren ermöglichen diese Werte einen Einblick in die trainierte Entscheidungsfunktion.

**Probleme des Ansatzes** Die Aufzeichnung der Systemcalls einer einzigen Programmausführung, wie in dem vorliegenden Fall, enthält offensichtlich häufig bereits genug Informationen, um Schadprogramm–Familien gut klassifizieren zu können. In [RHW+08] wird dennoch darauf hingewiesen, dass das aufgezeichnete Verhalten unvollständig ist. Darüber hinaus beinhalten die betrachteten Berichte keine mehrfach vorkommenden Aktivitäten, so dass Nachahmungsangriffe nur schwer erkannt werden können. Der Ansatz ist zudem nicht dazu in der Lage, die Menge der unbekannten Beispiele weiter zu differenzieren.

### 3.7.3. Systemcall–Logdateien und Schadprogramm–Familien

Die vorgestellten Ansätzen [KAT07] und [RHW+08] nutzen die in eine Vektordarstellung transformierten Systemcall–Logdateien als Datengrundlage zur Klassifikation von Schadprogramm–Familien. In beiden Arbeiten wird die Stützvektormethode als Lernverfahren eingesetzt. In [RHW+08] werden unterschiedliche Detaillierungsgrade der Systemcalls berücksichtigt. Parameter und ihre Werte werden folglich teilweise mit einbezogen. In [KAT07] wird die Trainingsmenge durch die Kombination eines Clusteringverfahrens mit dem Training der SVM reduziert. Dadurch kann die Trainingszeit der SVM auf großen Datensätzen um durchschnittlich 24% gesenkt werden. Parameterwerte werden dabei nicht berücksichtigt. Beide Ansätze liefern zufriedenstellende Ergebnisse bezüglich der Genauigkeit der Modelle.

Die Arbeiten zeigen, dass sich Systemcall–Logdateien und die Vektordarstellung nicht nur zur Anomalieerkennung eignen, sondern auch eine passende Datengrundlage zur Klassifikation von Schadprogramm–Familien darstellen.

# 4. Eine Auswahl überwachter Lernverfahren

Im Bereich der Intrusion Detection gibt es bereits eine Vielzahl von Ansätzen, die Data Mining Techniken zur Datenanalyse nutzen. Einige grundlegende Arbeiten zu diesem Thema wurden in dem vorangegangenen Kapitel 3 vorgestellt. In Abschnitt 3.1 wurde ein Überblick über das Gebiet der Wissensentdeckung, des Data Minings und maschineller Lernverfahren gegeben.

In dieser Diplomarbeit werden maschinelle Lernverfahren zur Klassifikation von Schadprogramm–Familien auf der Grundlage von Systemcall–Logdateien in der Vektordarstellung eingesetzt. Die Transformation der Logdateien in diese Darstellungsform wird in Kapitel 5 geschildert. Da die Schadprogramm–Familien (die Klassen) der vorliegenden Beispiele bekannt sind, werden Algorithmen des *überwachten* Lernens (siehe Abschnitt 3.1) verwendet. Das vorliegende Kapitel dient der Beschreibung der Lernverfahren, die bei den in Kapitel 5 dokumentierten Experimenten angewendet werden und zum Teil bereits in den beschriebenen verwandten Arbeiten erfolgreich zur Erkennung von Angriffen genutzt wurden. Da im Folgenden ein Mehrklassenproblem zu lösen ist werden Verfahren vorgestellt, die auf derartige Probleme anwendbar sind.

## 4.1. Instanzbasierte Klassifikation

Instanzbasierte Klassifikationsverfahren zählen zu den einfachsten maschinellen Lernalgorithmen. Die Trainingsbeispiele werden als Vektoren $\vec{x}_i$ in einem $m$–dimensionalen Merkmalsraum $\mathbb{R}^m$ dargestellt. Die Lernphase besteht aus dem einfachen Speichern dieser Merkmalsvektoren und ihrer Klassenzugehörigkeiten. Dieses Vorgehen wird auch als "lazy" (faul) bezeichnet.

Ein neues Beispiel wird ebenfalls als Merkmalsvektor im $\mathbb{R}^m$ dargestellt. Mittels einer Ähnlichkeits– oder Distanzfunktion wird die Ähnlichkeit/Distanz des Beispiels zu allen gespeicherten Merkmalsvektoren der Trainingsmenge berechnet und aufgrund dieser Werte klassifiziert. Bei der im Folgenden verwendeten Implementierung dieses Rahmenkonstrukts wird die Euklidische Distanz verwendet. Andere Distanzmaße wie das in [LV02] (Abschnitt 3.4.1) genutzte Cosinus–Maß sind jedoch ebenfalls möglich.

**Definition 4.1 (Euklidische Distanz)** *Die* euklidische Distanz *zwischen zwei Vektoren $\vec{x}_i$ und $\vec{x}_j$ wird berechnet durch*

$$d(\vec{x}_i, \vec{x}_j) = \|\vec{x}_i - \vec{x}_j\| = \sqrt{(x_{i_1} - x_{j_1})^2 + \ldots + (x_{i_m} - x_{j_m})^2} = \sqrt{\sum_{n=1}^{m} (x_{i_n} - x_{j_n})^2}.$$

Die Klassifikation eines Beispiels erfolgt durch die *k–Nearest–Neighbor* Funktion ($k$NN). Diese Funktion wählt die unter den $k$ *nächsten Nachbarn* am häufigsten vorkommende Klasse. Dabei ist $k$ eine positive ganze Zahl.

Abbildung 4.1 zeigt ein Beispiel für die Klassifikation durch die $k$NN–Funktion mit $k = 3$ im zweidimensionalen Fall. Das durch den roten Punkt gekennzeichnete Beispiel wird der Klasse der grauen Punkte zugeordnet, da diese Klasse unter den 3 nächsten Nachbarn am häufigsten vorkommt.

Eine gute Wahl von $k$ ist abhängig von den Daten. Im Allgemeinen reduziert ein großes $k$ den Effekt von vorhandenem Rauschen in den Daten, führt jedoch auch zu weniger eindeutigen Grenzen zwischen den einzelnen Klassen. Bei $k = 1$ hingegen wird ein Beispiel mit der Klasse des nächsten Nachbarn, also der Klasse des ähnlichsten Beispiels gelabelt. Es wird sozusagen "auswendig gelernt", was mit hoher Wahrscheinlichkeit zu großen Fehlern auf neuen, unbekannten Daten führt. Grundsätzlich kann die Genauigkeit des Algorithmus durch Rauschen in den Daten oder irrelevante Merkmale stark beeinträchtigt werden.

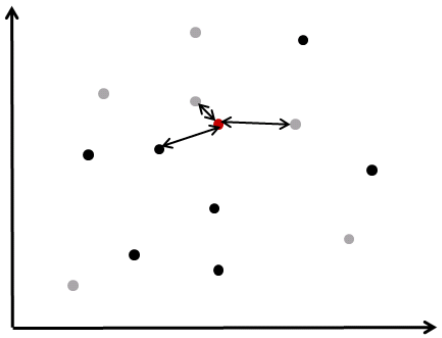


Abbildung 4.1.: Ein Beispiel für die Klassifikation durch die $k$NN–Funktion mit $k$=3 im 2–dimensionalen Fall. Es liegen 2 Klassen vor, deren Trainingsinstanzen durch hellgraue und durch schwarze Punkte gekennzeichnet sind. Der rote Punkt symbolisiert das zu klassifizierende Beispiel. Da die Menge der 3 Nachbarn 2 graue Trainingsinstanzen und nur 1 schwarze Trainingsinstanz enthält, wird das Beispiel der grauen Klasse zugeordnet.

Ein weiterer Nachteil des $k$–Nearest–Neighbor–Verfahrens ist die Tendenz zur Dominierung der Klassifikation eines neuen Vektors durch Klassen mit vielen Beispielen, da Beispiele dieser Klassen mit großer Wahrscheinlichkeit häufig unter den $k$ nächsten Nachbarn sind. Eine Möglichkeit dieses Problem zu umgehen ist die Einbeziehung der Distanzen der $k$ nächsten Nachbarn und die Klassifikation des neuen Vektors unter Berücksichtigung dieser Distanzen.

### Eine speicherreduzierende Erweiterung

Zur Reduzierung des durch den vorgestellten $k$NN–Algorithmus benötigten Speichers gibt es zwei grundlegende Strategien, die nicht mehr *alle* Beispiele der Lernmenge speichern. *Instanz–filternde* Algorithmen verwerfen richtig klassifizierte Beispiele in der Annahme,

dass gerade die falsch klassifizierten Beispiele zusätzliche, zur Klassentrennung wertvolle Informationen enthalten. *Instanz–mittelnde* Verfahren ersetzen die klassifizierende Instanz mit dem Durchschnitt des Instanz–Beispiels und des neu klassifizierten Beispiels.

Eine Erweiterung dieser Strategien zur besseren Handhabung von Rauschen in den Trainingsdaten wurde in [AK91] vorgestellt und liegt der in Kapitel 5 verwendeten Implementierung zugrunde. Dabei werden die von einer Instanz klassifizierten Datensätze für alle gespeicherten Instanzen behalten. Es wird ein so genanntes Konzept $C$ erstellt, das die klassifizierenden Instanzen enthält. Für ein Trainingsbeispiel $\vec{x}$ wird der nächste *akzeptierte* Nachbar $n$ in $C$ gesucht. Eine Instanz wird "akzeptiert", wenn seine Klassifikationsgenauigkeit (Accuracy, siehe auch Definition 4.10) statistisch gesehen erheblich besser ist als die aktuell beobachtete Häufigkeit der Klasse. Wird das neue Trainingsbeispiel $\vec{x}$ durch den akzeptierten Nachbarn $n$ richtig klassifiziert, wird es verworfen. Anderenfalls wird $\vec{x}$ dem Konzept $C$ hinzugefügt.

Anschließend werden die klassifizierten Datensätze aller gespeicherten Instanzen aktualisiert, die dem Trainingsbeispiel mindestens so ähnlich sind wie sein nächster *akzeptierter* Nachbar. In einem letzten Schritt werden alle gespeicherten Instanzen, die fehlerhaft erscheinen und folglich wahrscheinlich Rauschen darstellen, aussortiert. Das können zum Beispiel Instanzen sein, deren Klassifikations–Performanz nach mehreren Klassifikationsversuchen niedrig ist.

Bei der Anwendung dieses Verfahrens auf Systemcall–Logdateien werden die Schadprogramm–Familien folglich aufgrund von Distanzen ihrer Beispielvektoren zueinander erkannt.

## 4.2. Klassifikation durch Regeln

Ein weiterer Ansatz zur Klassifikation gelabelter Daten ist die Suche nach gültigen Regeln, die Abhängigkeiten zwischen den Merkmalen gegebener Beobachtungen und deren Klassenzugehörigkeit beschreiben. Regeln besitzen einen Bedingungsteil "`A`", der Eigenschaften von Merkmalen durch UND–Verknüpfungen beschreibt und eine Konklusion "`B`", die die Klasse bei Gültigkeit der in `A` beschriebenen Eigenschaften angibt. Regeln können zum Beispiel durch Formulierungen wie "`WENN A DANN B`" oder durch Ausdrücke der Form "$A \rightarrow B$" beschrieben werden.

### Die Entscheidungstabelle

Zur übersichtlichen und vollständigen Darstellung eines komplexen Regelwerks aus mehreren Bedingungen und der jeweils zugehörigen Klasse werden häufig *Entscheidungstabellen* aufgestellt.

Das Schema einer Entscheidungstabelle besteht aus einem Bereich, der die zu berücksichtigenden Merkmale enthält und einem Bereich, der die Klasse enthält. Die vorliegenden Beispiele, die für jedes Merkmal und für die Klassenzugehörigkeit einen Wert enthalten, stellen den Inhalt der Tabelle dar. Zur Erstellung einer Entscheidungstabelle muss festgelegt werden, welche Merkmale im Tabellenschema enthalten sein sollen und welche Beispiele in der Tabelle gespeichert werden.

Angenommen, die Merkmalsmenge eines Klassifikationsproblems bestehe aus den Merkmalen $s_1$=`openFile`, $s_2$=`openKey` und $s_3$=`queryValue` mit den möglichen Werten 1,2 und 3. Die Menge der Beispiele $\{\vec{x_1}, \vec{x_2}, \vec{x_3}, \vec{x_4}, \vec{x_5}\}$ sei gegeben durch

$$\vec{x}_1 = \begin{pmatrix} 1 \\ 2 \\ 1 \end{pmatrix}, \ \vec{x}_2 = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}, \ \vec{x}_3 = \begin{pmatrix} 1 \\ 3 \\ 1 \end{pmatrix}, \ \vec{x}_4 = \begin{pmatrix} 1 \\ 3 \\ 3 \end{pmatrix}, \ \vec{x}_5 = \begin{pmatrix} 1 \\ 3 \\ 2 \end{pmatrix}$$

mit den Klassen $y_1 = Allaple$, $y_2 = Delf$ und $y_3 = y_4 = y_5 = Banker$.

Tabelle 4.1 stellt die Entscheidungstabelle dieser Beispiele dar, wenn das Tabellenschema alle Merkmale enthält und alle Beispiele in der Tabelle gespeichert werden. Offensichtlich ist das Merkmal `openFile` irrelevant und trägt nicht zu der Auswahl eines

| | `openFile` | `openKey` | `queryValue` | Label |
|---|---|---|---|---|
| $x_1$ | 1 | 2 | 1 | *Allaple* |
| $x_2$ | 1 | 2 | 3 | *Delf* |
| $x_3$ | 1 | 3 | 1 | *Banker* |
| $x_4$ | 1 | 3 | 3 | *Banker* |
| $x_5$ | 1 | 3 | 2 | *Banker* |

Tabelle 4.1.: Beispiel einer Entscheidungstabelle.

Labels bei. Das Merkmal `queryValue` hat keinen Einfluss auf die Klasse *Banker*. Diese kann anhand des Merkmals `openKey` mit dem Wert 3 bereits vorhergesagt werden. Die vollständige Entscheidungstabelle kann somit zu der in Tabelle 4.2 dargestellten Entscheidungstabelle zusammengefasst werden.

Das "-" deutet an, dass dieses Merkmal irrelevant ist. Das Merkmal `openFile` wurde

| `openKey` | `queryValue` | Label |
|---|---|---|
| 2 | 1 | *Allaple* |
| 2 | 3 | *Delf* |
| 3 | - | *Banker* |

Tabelle 4.2.: Beispiel einer vereinfachten Entscheidungstabelle.

entfernt. Durch die dritte Zeile der Tabelle 4.2 werden die Beispiele $\vec{x_3}$ bis $\vec{x_5}$ abgedeckt. Eine Entscheidungstabelle kann demnach häufig durch das Entfernen von Redundanzen und unlogischen oder inkonsistenten Regeln und durch die Konsolidierung von Regeln vereinfacht und verkleinert werden. Die zu dem Beispiel gehörenden Regeln lauten

`WENN openKey` =2 `UND queryValue` =1 `DANN` *Allaple*
`WENN openKey` =2 `UND queryValue` =3 `DANN` *Delf*
`WENN openKey` =3 `DANN` *Banker*.

**DTM** Die in Kapitel 5 verwendete Implementierung der Entscheidungstabelle basiert auf dem in [Koh95] beschriebenen Ansatz, bei dem eine DTM (Decision Table Majority)

verwendet wird. Die DTM ist eine Entscheidungstabelle mit einer Default–Abbildung der Beispiele auf die mehrheitlich vorhandene Klasse. Ein neues Beispiel $\vec{x}_i$ wird folgendermaßen klassifiziert:

Sei $\mathcal{X}^*$ die Menge der gelabelten Instanzen in der Tabelle, die mit der gegebenen Instanz $\vec{x}_i$ exakt übereinstimmen. Wenn $\mathcal{X}^* = \emptyset$, wird die in der Tabelle mehrheitlich vorhandene Klasse (Majority) zurückgegeben. Anderenfalls wird die in $\mathcal{X}^*$ mehrheitlich vorhandene Klasse zurückgegeben.

Entscheidungstabellen können also auch als eine einfache Form von Nearest–Neighbor–Klassifizierern betrachtet werden, bei denen die Ähnlichkeitsfunktion auf die Rückgabe exakter Übereinstimmungen des zu klassifizierenden Beispiels mit einer gespeicherter Instanz beschränkt ist. Das Nearest–Neighbor Verfahren wurde in dem vorangegangenen Abschnitt 4.1 beschrieben.

**Die optimale Merkmalsmenge** Zur Aufstellung einer Entscheidungstabelle wird in [Koh95] zunächst eine optimale Teilmenge der Merkmale gesucht, die im Tabellenschema enthalten sein soll. Für eine gegebene Zielfunktion $f : \vec{x}_i \mapsto y_i$ und eine Hypothesenklasse $\mathcal{H}$ minimiert die optimale Merkmalsmenge $\mathcal{F}^* \in \mathcal{F}$ den Fehler bezüglich der Zielfunktion $f$. Da $f$ nicht bekannt ist, wird der Fehler durch die Anwendung des Verfahrens auf eine unabhängige Testmenge $\mathcal{T}$ geschätzt und für eine Hypothese $h \in \mathcal{H}$ definiert als

$$\widehat{err}(h, \mathcal{T}) = \frac{1}{|\mathcal{T}|} \sum_{(\vec{x}_i, y_i) \in \mathcal{T}} L(h(\vec{x}_i), y_i).$$

$L$ beschreibt eine Fehlerfunktion mit

$$L(h(\vec{x}_i), y_i) = \begin{cases} 0 & \text{wenn } h(\vec{x}_i) = y_i \\ 1 & \text{sonst,} \end{cases}$$

also $L(h(\vec{x}_i), y_i) = [\![h(\vec{x}_i) \neq y_i]\!]$.

Sei $\mathcal{F} = \{s_1, \ldots, s_m\}$ die geordnete Menge der Merkmale (Systemcalls), $\mathcal{Y}$ die Menge der Klassen (Schadprogramm–Familien) und $\mathcal{X}$ die Menge der Beispiele über den Merkmalen in $\mathcal{F}$ und den Klassen $\mathcal{Y}$. Für eine gegebene Teilmenge der Merkmale $\mathcal{F}' \subseteq \mathcal{F}$ bildet die Menge $\mathcal{F}'$ das Schema der Entscheidungstabelle und der Inhalt besteht aus allen Beispielen in $\mathcal{X}$, abgebildet auf $\mathcal{F}'$. Das Ziel des Algorithmus ist folglich das Finden der Menge $\mathcal{F}^*$, für die

$$\mathcal{F}^* = \arg\min_{\mathcal{F}' \subseteq \mathcal{F}} err(DTM(\mathcal{F}', \mathcal{X}), f)$$

gilt. $DTM(\mathcal{F}', \mathcal{X})$ bezeichnet eine Entscheidungstabelle mit dem Schema $\mathcal{F}'$ und allen Beispielen $\mathcal{X}$, abgebildet auf $\mathcal{F}'$. Unter der Annahme, dass alle Instanzen der Trainingsmenge $\mathcal{X}$ in der Entscheidungstabelle gespeichert werden, ist $\mathcal{F}^*$ also die optimale Merkmalsmenge für die DTM.

Der Raum möglicher optimaler Merkmalsmengen $\mathcal{F}^*$ enthält bei $m$ Merkmalen $2^m$ Untermengen. Da dieser Raum zu groß ist, um ihn vollständig zu durchsuchen, transformiert Kohavi [Koh95] das Problem in eine Zustandsraumsuche und benutzt die Bestensuche.

Dabei stellen Teilmengen der Merkmale die Zustände dar und Operatoren können Merkmale hinzufügen oder löschen. Der Startzustand kann entweder die Menge aller Merkmale oder die leere Menge sein. Da die optimale Teilmenge gesucht wird, gibt es keinen Zielzustand. Daher wird die Trainingsmenge $\mathcal{X}$ in $k$ Testmengen $\mathcal{T}$ zerlegt und jede Teilmenge $\mathcal{F}'$ wird durch die Klassifikation dieser Testmengen $\mathcal{T}$ und die Schätzung der Genauigkeit (Accuracy siehe auch Definition 4.10) evaluiert. Der Algorithmus terminiert, wenn nach einer festgelegten Anzahl an Knotenexpansionen kein Knoten mit besserer (geschätzter) Genauigkeit als der aktuell besten erreicht wird. Für weitere Details des Algorithmus und der Festlegung der Anzahl durchzuführender Knotenexpansionen sei an dieser Stelle auf [Koh95] verwiesen.

Das Verfahren wurde ursprünglich zur Klassifikation von Beispielen mit diskreten Merkmalswerten entwickelt. Obwohl in dem vorliegenden Fall der Klassifikation von Systemcall–Logdateien beispielsweise durch die $TF$– oder die $TFIDF$–Werte kontinuierliche Merkmalswerte gegeben sind, können mit diesem Verfahren gute Ergebnisse erzielt werden. Laut [Koh95] ignoriert der Algorithmus in solchen Fällen kontinuierliche Merkmale oder nutzt Merkmale mit wenigen Werten.

## 4.3. Entscheidungsbäume

Entscheidungsbäume können als alternative Darstellung eines Regelwerks angesehen werden. Ein Entscheidungsbaum–Lerner betrachtet die Merkmale eines Klassifizierungsproblems als Information. Die Merkmale bilden die Knoten des Baumes, die Merkmalsausprägungen bilden die Kanten und die Blätter die Klassen. Somit wird in jedem Knoten des Baumes ein Merkmal abgefragt und für die möglichen Merkmalswerte werden Entscheidungsalternativen angegeben. Die inneren Knoten eines Entscheidungsbaums bilden folglich selbst wiederum die Wurzel eines weiteren Entscheidungsbaums. Geometrisch betrachtet zerlegt ein Entscheidungsbaum den Merkmalsraum orthogonal zu den Merkmalsachsen. Bei $k$ Blättern entstehen dabei $k$ nicht überlappende Partitionen $P_1, \ldots, P_k$, wobei die $k$–te Partition dem $k$–ten Blatt entspricht.

Der linke Teil der Abbildung 4.2 zeigt ein einfaches Beispiel für einen Entscheidungsbaum bei 2 Merkmalen und 3 Klassen. Die entsprechende Partition des Merkmalsraums ist im rechten Teil der Abbildung zu sehen. Dort sind die Beispiele unterschiedlicher Klassen durch verschiedene Arten von "Punkten" gekennzeichnet und die Partitionen sind mit den Klassennamen beschriftet.

Die durch einen Entscheidungsbaum definierten Klassifikationsregeln lassen sich ablesen, indem der Pfade von der Wurzel des Baumes zu den Blättern verfolgt wird. Sie lauten für das Beispiel aus Abbildung 4.2

`WENN openKey` $> 5$ `DANN` *Banker*,
`WENN openKey` $\leq 2$ `DANN` *Delf*,
`WENN openKey` $\leq 5$ `UND openKey` $> 2$ `UND queryValue` $\leq 7$ `DANN` *Allaple*,
`WENN openKey` $\leq 5$ `UND openKey` $> 2$ `UND queryValue` $> 7$ `DANN` *Delf*.

Ein Algorithmus zum Lernen eines Entscheidungsbaums sollte einen Baum mit möglichst wenigen Knoten aufstellen, der die gegebenen Beispiele der Lernmenge nach ihren

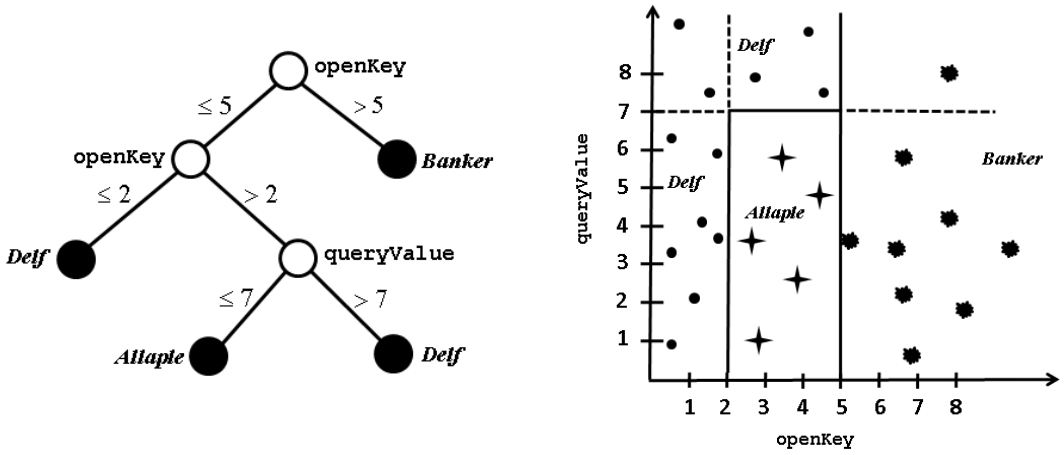



Abbildung 4.2.: Ein einfaches Beispiel für einen Entscheidungsbaum mit 2 Merkmalen und 3 Klassen. Der rechte Teil der Abbildung stellt die entsprechende Zerlegung des Merkmalsraums in Partitionen dar. Zusätzlich zu den Beschriftungen der Partitionen mit den Namen der vorliegenden Klasse sind die Beispiele der Klassen durch unterschiedliche Arten von "Punkten" gekennzeichnet.

Klassenzugehörigkeiten bestmöglich partitioniert. Die Implementierung des in dieser Arbeit verwendeten Entscheidungsbaumlerners basiert auf den Arbeiten von Geoffrey I. Webb [Web99] und J. Ross Quinlan [Qui86], [Qui93].

### Der ID3–Algorithmus

Die Grundlage der verwendeten Implementierung ist der von Quinlan 1986 [Qui86] erstmals publizierte *ID3–Algorithmus*. Das Verfahren ist iterativ und spannt den Baum beginnend an der Wurzel top–down anhand der Ausprägungen der Merkmale auf. Dabei zerlegt ein Merkmal $s$ mit $k$ Werten die Beispielmenge $\mathcal{X}$ aus Beispielen $(\vec{x}_i, y_i)$ in $k$ Untermengen $X_1, \ldots, X_k$. Es wird so lange in Untermengen verzweigt, bis alle Beispiele einer Untermenge zu derselben Klasse gehören.

Zur Auswahl des Merkmals, das den nächsten Knoten bildet, wird mit Hilfe der *Entropie* (der Unordnung) und der *bedingten Entropie* (der bedingten Unordnung) das Merkmal ausgewählt, das den größten Informationsgewinn liefert, das also die Unordnung der Beispiele eines Knotens bezüglich ihrer Klassenzugehörigkeiten am meisten senkt.

**Definition 4.2 (Entropie)** *Die* Entropie *in einem Knoten berechnet sich durch*

$$H(Knoten) = -\sum_{y=1}^{|\mathcal{Y}|} p(Knoten_y)\, log_2\, p(Knoten_y)$$

*mit $|\mathcal{Y}|$ der Anzahl an Klassen und $p(Knoten_y)$ der Wahrscheinlichkeit dafür, dass am betrachteten Knoten die Klasse $y$ vorliegt.*

Dabei kann $p(Knoten_y)$ berechnet werden durch

$$p(Knoten_y) = \frac{|\{(\vec{x}_{Knoten}, y)\}|}{|\{\vec{x}_{Knoten}\}|},$$

also durch die Anzahl der Beispiele $\vec{x}$ an diesem Knoten, die zur Klasse $y$ gehören, dividiert durch die gesamte Anzahl der Beispiele an diesem Knoten.

Die Entropie ist ein Maß der Unordnung. Das heißt je höher die Entropie, desto größer die Unordnung. Zur Bestimmung des nächsten Verzweigungs–Merkmals $s$ wird nun die *bedingte* Entropie benutzt. Dieses Maß beschreibt die Entropie die noch verbleibt, wenn an dem Merkmal $s$ verzweigt wird.

**Definition 4.3 (Bedingte Entropie)** *Die* bedingte Entropie *in einem Knoten berechnet sich durch*

$$H(Knoten|s) = -\sum_{i=1}^{k} p(Knoten_{s=i}) \cdot H(Knoten|s=i)$$

$$= -\sum_{i=1}^{k} \big(p(Knoten_{s=i}) \cdot \big(\sum_{y=1}^{|\mathcal{Y}|} p(Knoten_y|s=i) \cdot log_2\, p(Knoten_y|s=i)\big)\big)$$

*mit $k$ der Anzahl der Ausprägungen des Merkmals $s$, $p(Knoten_{s=i})$ der Wahrscheinlichkeit der Ausprägung $i$ des Merkmals $s$ am Knoten und $p(Knoten_y|s=i)$ der Wahrscheinlichkeit dafür, dass bei Ausprägung $s=i$ die Klasse $y$ beobachtet wird.*

Die Wahrscheinlichkeit der Ausprägung $i$ des Merkmals $s$ am Knoten lässt sich berechnen durch

$$p(Knoten_{s=i}) = \frac{|\{\vec{x}_{Knoten_{s=i}}\}|}{|\{\vec{x}_{Knoten}\}|},$$

also durch die Anzahl der Beispiele $\vec{x}$ an diesem Knoten, die für das Merkmal $s$ den Wert $i$ aufweisen dividiert durch die gesamte Anzahl der Beispiele an diesem Knoten.

Die Wahrscheinlichkeit für die Beobachtung der Klasse $y$ bei der Merkmalsausprägung $s=i$ wird berechnet durch

$$p(Knoten_y|s=i) = \frac{|\{(\vec{x}, y)_{s=i}\}|}{|\{\vec{x}_{s=i}\}|},$$

also durch die Anzahl der Beispiele $\vec{x}$ der Klasse $y$ mit $s=i$ dividiert durch die Anzahl aller Beispiele mit $s=i$.

Der *Informationsgewinn* (Information Gain) ist ein Maß für den "Zuwachs der Ordnung" durch die Verzweigung am Merkmal $s$ und wird berechnet durch

$$G(s) = H(Knoten) - H(Knoten|s). \tag{4.1}$$

Als nächstes Verzweigungsmerkmal wird demnach das Attribut mit dem größten Informationsgewinn ausgewählt.

Die Notation des Informationsgewinns tendiert zur Bevorzugung von Merkmalen mit vielen unterschiedlichen Werten. Um dieses Verhalten auszugleichen, wird der Informationsgewinn mit Hilfe der *Split–Information* normalisiert.

**Definition 4.4 (Split–Information)** *Die* Split–Information *in einem Knoten berechnet sich durch*

$$SplitInfo(s) = -\sum_{i=1}^{k} p(Knoten_{s=i}) \cdot log_2\, p(Knoten_{s=i}).$$

*mit k der Anzahl der Ausprägungen des Merkmals s und $p(Knoten_{s=i})$ der Wahrscheinlichkeit der Ausprägung i des Merkmals s am Knoten.*

Mit der Split–Information und dem Informationsgewinn aus Gleichung (4.1) kann die *Informationsrate* (Information Gain Ratio) berechnet werden durch

$$GR(s) = \frac{G(s)}{SplitInfo(s)}.$$

## Der C4.5–Algorithmus

Bei den später folgenden Experimenten wird die ebenfalls von J. Ross Quinlan stammende Weiterentwicklung des ID3–Algorithmus, der so genannte *C4.5*–Algorithmus [Qui93] benutzt. Dieser Algorithmus handhabt unbekannte Merkmalswerte bei der Konstruktion des Entscheidungsbaums. Es werden lediglich die Beispiele betrachtet, in denen der Merkmalswert definiert ist. Bei der Klassifikation mit Hilfe eines bereits bestehenden Entscheidungsbaums werden die Wahrscheinlichkeiten für die möglichen Klassen bei fehlenden Merkmalswerten geschätzt, indem alle Pfade ausgehend von dem unbekannten Merkmal überprüft werden.

Eine zusätzliche Erweiterung des ID3–Algorithmus besteht in der Handhabung kontinuierlicher Merkmalswerte, die bei dem in dieser Diplomarbeit vorliegenden Klassifikationsproblem vorhanden sind. Der C4.5–Algorithmus behandelt solche Merkmalswerte, indem die in der Beispielmenge eines Knotens vorkommenden Werte $i_1, \ldots, i_k$ eines Merkmals $s$ zunächst aufsteigend sortiert werden. Anschließend wird die Beispielmenge für jeden Wert $i_j$ mit $j = 1, \ldots k$ in die Beispiele aufgeteilt, deren Werte $\leq i_j$ sind und die Beispiele, für die $s > i_j$ gilt. Für jede dieser Partitionen wird der Informationsgewinn $G$ oder die Informationsrate $GR$ berechnet und es wird die Partition gewählt, die den maximalen Wert liefert.

**Pruning** Der so konstruierte Entscheidungsbaum kann die meisten Beispiele der Lernmenge richtig klassifizieren. Er kann jedoch auch sehr komplex werden und Pfade enthalten, die aufgrund einiger weniger Beispiele entstanden sind und bei der Klassifizierung neuer Beispiele aus einem Testdatensatz nicht sehr hilfreich sind. Daher wird auf dem Baum nach seiner Fertigstellung ein *Pruning* (Beschneidung) durchgeführt. Dabei werden Teilbäume durch Blätter ersetzt.

Zur Auswahl der zu ersetzenden Teilbäume können verschiedene Kriterien genutzt werden. Eines dieser Kriterien ist die Fehlerrate. Bleibt die Fehlerrate durch die probeweise Löschung eines Teilbaums unverändert oder sinkt, wird dieser Teilbaum aus dem Entscheidungsbaum entfernt. Die Fehlerrate wird geschätzt, indem die Anzahl der Fehler auf einem Testdatensatz durch die Anzahl der Datensätze des Testdatensatzes dividiert

wird. Ein alternatives Kriterium ist die Konfidenz eines Teilbaums, die durch die Anzahl der durch den Teilbaum richtig klassifizierten Beispiele im Verhältnis zu allen durch den Teilbaum klassifizierten Beispielen berechnet werden kann. Liegt die Konfidenz unter einem definierten Schwellwert, wird der Teilbaum gelöscht.

**Grafting** Nach der Erstellung und dem Pruning eines Entscheidungsbaums kann noch ein weiterer Nachbearbeitungsschritt zur Verbesserung der Vorhersagegenauigkeit durchgeführt werden, der sich *Grafting* nennt. Dieses Verfahren ist komplementär zum Pruning. Dem Baum werden neue Verzweigungen hinzugefügt. Für Details des Entscheidungsbaum–Graftings sei an dieser Stelle auf [Web99] verwiesen.

## 4.4. Die Stützvektormethode

Statt den Merkmalsraum in $k$ Partitionen zu zerlegen wie der in dem vorangegangenen Abschnitt 4.3 beschriebene Entscheidungsbaum, passt die Stützvektormethode (engl. Support Vector Machine, abgekürzt SVM) eine (mehrdimensionale) separierende Hyperebene in den Merkmalsraum ein. Diese Hyperebene soll die gegebenen Beispiele gemäß ihrer Klassenzugehörigkeiten bestmöglich voneinander trennen (siehe [Vap95], [CV95], [Bur98]). Dazu muss jedes Beispiel wieder durch einen Vektor in einem Vektorraum repräsentiert werden. Die Anzahl der Merkmale eines Klassifikationsproblems definiert die Dimension der Vektoren.

Die Beispiele, die der Hyperebene am nächsten liegen bestimmen deren Lage und werden als *Stützvektoren* bezeichnet. Das Ziel der SVM besteht darin, den Abstands der Stützvektoren zu der trennenden Hyperebene zu maximieren (*Maximum Margin Methode*). Der dadurch entstehende, breite, leere Rand (der *Margin*) soll später dafür sorgen, dass auch Objekte, die nicht genau den Trainingsobjekten entsprechen, möglichst zuverlässig klassifiziert werden. Die Maximierung des Margins wird als Optimierungsproblem betrachtet.

Bei binären Klassenzugehörigkeiten der Beispiele bedeutet dies formal, dass ein Klassifizierungsproblem $\mathcal{Y} = \{-1; +1\}$ und eine Menge von Trainingsbeispielen $(\vec{x}_i, y_i) \in \mathcal{X}$ mit $\mathcal{X} \in \mathbb{R}^m$ und $y_i \in \mathcal{Y}$ gegeben ist. Weiter gilt $\mathcal{X} = C_+ \cup C_-$ mit

$$C_+ = \{\vec{x}_i |\ y_i = +1\} \text{ und } C_- = \{\vec{x}_i |\ y_i = -1\}.$$

Gesucht wird eine Hyperebene $H$ mit

$$H = \{\vec{x}_i |\ \langle \vec{w}, \vec{x}_i \rangle + b = 0\}, \tag{4.2}$$

die die Mengen $C_+$ und $C_-$ bestmöglich trennt. Dabei bezeichnet $\langle \cdot, \cdot \rangle$ ein Skalarprodukt, $\vec{w}$ den Normalenvektor und $b$ die Verschiebung. Abbildung 4.3 zeigt beispielhaft eine linear trennende Hyperebene $H$ im zweidimensionalen Fall.

Bei gegebener Hyperebene $H$ erfolgt die Klassifikation eines Beispiels durch die Vorhersage der Lage des Punktes bezüglich der Hyperebene. Die Hypothese lautet demnach

$$\hat{y} = sign\left(\langle \vec{w}, \vec{x}_i \rangle + b\right). \tag{4.3}$$

Dabei ist das Vorzeichen ($sign$) eines Punktes in Richtung des Normalenvektors, der senkrecht auf der trennenden Hyperebene steht positiv und das eines Punktes auf der anderen Seite der Hyperebene negativ. $\hat{y}$ bezeichnet somit die vorhergesagte Klasse.

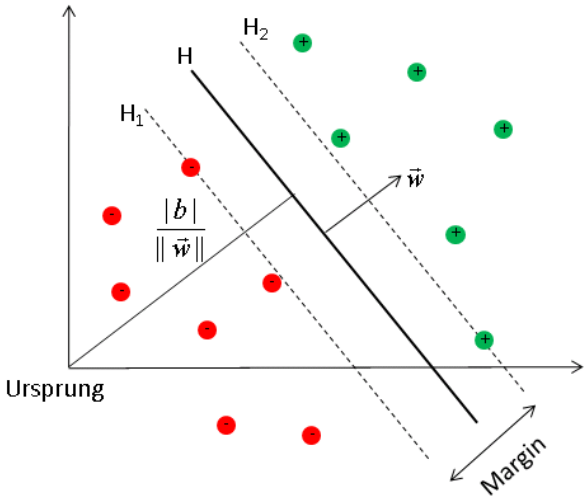



Abbildung 4.3.: Linear trennende Hyperebene der SVM bei einem binären Klassifikationsproblems im zweidimensionalen Fall.

### 4.4.1. Die optimale Hyperebene

Eine separierende Hyperebene heißt *optimal*, wenn der *Margin*, also der Abstand zum nächsten positiven und zum nächsten negativen Beispiel, maximal ist. Zwischen den zur optimalen Hyperebene parallelen Hyperebenen $H_1$ und $H_2$ (siehe Abbildung 4.3), die den Margin bestimmen, liegen also keine Beispiele.

Zur Maximierung des Margins werden $\vec{w}$ und $b$ zunächst relativ zu den Trainingsbeispielen $\mathcal{X}$ skaliert, so dass

$$\min_{\vec{x}_i \in \mathcal{X}} \langle \vec{w}, \vec{x}_i \rangle + b = 1$$

gilt. Der Abstand der Hyperebene $H$ zum Ursprung beträgt dann $\frac{|b|}{\|\vec{w}\|}$. Der Abstand von $H_1$ zum Ursprung ist $\frac{b-1}{\|\vec{w}\|}$, der Abstand von $H_2$ zum Ursprung ist $\frac{b+1}{\|\vec{w}\|}$, so dass diese beiden Hyperebenen einen Abstand von $\frac{1}{\|\vec{w}\|}$ zur separierenden Hyperebene $H$ haben. Der Margin beträgt folglich $\frac{2}{\|\vec{w}\|}$.

Um nun die geometrische Breite $\frac{1}{\|\vec{w}\|}$ zu maximieren, muss die Länge von $\|\vec{w}\|$ *minimiert* werden, was gleichbedeutend mit der Minimierung von $\|\vec{w}\|^2$ ist. Die Bedingung, dass keine Beispiele zwischen den Hyperebenen $H_1$ und $H_2$ liegen, lässt sich durch die Nebenbedingungen $\forall \vec{x}_i : y_i \left( \langle \vec{w}, \vec{x}_i \rangle + b \right) \geq 1$ ausdrücken.

Das Optimierungsproblem besteht also in der *Minimierung* der Funktion $\frac{1}{2}\|\vec{w}\|^2$ und wird dementsprechend formuliert als

$$\min \frac{1}{2}\|\vec{w}\|^2 \tag{4.4}$$

unter den Nebenbedingungen

$$y_i \left( \langle \vec{w}, \vec{x}_i \rangle + b \right) \geq 1 \; \forall \, \vec{x}_i \in \mathcal{X}. \tag{4.5}$$

Die so erhaltene Optimierungsfunktion kann für $|\mathcal{X}| = n$ mit Hilfe der Lagrange–Methode zusammengefasst werden zu

$$L(\vec{w}, b, \vec{\alpha}) = \frac{1}{2}\|\vec{w}\|^2 - \sum_{i=1}^{n} \alpha_i \left(y_i \left(\langle \vec{w}, \vec{x}_i \rangle + b\right) - 1\right) \tag{4.6}$$

unter den Nebenbedingungen

$$\vec{w} = \sum_{i=1}^{n} \alpha_i y_i \vec{x}_i \quad \text{und} \quad \sum_{i=1}^{n} \alpha_i y_i = 0. \tag{4.7}$$

mit $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n)$ und $\alpha_i \geq 0$ dem Lagrange–Multiplikator. Die Funktion (4.6) ist bezüglich $\vec{\alpha}$ zu maximieren und bezüglich $\vec{w}$ und $b$ zu minimieren. Durch das Nullsetzen der Ableitungen

$$\frac{\partial}{\partial \vec{w}} L(\vec{w}, b, \vec{\alpha}) = 0 \quad \text{und} \quad \frac{\partial}{\partial b} L(\vec{w}, b, \vec{\alpha}) = 0$$

erhält man

$$\sum_{i=1}^{n} \alpha_i y_i = 0 \quad \text{und} \quad \vec{w} = \sum_{i=1}^{n} \alpha_i y_i \vec{x}_i.$$

Durch Einsetzten des Ergebnisses in $L(\vec{w}, b, \vec{\alpha})$ und Umformung ergibt sich das *duale Problem* der *Maximierung* von

$$L(\vec{\alpha}) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j \alpha_i \alpha_j \langle \vec{x}_i, \vec{x}_j \rangle \tag{4.8}$$

unter den Nebenbedingungen

$$\alpha_i \geq 0 \,\forall\, i = 1, \ldots n \quad \text{und} \quad \sum_{i=1}^{n} \alpha_i y_i = 0. \tag{4.9}$$

Die Lösung $\vec{\alpha}^*$ dieses Problems enthält ein $\alpha_i$ für jedes Beispiel $\vec{x}_i$ mit

$\alpha_i = 0$, falls $\vec{x}_i$ in dem richtigen Halbraum liegt und
$\alpha_i > 0$, falls $\vec{x}_i$ auf den Hyperebenen $H_1$ oder $H_2$ liegt.

Die Beispiele $\vec{x}_i$ mit $\alpha_i > 0$ sind die *Stützvektoren*. Mit der Lösung $\vec{\alpha}^*$ berechnet sich der Normalenvektor $\vec{w}$ durch

$$\vec{w} = \sum \alpha_i y_i \vec{x}_i.$$

Der optimale Normalenvektor besteht folglich aus einer Linearkombination von Stützvektoren. Diese Stützvektoren liegen der Hyperebene am nächsten und bestimmen deren Lage, denn für alle anderen Vektoren gilt $\alpha_i = 0$ und sie haben somit keinen Einfluss auf den Normalenvektor und die Hyperebene.

Die Optimierung des finalen Problems, also die Optimierung der $\alpha$'s, kann mit Hilfe verschiedener Methoden erfolgen. Es können numerische Verfahren, evolutionäre Algorithmen [Mie06] oder die sequentielle Optimierung (Sequential Minimal Optimization, SMO) [Pla98] eingesetzt werden.

### Relaxierung des Optimierungsproblems

Für den in der Praxis häufig auftretenden Fall nicht komplett linear trennbarer Daten wird das Problem relaxiert. Dazu werden auch Beispiele erlaubt, die innerhalb des Margins oder auf der falschen Seite der Hyperebene liegen. Diese Beispiele werden allerdings mit einem Strafterm $\xi_i > 0$ versehen. Bei korrekt klassifizierten Beispielen wird $\xi_i = 0$ gesetzt. Die entsprechenden Anpassungen der Gleichungen (4.4) und (4.5) führen zu dem Optimierungsproblem

$$\min \frac{1}{2}\|\vec{w}\|^2 + C\sum_{i=1}^{n}\xi_i \text{ für ein festes } C \in \mathbb{R}_{>0} \tag{4.10}$$

unter den Nebenbedingungen

$$y_i\left(\langle\vec{w}, \vec{x}_i\rangle + b\right) \geq 1 - \xi_i \; \forall\, i = 1, \ldots n. \tag{4.11}$$

Daraus folgt

$$0 \leq \alpha_i \leq C.$$

$C$ bestimmt somit den Einfluss von Trainingsfehlern. Die möglichen Lagen von Beispielen bezüglich der Hyperebene und die entsprechenden Parameterwerte $\xi$ und $\alpha$ sind für ein binäres Klassifikationsproblem im zweidimensionalen Fall in Abbildung 4.4 dargestellt.

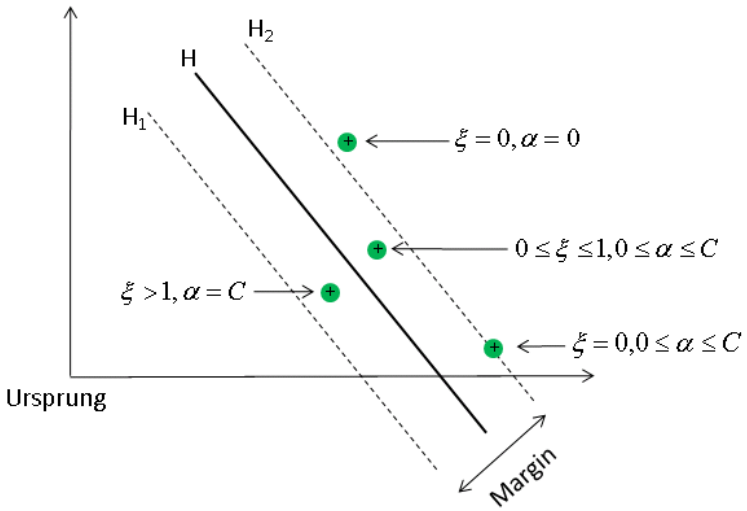



Abbildung 4.4.: Mögliche Lagen von Beispielen bezüglich der Hyperebene und die entsprechenden Werte für $\xi$ und $\alpha$ bei Relaxierung des SVM–Optimierungsproblems für ein binäres Klassifikationsproblem im zweidimensionalen Fall.

## 4.4.2. Kernfunktionen

Bei nicht linear trennbaren Daten kann eine Transformation der Daten in einen anderen (meist höherdimensionalen) Merkmalsraum eine lineare Trennung dennoch ermöglichen.

Dazu werden die Beispielvektoren zunächst mit einer Funktion

$$\begin{aligned}\Phi : \mathbb{R}^m &\to \mathcal{M} \\ \vec{x} &\mapsto \Phi(\vec{x})\end{aligned}$$

in einen *Hilbertraum* $\mathcal{M}$ abgebildet, in dem ein Skalarprodukt definiert ist.

Zur Klassifikation muss die Hyperebene also nun in $\mathcal{M}$ konstruiert werden. Dazu ist ein Skalarprodukt $\langle\Phi(\vec{x}_i), \Phi(\vec{x}_j)\rangle_{\mathcal{M}}$ zu berechnen. Die Transformation der Daten und die Berechnung des Skalarprodukts kann aufgrund hoher Dimensionalität des Merkmalsraums $\mathcal{M}$ sehr schwierig werden. Daher werden diese Schritte durch eine so genannte Kernfunktion $k$ ersetzt, die im $\mathbb{R}^m$ lebt, sich aber wie ein Skalarprodukt in $\mathcal{M}$ verhält. Dabei ist die Kenntnis der Abbildungsfunktion $\Phi$ nicht nötig. Man braucht lediglich die Kernfunktion als Ähnlichkeitsmaß.

**Definition 4.5 (Kernfunktion)** *Eine Funktion $k : \mathbb{R}^m \times \mathbb{R}^m \to \mathbb{R}$, die ein Skalarprodukt zwischen zwei Punkten in einem hochdimensionalen Raum berechnet, also*

$$k(\vec{x}_i, \vec{x}_j) = \langle \Phi(\vec{x}_i), \Phi(\vec{x}_j) \rangle_{\mathcal{M}}$$

*und sich für alle $\vec{x}$ im $\mathbb{R}^m$ als positiv (semi–)definite Gram–Matrix mit symmetrischer Funktion $k$ darstellen lässt, heißt* Kernfunktion.

Anders ausgedrückt können nur solche Funktionen als Kernfunktionen genutzt werden, die die *Mercer Bedingung* erfüllen. Für Details zur Mercer Bedingung und der entsprechenden Herleitung sei an dieser Stelle auf [CV95], [Vap95] und [Bur98] verwiesen.

Häufig genutzte Kernfunktionen sind die Folgenden:

#### Lineare Kernfunktion

$$k(\vec{x}_i, \vec{x}_j) = \langle \vec{x}_i, \vec{x}_j \rangle \tag{4.12}$$

#### Polynomielle Kernfunktion

$$k(\vec{x}_i, \vec{x}_j) = (\gamma \langle \vec{x}_i, \vec{x}_j \rangle + r)^d, \ \gamma > 0 \tag{4.13}$$

#### Sigmoid–Kernfunktion

$$k(\vec{x}_i, \vec{x}_j) = \tanh(\gamma \langle \vec{x}_i, \vec{x}_j \rangle + r) \tag{4.14}$$

#### Radial–Basisfunktion (RBF)

$$k(\vec{x}_i, \vec{x}_j) = \exp\left(-\gamma ||\vec{x}_i - \vec{x}_j||^2\right), \ \gamma > 0 \tag{4.15}$$

Dabei bezeichnen die Parameter $\gamma$, $r$ und $d$ die Kernparameter (aus [HCL09]).

Die in Kapitel 5 verwendete RBF–Funktion ist eine reellwertige Funktion, deren Wert allein von der Distanz zweier Vektoren abhängt. Es gilt also $\Phi(\vec{x}_i, \vec{x}_j) = \Phi(||\vec{x}_i - \vec{x}_j||)$. Dabei wird als Distanzmaß häufig die *euklidische Distanz* gewählt, die bereits zu Beginn des vorliegenden Kapitels (Definition 4.1) definiert wurde. Andere Distanzmaße sind jedoch ebenfalls möglich.

### Eine Kernfunktion für Baumstrukturen

Im Bereich der Textanalyse werden häufig Kernfunktionen für Bäume verwendet, da die Darstellung von Grammatiken in der Form von Bäumen für viele Lernverfahren nützliche Informationen enthält. Da die Systemcall–Logdateien in dieser Diplomarbeit ebenfalls als Texte betrachtet werden und aufgrund des vorliegenden XML–Formats bereits eine inhärente Baumstruktur aufweisen (siehe Anhang B, Abbildung B.2), wird eine solche Kernfunktion im Folgenden vorgestellt.

Collins und Duffy präsentieren in [CD01] die Idee der Berücksichtigung struktureller Informationen in einer Kernfunktion durch die Einbeziehung aller Teilbäume. Dazu sei $m$ die Anzahl aller in den vorliegenden Trainingsdaten enthaltenen Teilbäume. Jeder Baum $T$ wird als $m$–dimensionaler Vektor $\vec{h}(T)$ dargestellt, in dem die $i$–te Komponente $h_i(T)$ die Häufigkeit des $i$–ten Teilbaums in $T$ angibt. $N_T$ bezeichnet die Menge aller Knoten in $T$. $I_i(n)$ sei eine Indikatorfunktion mit

$$I_i(n) = \begin{cases} 1 & \text{wenn der Teilbaum } i \text{ in dem Knoten } n \text{ verwurzelt ist} \\ 0 & \text{sonst.} \end{cases}$$

Für zwei Bäume $T_1$ und $T_2$ wird der Kern definiert durch

$$\begin{aligned} k(T_1, T_2) &= \langle \vec{h}(T_1), \vec{h}(T_2) \rangle \\ &= \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} \sum_{i=1}^{m} I_i(n_1) I_i(n_2) \\ &= \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} C(n_1, n_2) \end{aligned} \tag{4.16}$$

mit $C(n_1, n_2) = \sum_i I_i(n_1) I_i(n_2)$. Bei der Bezeichnung der Produktionsregeln des Knoten $n$ mit $Prod(n)$ ist C wie folgt definiert:

$$C(n_1, n_2) = \begin{cases} 0 & \text{wenn } Prod(n_1) \neq Prod(n_2), \\ \lambda & \text{wenn } Prod(n_1) = Prod(n_2) \text{ und} \\ & n_1 \text{ und } n_2 \text{ Präterminalknoten}, \\ C^*(n_1, n_2) & \text{wenn } Prod(n_1) = Prod(n_2) \text{ und} \\ & n_1 \text{ und } n_2 \text{ keine Präterminalknoten.} \end{cases} \tag{4.17}$$

*Präterminalknoten* sind Knoten, deren Kinder Blattknoten sind. Die Funktion $C^*(n_1, n_2)$ wird rekursiv definiert durch

$$C^*(n_1, n_2) = \lambda \prod_{j=1}^{nc(n_1)} (1 + C(ch(n_1, j), ch(n_2, j))) \tag{4.18}$$

mit $nc(n_1)$ der Anzahl der Kinder des Knotens $n_1$ und $ch(n, j)$ dem $j$–ten Kindsknoten des Knoten $n$. Da $Prod(n_1) = Prod(n_2)$ gilt, gilt auch $nc(n_1) = nc(n_2)$. Der Parameter $0 < \lambda < 1$ bewirkt eine exponentielle Dämpfung großer Teilbäume.

Eine *kontext–sensitive* Erweiterung dieses Kerns wurde von Zhou et. al. in [ZZJZ07] vorgestellt. In diesem Kern wird also auch die *Position* des Teilbaums in dem Gesamtbaum berücksichtigt. Die Kernfunktion wird definiert als

$$k_C(T_1, T_2) = \sum_{i=1}^{m} \sum_{n_1^i[1] \in N_1^i[1], n_1^i[2] \in N_1^i[2]} C(n_1^i[1], n_1^i[2]). \tag{4.19}$$

$N_1^i[j]$ ist hier *nicht* die Menge aller Knoten wie in Gleichung (4.16), sondern bezeichnet die Menge der Wurzelknotenpfade im Baum $T_j$ mit einer Länge von $i$. Die maximale Länge der zu betrachtenden Wurzelknotenpfade wird durch $m$ festgelegt. Auch $m$ bezeichnet hier also nicht wie in Gleichung (4.16) die Anzahl der Teilbäume. Ein Wurzelknotenpfad der Länge $i$ im Baum $T_j$ wird mit $n_1^i[j] = (n_1, n_2, \ldots, n_i)[j]$ bezeichnet. Ein Wurzelknotenpfad ist ein Pfad im Gesamtbaum, der zu der Wurzel des betrachteten Teilbaums führt. Es werden also die $i-1$ Vorgängerknoten in $T_j$ berücksichtigt. $n_{k+1}[j]$ ist der Vorgängerknoten von $n_k[j]$ und $n_1[j]$ ist der Wurzelknoten des kontextfreien Teilbaums.

### 4.4.3. SVM für Mehrklassenprobleme

Das in 4.4.1 vorgestellte Verfahren dient der Lösung eines Zweiklassenproblems. Das in dieser Diplomarbeit vorliegende Klassifikationsproblem ist allerdings ein Mehrklassenproblem. Ein Klassifikator für ein Mehrklassenproblem realisiert eine Funktion $H : \mathcal{X} \to \mathcal{Y}$, die eine Instanz $\vec{x}_i \in \mathbb{R}^m$ auf ein Element $y \in \mathcal{Y} = \{1, \ldots, c\}$ abbildet. Ein Mehrklassenproblem kann durch die SVM für Zweiklassenprobleme mittels eines one–against–all Schemas gelöst werden, bei dem eine Hyperebene pro Klasse berechnet wird, die die Beispiele dieser Klasse bestmöglich von allen Beispielen der übrigen Klassen trennt.

Koby Crammer und Yoram Singer [CS01] haben 2001 einen SVM–Algorithmus vorgestellt, der *direkt* auf Mehrklassenprobleme anwendbar ist. Der Klassifikator aus Gleichung (4.3) lautet dabei

$$\hat{y} = H_{\mathbf{M}}(\vec{x}_i) = \arg\max_{r=1}^{c} \langle M_r, \vec{x}_i \rangle. \tag{4.20}$$

Hier bezeichnet $\mathbf{M}$ eine Matrix der Größe $c \times m$ über $\mathbb{R}$ und $M_r$ ist die $r$–te Zeile von $\mathbf{M}$. Ein Skalarprodukt der $r$–ten Zeile mit $\vec{x}_i$ ergibt den Ähnlichkeitswert für das Beispiel $\vec{x}_i$ und die Klasse $r$. Es wird folglich die Klasse $\hat{y}$ vorhergesagt, deren Zeile den größten Ähnlichkeitswert bezüglich $\vec{x}_i$ erreicht. Ein solches Vorgehen wird auch als *WTA* (Winner Takes All) bezeichnet.

Der empirische Fehler für ein Mehrklassenproblem mit $|\mathcal{X}| = n$ Beispielen $(\vec{x}_i, y_i)$ und $y_i$ der Klasse des Beispiels $\vec{x}_i$ wird definiert als

$$\epsilon_S(\mathbf{M}) = \frac{1}{n} \sum_{i=1}^{n} [\![ H_{\mathbf{M}}(\vec{x}_i) \neq y_i ]\!]$$

mit

$$[\![ H_{\mathbf{M}}(\vec{x}_i) \neq y_i ]\!] = \begin{cases} 1 & \text{wenn } H_{\mathbf{M}}(\vec{x}_i) \neq y_i \\ 0 & \text{sonst.} \end{cases}$$

Crammer und Singer ersetzen den Klassifikationsfehler $[\![H_{\mathbf{M}}(\vec{x}_i) \neq y_i]\!]$ durch eine lineare Grenze

$$\max_r\{\langle M_r, \vec{x}_i\rangle + 1 - \delta_{y_i,r}\} - \langle M_{y_i}, \vec{x}_i\rangle$$

mit

$$\delta_{y_i,r} = \begin{cases} 1 \text{ wenn } y_i = r \\ 0 \text{ sonst.} \end{cases}$$

Der Fehler ist also linear proportional zur Differenz zwischen dem Ähnlichkeitswert der korrekten Klasse $y_i$ und dem Maximum der Ähnlichkeitswerte der übrigen Klassen.

Zur Formulierung des Optimierungsproblems wird die $l_2$–Norm einer Matrix $\mathbf{M}$ definiert als die $l_2$–Norm des Vektors, der durch Konkatenation der Zeilen von $\mathbf{M}$ entsteht, also durch

$$\|\mathbf{M}\|_2 = \|(M_1, \ldots, M_c)\|_2 = \sqrt{\sum_{i,j} M_{i,j}^2}.$$

Das Optimierungsproblem aus Gleichung (4.4) ergibt sich somit als

$$\min_{\mathbf{M}} \frac{1}{2}\|\mathbf{M}\|_2^2 \tag{4.21}$$

unter den Nebenbedingungen

$$\forall i, r: \ \langle M_{y_i}, \vec{x}_i\rangle + \delta_{y_i,r} - \langle M_r, \vec{x}_i\rangle \geq 1. \tag{4.22}$$

Das relaxierte Problem mit einem Strafterm ähnlich der Gleichung (4.10) ergibt sich dementsprechend hier als

$$\min_{\mathbf{M},\xi} \frac{1}{2}\|\mathbf{M}\|_2^2 + \sum_{i=1}^{n} \xi_i \tag{4.23}$$

unter den Nebenbedingungen

$$\forall i, r: \ \langle M_{y_i}, \vec{x}_i\rangle + \delta_{y_i,r} - \langle M_r, \vec{x}_i\rangle \geq 1 - \xi_i. \tag{4.24}$$

Auch bei dieser Formulierung der SVM für Mehrklassenprobleme hängt das duale Problem vom Skalarprodukt ab, so dass wieder Kernfunktionen benutzt werden können. Die Optimierung wird aufgrund der $nc$ vielen einzusetzenden Variablen zur Steigerung der Effizienz in kleinere Probleme zerlegt. Für weitere Details zu der Herleitung des dualen Problems und der Optimierung sei an dieser Stelle auf [CS01] verwiesen.

Der präsentierte Ansatz führt im Vergleich zu der Lösung eines Mehrklassenproblems mit der SVM durch ein one–against–all Schema zu besseren Laufzeiten bei mindestens ebenso hoher Genauigkeit.

### SVM für Mehrklassenprobleme mit strukturierten Ausgabemengen

In [TJH+05] wird der Ansatz von Crammer und Singer für den Fall strukturierter Ausgabemengen wie Sequenzen, Strings, Bäume, Netze oder Graphen generalisiert. Es wird eine Diskriminanzfunktion $F : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}$ über Eingabe–/Ausgabe–Paaren $(\vec{x}_i, \vec{y}_i)$ gelernt.

Durch die Maximierung von $F$ über die Ausgabevariablen für eine spezielle Eingabe $\vec{x}_i$ kann eine Vorhersage abgeleitet werden. Die allgemeine Formulierung der Hypothese $f$, ähnlich den Gleichungen (4.3) und (4.20), lautet

$$f(\vec{x}_i; \vec{w}) = \arg\max_{\vec{y} \in \mathcal{Y}} F(\vec{x}_i, \vec{y}; \vec{w}), \tag{4.25}$$

wobei $\vec{w}$ einen Parametervektor beschreibt. $F$ kann als Kompatiblitätsfunktion betrachtet werden. Diese misst, wie kompatibel Paare $(\vec{x}_i, \vec{y})$ sind. $F$ wird als linear bezüglich einer kombinierten Merkmalsrepräsentation $\Psi(\vec{x}, \vec{y})$ von Eingaben und Ausgaben angenommen, das heißt

$$F(\vec{x}_i, \vec{y}; \vec{w}) = \langle \vec{w}, \Psi(\vec{x}_i, \vec{y}) \rangle. \tag{4.26}$$

Die spezielle Form von $\Psi$ ist problemabhängig und wird später in diesem Abschnitt beispielhaft für die WTA–Regel aus Gleichung (4.20) angegeben.

Im linear separierbaren Fall lautet das entsprechend angepasste Optimierungsproblem aus den Gleichungen (4.4) und (4.21)

$$\min_{\vec{w}} \frac{1}{2} \|\vec{w}\|^2 \tag{4.27}$$

unter den Nebenbedingungen

$$\forall i, \forall \vec{y} \in \mathcal{Y} \setminus \vec{y}_i : \langle \vec{w}, \delta\Psi_i(\vec{y}) \rangle \geq 1 \tag{4.28}$$

mit $\delta\Psi_i(\vec{y}) \equiv \Psi(\vec{x}_i, \vec{y}_i) - \Psi(\vec{x}_i, \vec{y})$.

Das bereits aus den Gleichungen (4.10) und (4.23) bekannte relaxierte Problem für den nicht linear trennbaren Fall lautet hier

$$\min_{\vec{w}, \xi} \frac{1}{2} \|\vec{w}\|^2 + \frac{C}{n} \sum_{i=1}^{n} \xi_i \tag{4.29}$$

unter den Nebenbedingungen

$$\forall i, \forall \vec{y} \in \mathcal{Y} \setminus \vec{y}_i : \langle \vec{w}, \delta\Psi_i(\vec{y}) \rangle \geq 1 - \xi_i, \ \xi_i \geq 0. \tag{4.30}$$

Das Lernen mit strukturierten Ausgaberäumen erfordert in den meisten Fällen andere Fehlerfunktionen als die bei binärer Klassifikation verwendeten. Daher wird angenommen, dass eine beschränkte Fehlerfunktion $\Delta : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ zur Verfügung steht, so dass $\Delta(\vec{y}_i, \vec{y})$ den Fehler angibt, der mit einer Vorhersage von $\vec{y}$ bei tatsächlich vorliegender Klasse $\vec{y}_i$ verbunden ist. Eine mögliche Generalisierung des Optimierungsproblems für beliebige Fehlerfunktionen besteht in der Neu–Skalierung der Strafterme. Das führt zu dem Optimierungsproblem aus Gleichung (4.29), in dem die Nebenbedingungen angepasst werden und nun wie folgt lauten:

$$\forall i, \forall \vec{y} \in \mathcal{Y} \setminus \vec{y}_i : \langle \vec{w}, \delta\Psi_i(\vec{y}) \rangle \geq 1 - \frac{\xi_i}{\Delta(\vec{y}_i, \vec{y})}, \ \ \xi_i \geq 0 \tag{4.31}$$

Das duale Problem der *Maximierung* wird hergeleitet aus dem Optimierungsproblem in den Gleichungen (4.27) und (4.28). Folglich lautet das in Gleichung (4.8) für den Zweiklassenfall vorgestellte Problem hier

$$\max_{\alpha} \sum_{i,\vec{y}\neq\vec{y}_i} \alpha_{i\vec{y}} - \frac{1}{2} \sum_{\substack{i,\vec{y}\neq\vec{y}_i \\ j,\vec{y}'\neq\vec{y}_j}} \alpha_{i\vec{y}}\alpha_{j\vec{y}'}\langle\delta\Psi_i(\vec{y}), \delta\Psi_j(\vec{y}')\rangle \tag{4.32}$$

unter den Nebenbedingungen

$$\forall i, \forall\vec{y} \neq \mathcal{Y} \setminus \vec{y}_i : \quad \alpha_{i\vec{y}} \geq 0. \tag{4.33}$$

Da es sich bei den $y_i$,$y_j$ aus Gleichung (4.8) nun um Strukturen handelt, muss für jedes der $n$ Beispiele $\vec{x}_i$ mit der Ausgabe $\vec{y}_i$ festgestellt werden, wie groß der Abstand zu allen anderen $\Psi(\vec{x}_i, \vec{y})$ ist. Das bedeutet, dass nun statt *einem* $\alpha$–Wert pro $\vec{x}_i \in \mathcal{X}$ ein $\alpha$ für jedes Paar in $\mathcal{X} \times \mathcal{Y}$ optimiert werden muss. $\alpha_{i\vec{y}}$ bezeichnet also den Lagrange–Multiplikator, der die Nebenbedingungen für die Klasse $\vec{y} \neq \vec{y}_i$ und das Beispiel $(\vec{x}_i, \vec{y}_i)$ beschreibt. Für weitere Details der Herleitungen sei an dieser Stelle auf [TJH+05] verwiesen.

**Lösung des Optimierungsproblems** Da es sich bei dem Optimierungsproblem aus Gleichung (4.32) um $n|\mathcal{Y}|$ Nebenbedingungen handelt, ist eine Optimierung mit quadratischer Programmierung nicht mehr möglich. Tsochantaridis et. al. [TJH+05] präsentieren daher einen Algorithmus zur Lösung des Optimierungsproblems, der die spezielle Struktur des Maximum–Margin–Problems nutzt. Es wird eine Teilmenge der Nebenbedingungen gesucht, durch die alle anderen Nebenbedingungen mit einer Ungenauigkeit von höchstens $\epsilon$ ebenfalls erfüllt sind. Dazu wird eine Kostenfunktion $H(\vec{y})$ definiert und ein zu Beginn leeres Working Set $S_i$ für jedes Beispiel geführt. $S_i$ enthält später die Nebenbedingungen, die die Relaxierung des Problems definieren. Für jedes Beispiel $(\vec{x}_i, \vec{y}_i)$ wird die "am stärksten" verletzte Nebenbedingung bezüglich eines Ausgabewertes $\hat{y}$ gesucht. Übersteigt die Verletzung den aktuellen Wert von $\xi_i$ um mehr als $\epsilon$, wird die entsprechende Nebenbedingung dem Working Set hinzugefügt. So wird das Problem zunehmend stärker beschränkt. Die $\alpha$'s werden bezüglich aller Working Sets gemeinsam optimiert oder alternativ zunächst für ein $S_i$, so dass die Optimierung über alle Working Sets wesentlich seltener durchgeführt wird. Der Algorithmus terminiert, wenn kein Working Set $S_i$ mehr verändert wurde. Weitere Details zu dieser Optimierungsprozedur sind [TJH+05] zu entnehmen.

**WTA als Spezialfall** Das von Crammer und Singer [CS01] vorgestellte Winner–Takes–All–Prinzip (WTA) für das Mehrklassenproblem kann als Spezialfall des eben vorgestellten Rahmenkonstrukts aus [TJH+05] ausgedrückt werden. Dazu sei $\mathcal{Y} = \{\vec{y}_1, \ldots, \vec{y}_c\}$ und $\vec{w} = (\vec{v}_1', \ldots, \vec{v}_c')'$ mit $\vec{v}_r$ dem Gewichtsvektor der $r$–ten Klasse $\vec{y}_r$. Dann ist die WTA–Regel gemäß den Gleichungen (4.25) und (4.26) gegeben durch

$$\begin{aligned} f(\vec{x}_i; \vec{w}) &= \arg\max_{\vec{y}_r \in \mathcal{Y}} F(\vec{x}_i, \vec{y}_r; \vec{w}), \\ \text{mit } F(\vec{x}_i, \vec{y}_r; \vec{w}) &= \langle \vec{v}_r, \Phi(\vec{x}_i)\rangle. \end{aligned} \tag{4.34}$$

Dabei bezeichnet $\Phi(\vec{x}_i) \in \mathbb{R}^m$ eine beliebige Eingaberepräsentation.

Die WTA–Regel lässt sich durch Nutzung einer kombinierten Merkmalsrepräsentation darstellen. Dazu wird zunächst die orthogonale (binäre) Darstellung der $c$ Klassen $\vec{y}_r \in \mathcal{Y}$ durch Einheitsvektoren definiert als

$$\Lambda^b(\vec{y}) \equiv (\delta(\vec{y}_1, \vec{y}), \delta(\vec{y}_2, \vec{y}), \ldots, \delta(\vec{y}_c, \vec{y}))' \in \{0,1\}^c,$$

so dass $\langle \Lambda^b(\vec{y}_i), \Lambda^b(\vec{y}) \rangle = \delta(\vec{y}_i, \vec{y})$ gilt. Zur Kombination der Merkmalsabbildungen über $\mathcal{X}$ und $\mathcal{Y}$ wird das Tensorprodukt verwendet, das wie folgt definiert ist:

$$\otimes : \mathbb{R}^m \times \mathbb{R}^c \to \mathbb{R}^{m \cdot c},\ (a \otimes b)_{i+(j-1)m} \equiv a_i \cdot b_j.$$

Dabei notiert $a$ die Basis von $\mathbb{R}^m$ ($\mathcal{X}$) und $b$ die Basis von $\mathbb{R}^c$ ($\mathcal{Y}$). Die kombinierte Merkmalsabbildung im Fall des Mehrklassenproblems lässt sich somit darstellen als

$$\Psi(\vec{x}_i, \vec{y}) \equiv \Phi(\vec{x}_i) \otimes \Lambda^b(\vec{y}).$$

Mit Hilfe dieses Ansatzes lassen sich nun Mehrklassenprobleme zu noch besseren Laufzeiten lösen. Während bei der Berechnung nach einem one–against–all Schema zur Klassifikation eines Beispiels bei $c$ Klassen $c$ Evaluationen stattfinden, muss hier lediglich eine Evaluation durchgeführt werden.

## 4.5. Bayes–Klassifikation

Bayes–Klassifikatoren sind wahrscheinlichkeitstheoretische Lernverfahren. Sie ordnen ein Objekt derjenigen Klasse zu, zu der es mit der höchsten Wahrscheinlichkeit gehört und basieren auf *bedingten Wahrscheinlichkeiten* und dem *Bayes–Theorem.*

**Definition 4.6 (Bedingte Wahrscheinlichkeit)** *Die* bedingte Wahrscheinlichkeit "A gegeben B" *zweier Ereignisse A und B wird definiert als*

$$p(A|B) = \frac{p(A,B)}{p(B)}.$$

*Dabei bezeichnet $p(A,B)$ die Wahrscheinlichkeit dafür, dass A und B gemeinsam auftreten. $p(B)$ bezeichnet die a priori Wahrscheinlichkeit für das Ereignis B.*

Die bedingte Wahrscheinlichkeit beschreibt somit die Wahrscheinlichkeit für das Ereignis A wenn das Ereignis B gegeben ist. Das *Bayes–Theorem* kann direkt aus der Definition 4.6 hergeleitet werden und lautet wie folgt:

$$p(A|B) = \frac{p(B|A) \cdot p(A)}{p(B)} \tag{4.35}$$

Bezogen auf die Klassifikation kann die *a posteriori* Wahrscheinlichkeit für eine Klasse $y_r \in \mathcal{Y}$, mit $|\mathcal{Y}| = c$ der Anzahl Klassen, bei dem Vorliegen eines Beispiels $\vec{x}_i$ nach Gleichung (4.35) angegeben werden als

$$p(y_r|\vec{x}_i) = \frac{p(\vec{x}_i|y_r) \cdot p(y_r)}{p(\vec{x}_i)} \quad \text{mit} \quad p(\vec{x}_i) = \sum_{r=1}^{c} p(\vec{x}_i|y_r) \cdot p(y_r). \tag{4.36}$$

Bei durch die Trainingsbeispiele gegebenen *a priori* Wahrscheinlichkeiten $p(y_r)$ für die Klassen und $p(\vec{x}_i)$ für das Beispiel und der klassenbedingten Wahrscheinlichkeitsdichte $p(\vec{x}_i|y_r)$ für das Beispiel $\vec{x}_i$ und jede Klasse $y_r$ kann die a posteriori Wahrscheinlichkeit berechnet werden.

Die Hypothese lautet dann

$$f(\vec{x}_i) = \hat{y}(\vec{x}_i) = \arg\max_{y_r \in \mathcal{Y}} p(y_r|\vec{x}_i), \tag{4.37}$$

mit $\hat{y}(\vec{x}_i)$ der Klassenvorhersage für das Beispiel $\vec{x}_i$. Die a posteriori Wahrscheinlichkeit aus Gleichung (4.36) gibt also die empirische Wahrscheinlichkeit dafür an, dass das gegebene Beispiel $\vec{x}_i$ aus der Klasse $y_r$ stammt. Die Funktion $f$ bildet das Beispiel $\vec{x}_i$ auf die Klasse ab, für die die a posteriori Wahrscheinlichkeit am höchsten ist.

Diesem naiven Verfahren liegt die Annahmen zugrunde, dass alle Merkmale gleich gewichtet und voneinander unabhängig sind. Ein Merkmal hängt folglich nur von der Klasse ab. Obwohl diese Annahme bei den meisten realen Problemen falsch ist, liefert der Naive Bayes–Klassifikator häufig gute Ergebnisse.

### 4.5.1. Bayes–Netze

Durch die Nutzung von Bayes–Netzen können Wahrscheinlichkeitsverteilungen von Merkmalen kompakt repräsentiert werden. Bayes–Netze modellieren diese Abhängigkeiten mittels eines Graphen.

Ein *Graph* $G(V, E)$ besteht im Allgemeinen aus einer nicht leeren Menge von *Knoten* $V$ und einer Menge von *Kanten* $E$ mit $V \cap E = \emptyset$. Die Kanten sind Paare von Knoten $e = (v_i, v_j)$. Sie definieren also eine Abbildung, die jedem Element $e \in E$ genau ein Paar $(i, j) \in V$ zuordnet. Die Kanten eines Graphen können *gerichtet* oder *ungerichtet* sein. Bei ungerichteten Graphen ist jede Kante $e = (v_i, v_j)$ ein ungeordnetes Paar. Bei gerichteten Graphen ist jede Kante $e = (v_i, v_j)$ ein geordnetes Paar und hat somit eine Orientierung. In einem gerichteten Graphen bezeichnet ein *Pfad* eine Folge von Kanten $e_1, e_2, ..., e_p$, wobei $p > 1$ die Länge des Pfades ist. Der Endpunkt einer Kante $e_i$ ist gleichzeitig der Anfangspunkt der Kante $e_{i+1}$. Ist der Endpunkt der Kante $e_p$ auch der Anfangspunkt der Kante $e_1$, wird der vorliegende Pfad als *Zyklus* bezeichnet.

Ein wahrscheinlichkeitstheoretisches Modell, das eine Menge von Zufallsvariablen und ihre bedingten Abhängigkeiten mittels eines *gerichteten azyklischen Graphen* (engl. "directed acyclic graph", abgekürzt "DAG") darstellt, nennt man *Bayes'sches Netzwerk* oder *Bayes–Netz*. Dabei stellen die Zufallsvariablen die Knoten und die bedingten Wahrscheinlichkeiten die Kanten dar.

**Definition 4.7 (Bayes–Netz)** *Ein* Bayes–Netz *besteht aus einer Menge von Zufallsvariablen* $V = \{v_i\}_{i=1,...m}$ *und gerichteten Kanten* $E$ *zwischen den Variablen, die zusammen einen gerichteten azyklischen Graphen (DAG)* $G(V, E)$ *bilden. Die gemeinsame Wahrscheinlichkeitsdichte der Variablen berechnet sich als das Produkt der individuellen bedingten Wahrscheinlichkeitsdichten bezüglich der Elternvariablen, also durch*

$$p(V) = \prod_{i=1}^{m} p\big(v_i \,\big|\, v_{\mathrm{pa}(v_i)}\big)$$

*wobei* $pa(v_i)$ *die Menge der Elternknoten von* $v_i$ *ist.*

Jedem Knoten eines Bayes–Netzes ist demzufolge eine bedingte Wahrscheinlichkeit, gegeben die Variablen an den direkten "Vorgängerknoten", zugeordnet.

In dem vorliegenden Fall der Klassifikation von Systemcallvektoren bilden die Menge der Merkmale $\mathcal{F}$ und die Menge der Klassen $\mathcal{Y}$ die Knotenmenge $V = \{\mathcal{F} \cup \mathcal{Y}\}$. Das Lernen einer Netzwerkstruktur kann als Optimierungsproblem betrachtet werden, das für die gegebenen Beispiele aus der Menge $\mathcal{X}$ ein Qualitätsmaß wie die Genauigkeit der Zuordnung der Beispiele zu ihren Klassen maximiert. Für weitere Details sei an dieser Stelle auf [Bou04] verwiesen.

### 4.5.2. Das Multinomiale Bayes–Modell

Es gibt zwei Bayes–Modelle, die sich in der zugrundeliegenden Vektordarstellung der Merkmale unterscheiden [MN98]. Beim *multivariaten* Modell wird vorausgesetzt, dass der Beispielvektor aus binären Merkmalen besteht, das *multinomiale* Modell kann auch numerische Merkmalsvektoren handhaben. Da in Kapitel 5 vorwiegend numerische Vektoren klassifiziert werden sollen, wird das *multinomiale* Modell verwendet und hier näher erläutert.

Bei dem multinomialen Modell wird angenommen, dass die Merkmalsvektoren aus einem mit $\theta$ parametrisierten Mischverteilungsverfahren stammen. Bei einem Mischverteilungsverfahren wird davon ausgegangen, dass die Verteilung der Objekte eine Mischung aus verschiedenen, klassenspezifischen Wahrscheinlichkeitsverteilungen darstellt. Das Mischverteilungsverfahren besteht aus Komponenten $c_k \in \mathcal{C} = \{c_1, \ldots, c_{|\mathcal{C}|}\}$. Jede Komponente wird mit einer disjunkten Untermenge von $\theta$ parametrisiert. Es wird also davon ausgegangen, dass ein Beispiel $d_i$ erstellt wird, indem eine Komponente entsprechend der a priori Wahrscheinlichkeiten $p(c_k|\theta)$ gewählt wird und diese Komponente ein Beispiel mit der Verteilung $p(d_i|c_k;\theta)$ erzeugt. Die Wahrscheinlichkeit eines Beispiels kann also definiert werden als

$$p(d_i|\theta) = \sum_{k=1}^{|\mathcal{C}|} p(c_k|\theta)p(d_i|c_k;\theta). \tag{4.38}$$

Es wird weiter angenommen, dass eine Eins–zu–Eins–Korrespondenz zwischen den Klassen und den Komponenten des Mischverteilungsverfahrens besteht, so dass $c_k$ sowohl die Komponente als auch die Klasse anzeigt.

Angewendet auf die Klassifikation von Systemcall–Logdateien, wie sie in dieser Arbeit erfolgen soll, sei $d_i$ eine Logdatei. Es wird nun angenommen, dass die Länge der Datei unabhängig von der Klasse ist und dass die Wahrscheinlichkeit des Auftretens eines Systemcalls in dieser Datei unabhängig von dem Kontext und der Position des Systemcalls ist. Somit wird jede Datei $d_i$ aus einer multinomialen Verteilung gezogen. Dabei werden so viele unabhängige Versuche durchgeführt wie die Datei Systemcalls enthält.

Durch das in Abschnitt 3.4 beschriebene Vector Space Modell können Vektoren dieser Dateien erstellt werden. Sei also $d_{ij}$ der $TF$–Wert der $j$–ten Vektorkomponente, also die Anzahl der Vorkommen des Systemcalls $s_j$ in der Datei $d_i$. Dann entspricht die Wahrscheinlichkeit eines Beispiels gegeben seiner Klasse, der multinomialen Verteilung

$$p(d_i|c_k;\theta) = p(|d_i|)|d_i|! \prod_{j=1}^{m} \frac{p(s_j|c_k;\theta)^{d_{ij}}}{d_{ij}!}$$

mit $\mathcal{F} = \{s_1, \ldots, s_m\}$ der geordneten Merkmalsmenge und $|\mathcal{F}| = m$ der Anzahl der Systemcalls und somit der Anzahl der Vektorkomponenten. $s_j$ bezeichnet einen Systemcall. Die Parameter der erzeugenden Komponenten für jede Klasse sind die Wahrscheinlichkeiten für jede Vektorkomponente $s_j$, also $\theta_{s_j|c_k} = p(s_j|c_k;\theta)$, wobei $0 \leq \theta_{s_j|c_k} \leq 1$ gilt und $\sum_j \theta_{s_j|c_k} = 1$.

Die optimalen Schätzungen für diese Parameter können aus einer Menge gelabelter Trainingsdaten berechnet werden. Die Wahrscheinlichkeitsschätzung eines Merkmals $s_j$ in Klasse $c_k$ (die Klassen wurden in den Gleichungen (4.36) und (4.37) mit $y_r$ notiert) ist

$$\hat{\theta}_{s_j|c_k} = p(s_j|c_k;\hat{\theta}) = \frac{1 + \sum_{i=1}^{|D|} d_{ij}\, p(c_k|d_i)}{m + \sum_{j=1}^{m}\sum_{i=1}^{|D|} d_{ij}\, p(c_k|d_i)}$$

mit $D$ den gelabelten Systemcall–Logdateien.

Die a priori Parameter der Klassen werden berechnet durch

$$\hat{\theta}_{c_k} = p(c_k|\hat{\theta}) = \frac{\sum_{i=1}^{|D|} p(c_k|d_i)}{|D|}.$$

Bei gegebenen Schätzungen dieser Parameter kann die Klassifikation durch Anwendung des Bayes–Theorems aus Gleichung (4.35) erfolgen durch

$$p(c_k|d_i;\hat{\theta}) = \frac{p(c_k|\hat{\theta})\, p(d_i|c_k;\hat{\theta}_k)}{p(d_i|\hat{\theta})}.$$

## 4.6. Gütekriteren der Klassifikation

Die in diesem Kapitel beschriebenen Verfahren erstellen Klassifikationsmodelle durch unterschiedliche Herangehensweisen. Zur Herstellung einer Vergleichbarkeit und zur Bewertung eines Klassifikationsverfahrens muss das durch dieses Verfahren erstellte Modell getestet werden. Dabei kann eine Vielzahl an Leistungsmerkmalen berechnet werden, aufgrund derer anschließend die Güte des Modells beurteilt werden kann.

### Kreuzvalidierung zur Evaluation eines Modells

Die Generalisierungsfähigkeit eines überwachten Lernverfahrens kann mit Hilfe eines Trainings– und eines Testdatensatzes überprüft werden. Der Trainings– oder Lerndatensatz besteht aus einem Teil der Daten, die an den Lernalgorithmus übergeben werden. Daraus lernt er seine Funktion $f(\vec{x}_i) = \hat{y}$. Der Testdatensatz enthält die übrigen Daten, die mit Hilfe der bereits vorliegenden, gelernten Funktion klassifiziert werden, um anschließend die vorhergesagte Klasse $\hat{y}$ mit der "wahren" Klasse $y_i$ vergleichen zu können. Der Testdatensatz dient also der Bewertung des aufgestellten Modells.

Bei den in dieser Arbeit durchgeführten Experimenten, die in dem folgenden Kapitel 5 beschrieben werden, wurde die *Kreuzvalidierung* zur Modellbewertung verwendet. Bei einer Kreuzvalidierung werden die vorliegenden Daten $k$ Mal in zwei disjunkte Teilmengen zerlegt, so dass eine (größere) Trainingsmenge und eine (kleinere) Testmenge entstehen. Der Wert $k$ ist zu spezifizieren. Das Lernverfahren wird auf die unterschiedlichen Mengen

| | | Tatsächliche (wahre) Klasse | | Gütemaß |
|---|---|---|---|---|
| | | Positiv | Negativ | |
| Vorhersage | Positiv | True Positive (TP) | False Positive (FP) | **Precision** |
| | Negativ | False Negative (FN) | True Negative (TN) | |
| Gütemaß | | **Recall** | Specificity<br>Fallout | **Accuracy** |

Tabelle 4.3.: Konfusionsmatrix zur Beurteilung eines Klassifikators durch den Vergleich der vorhergesagten Klasse mit der tatsächlichen (wahren) Klasse. Die näher beschriebenen Maße sind fett markiert.

angewendet, so dass die Ergebnisse der einzelnen Durchläufe zu einer Gesamtbewertung des Modells zusammengefasst werden können. Zur Erzeugung der Trainings– und der Testmengen gibt es verschiedene Strategien.

**k–fache Kreuzvalidierung** Bei der *einfachen k–fachen Kreuzvalidierung* werden die Daten, bestehend aus $n$ Beispielen, in $k < n$ disjunkte Teilmengen $X_1,\ldots X_k$ zerlegt. Das Klassifikationsverfahren wird $k$ Mal angewendet, so dass jede der $k$ Teilmengen einmal als Testmenge genutzt wird und die verbleibenden Mengen jeweils die Trainingsmenge bilden. Die Genauigkeit des Modells wird dann als Durchschnitt der Werte der $k$ Durchläufe berechnet.

**k–fache stratifizierte Kreuzvalidierung** Bei der *stratifizierten Kreuzvalidierung* werden die $k$ Teilmengen so gewählt, dass jede der Mengen eine annähernd gleiche Verteilung besitzt. Dadurch wird die Varianz der Fehlerabschätzung verringert.

**Leave–One–Out Kreuzvalidierung** Bei der *Leave–One–Out Kreuzvalidierung* handelt es sich um einen Spezialfall der k–fachen Kreuzvalidierung, bei dem $k = n$ ist. Somit werden $n$ Durchläufe gestartet, in denen immer ein einzelnes Beispiel die Testmenge bildet.

### Gütekriterien

Es gibt es eine Vielzahl an Gütekriterien, deren Informativität problemabhängig ist. Im Folgenden werden die in dem nachstehenden Kapitel 5 angegebenen Kriterien *Accuracy* und *Recall* und darüber hinaus die *Precision* definiert.

Die Bedeutung und Berechnung dieser Maße lässt sich zunächst am besten mit Hilfe der in Tabelle 4.3 dargestellten Konfusionsmatrix anhand eines 2–Klassen–Problems nachvollziehen. In der Konfusionsmatrix werden der Vollständigkeit halber weitere populäre Gütekriterien genannt, die anhand dieser Matrix dargestellt werden können. Auf diese Kriterien wird hier allerdings nicht näher eingegangen.

Der bei den später folgenden Ergebnisdokumentationen angegebene Recall spezifiziert die Trefferquote eines Verfahrens. Daher wird dieses Maß häufig auch als True Positive Rate (TPR) bezeichnet.

**Definition 4.8 (Recall)** *Der* Recall *(TPR) gibt den Anteil der positiven Beispiele an, die tatsächlich erkannt werden und wird berechnet durch*

$$Recall = \frac{TP}{TP + FN}.$$

In dem Fall des in dieser Diplomarbeit vorliegenden Mehrklassenproblems bezeichnet $TP$ die Anzahl der Beispiele, die der *richtigen* Schadprogramm–Klasse zugeordnet wurden, also

$$TP = \sum_{y_c \in \mathcal{Y}} TP_{y_c}, \tag{4.39}$$

mit $TP_{y_c}$ der Anzahl der Beispiele $\vec{x}_i$, für die mit $y_c$ die richtige Klasse vorhergesagt wurde. $FP$ bezeichnet somit die Anzahl der Beispiele, die der *falschen* Klasse zugeordnet wurden, also

$$FP = \sum_{y_c \in \mathcal{Y}} FP_{y_c}, \tag{4.40}$$

mit $FP_{y_c}$ der Anzahl der Beispiele $\vec{x}_i$, für die die Klasse $y_c$ vorhergesagt wurde, deren richtige Klasse $y_i$ aber eine andere ist.

Dementsprechend wird der Recall bei dem Mehrklassenproblem *pro Klasse* berechnet und gibt den Anteil der Beispiele einer Klasse an, die dieser Klasse tatsächlich zugeordnet wurden.

Ein weiteres Gütekriterium ist die Precision, die ein Maß für die Genauigkeit eines Modells definiert.

**Definition 4.9 (Precision)** *Die* Precision *gibt den Anteil richtig positiver Vorhersagen im Verhältnis zu allen positiven Vorhersagen an und wird berechnet durch*

$$Precision = \frac{TP}{TP + FP}.$$

Bei einem Mehrklassenproblem kann dieser Wert ebenfalls pro Klasse berechnet werden und gibt dann den Anteil der einer Klasse zugeordneten Beispiele an, die dieser Klasse *richtigerweise* zugeordnet wurden.

Die Accuracy eines Modells gibt die Nähe der berechneten Werte zu ihren tatsächlichen (wahren) Werten an.

**Definition 4.10 (Accuracy)** *Die* Accuracy *eines Modells gibt die Genauigkeit bezüglich des wahren Ziels an und wird berechnet durch*

$$ACC = \frac{TP + TN}{TP + FP + TN + FN}.$$

Bezüglich eines Mehrklassenproblems gibt die Accuracy demnach den Anteil der Beispiele an, die durch die Anwendung des bewerteten Verfahrens der richtigen Klasse zugeschrieben wurden.

# 5. Eigene Ansätze

In dieser Diplomarbeit werden Betriebssystem–Logdateien analysiert, in denen die bei der Ausführung eines Schadprogramms aufgerufenen Systemcalls und deren Parameter und Parameterwerte protokolliert wurden. Das Ziel ist die Extraktion von Charakteristika verschiedener Schadprogramm–Familien und die automatische Klassifikation der Schadprogramme.

In Kapitel 3 wurden bereits einige verwandte Ansätze aus den Bereichen der Anomalieerkennung und der Klassifikation von Schadprogramm–Familien vorgestellt. Dort werden Logdateien als Textdokumente betrachtet und zur Anwendung maschineller Lernverfahren als Vektoren dargestellt. Die Vektordarstellung wird auch in dieser Arbeit verwendet. Das für die Konvertierung genutzte Vector Space Modell wurde in Abschnitt 3.4 beschrieben und auf Systemcall–Logdateien übertragen.

Es gibt verschiedene Möglichkeiten, Logdateien als Vektoren darzustellen. Im Folgenden wird bewertet, welche Vektordarstellungen für das vorliegende Problem sinnvoll sind und welche maschinellen Lernverfahren sich zur Klassifikation eignen. Es werden verschiedene Merkmalsmengen extrahiert und die Werte der Vektorkomponenten werden auf unterschiedliche Art und Weise berechnet. Die gelernten Modelle sollen neue Logdateien bekannter Schadprogramm–Familien möglichst zuverlässig einer der Familien zuordnen können.

Der nachfolgende Abschnitt 5.1 gibt einen Überblick über den Datensatz aus Systemcall–Logdateien, der in dieser Diplomarbeit verwendet wurde. Zur Erstellung von Vektoren erfolgt zunächst eine Konvertierung der Logdateien in eine Textdarstellung, die in Abschnitt 5.2 erläutert wird. Anschließend beschreibt Abschnitt 5.3 die Merkmalsextraktion und die Vektordarstellung auf der Basis des Vector Space Modells. Abschnitt 5.4 gibt einen kurzen Überblick über die erstellten Vektoren. Schließlich werden in Abschnitt 5.5 die durchgeführten Experimente und deren Ergebnisse dokumentiert und ausgewertet. Die verwendeten Lernverfahren wurden bereits in Kapitel 4 erläutert.

## 5.1. Datengrundlage dieser Arbeit

Wie bei den in Kapitel 3 vorgestellten Arbeiten zur Angriffserkennung und vielen anderen Ansätzen im Bereich der Intrusion Detection dienen auch in dieser Diplomarbeit Betriebssystem–Logdateien als Datengrundlage.

Der vorliegende Datensatz wurde im Rahmen des BSI–Projekts[13] AMSEL (Automatisch Malware Sammeln und Erkennen Lernen) am Lehrstuhl 6 der Fakultät Informatik an der Universität Dortmund erstellt und zu Analysezwecken verfügbar gemacht. Während die Datensätze, die den Experimenten der meisten verwandten Arbeiten zugrunde liegen größtenteils Aufzeichnungen UNIX–ähnlicher Systeme enthalten, handelt es sich

[13] BSI = Bundesamt für Sicherheit in der Informationstechnik

bei dem Datensatz des Lehrstuhl 6 um *Windows* API Calls. Diese Windows API Calls (Systemcalls) ermöglichen die Nutzung der vom Windows–Betriebssystem bereitgestellten Systemfunktionen. Der Zweck und die Aufgabe von Systemcalls wurden bereits in Abschnitt 3.2 erläutert.

Der Lehrstuhl 6 (LS6) hat zur Erstellung des Datensatzes Schadprogramme gesammelt und diese mit Hilfe des Tools *CWSandbox*[14] für eine festgelegte Zeit in einer Malware Sandbox[15] ausgeführt. Dabei wurden die während der Ausführung aufgerufenen Systemcalls aufgezeichnet. Durch das Aufsetzten der passenden Windows–Sandbox–Umgebung konnten die Zugriffe auf Dokumente, die Aufrufe von URLs und ähnliche Details erfasst werden. Die erzeugten Systemcall–Folgen liegen in einem XML–Format vor und werden im Folgenden als Logdateien, Berichte oder Traces bezeichnet. Sie enthalten insgesamt 120 verschiedene Systemcalls mit ihren Parameternamen und Parameterwerten.

Abbildung A.1 in Anhang A zeigt einen beispielhaften Auszug einer XML–Datei des LS6–Datensatzes. Die inhärente Baumstruktur der vorliegenden XML–Dateien ist in Anhang B, Abbildung B.2 dargestellt.

### Die Schadprogramm–Familien der Logdateien

Die Zuordnung einer XML–Logdatei zu einer Schadprogramm–Familie erfolgte durch den Upload der Schadprogramm–Binärdateien zu Virustotal. *Virustotal* [16] ist eine Vereinigung verschiedener Antivirussoftware–Hersteller. Die Binärdateien werden durch die verschiedenen Antivirus–Produkte analysiert und mit den Signaturen bekannter Schadprogramme in den Datenbanken verglichen. Wird eine passende Signatur gefunden, wird der Name der entsprechenden Malware zurückgegeben. Die Ergebnisse aller Antivirus–Engines werden dem Nutzer von Virustotal übermittelt.

Da verschiedene Antivirus–Produkte häufig unterschiedliche Namen für dasselbe Schadprogramm verwenden, wurde bei dem vorliegenden Datensatz der am häufigsten vergebene Schadprogramm–Name als Label für die mit dieser Malware erzeugten Logdateien vergeben. Bei den Ergebnissen der später folgenden Experimente sollte daher beachtet werden, dass die Label nicht zwingend eindeutig sind. Das bedeutet, dass bei Fehlklassifikationen der Logdateien nicht sicher ist, ob es sich tatsächliche um eine Fehlklassifikation handelt oder ob lediglich die Schadprogramm–Familie nicht richtig gewählt wurde.

Der Datensatz des LS6 besteht aus insgesamt 2.015 Logdateien, die 22 verschiedenen Schadprogramm–Familien (Klassen) zugeordnet und entsprechend gelabelt wurden. Die Verteilung der Dateien auf die Familien ist in Abbildung 5.1 zu sehen.

### Datensatz aus [RHW+08]

Zusätzlich zu dem beschriebenen Datensatz des LS6 standen die in [RHW+08] verwendeten Traces zur Verfügung. Neben 489 gutartigen Logdatei–Beispielen enthält dieser Datensatz maliziöse Beispiele, die mit Hilfe von im Jahr 2007 gesammelten und ebenfalls

[14] CWSandbox (http://www.sunbeltsoftware.com/Developer/Sunbelt-CWSandbox) ist ein kommerzielles Tool zur Aufzeichnung (und zur Analyse) der Systemcall–Folgen von Schadprogrammen.

[15] Eine Sandbox ist eine abgeschlossene, kontrollierte Umgebung, in der der Zugriff von Programmen auf die Systemumgebung kontrolliert wird.

[16] http://www.virustotal.com/de

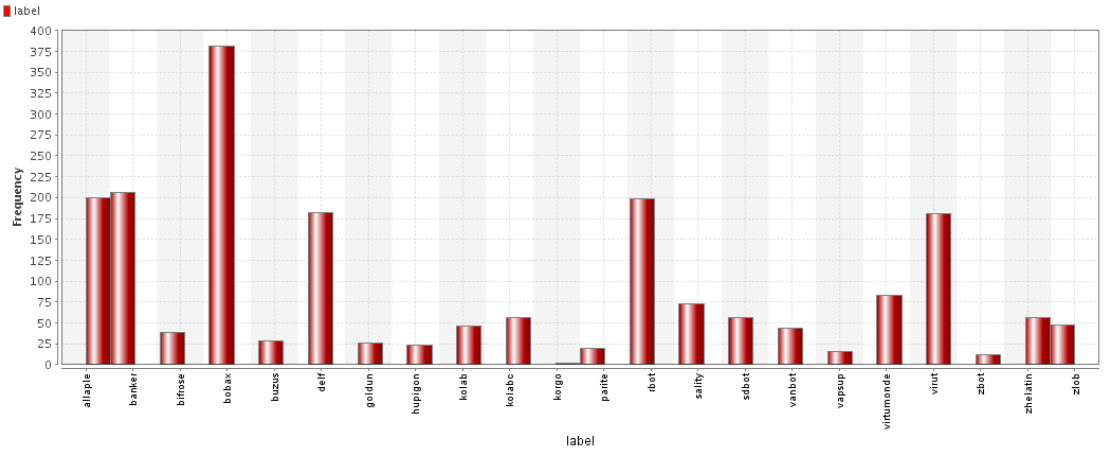



Abbildung 5.1.: Überblick über die Verteilung der 2.015 Logdateien des LS6–Datensatzes auf die 22 vorhandenen Schadprogramm–Familien.

in *CWSandbox* ausgeführten Schadprogrammen erstellt wurden. Die Schadprogramme wurden durch die Antivirussoftware des Herstellers *Avira Antivir*[17] gelabelt und die 14 häufigsten Schadprogramm–Familien wurden in [RHW+08] verwendet. Bei der ersten Vergabe der Label durch *Avira Antivir* konnten einige Schadprogramme nicht identifiziert werden. Diese Dateien wurden einige Wochen später erneut analysiert. Dabei konnten weitere 3.139 Beispiele den zuvor bereits ausgewählten Familien zugeordnet werden. 457 Beispiele wurden dabei 9 Schadprogramm–Familien zugeschrieben, die in dem ersten Datensatz *nicht* enthalten waren.

Leider liegt dieser Datensatz nur in der in [RHW+08] gewählten und hier in Abschnitt 3.7.2 beschriebenen, konvertierten Form vor. Die originalen *CWSandbox*–Berichte konnten nicht mehr zur Verfügung gestellt werden. Da ein wesentlicher Teil dieser Diplomarbeit in der Bewertung verschiedener Darstellungsformen und Detaillierungsgrade erstellter Systemcallvektoren liegt, konnte der Datensatz in dieser Arbeit nicht verwendet werden.

### Verwendeter Lern– und Testdatensatz

Die Experimente, die in den folgenden Abschnitten beschrieben werden, wurden folglich auf den Daten des LS6 durchgeführt. Dabei werden lediglich die 15 Schadprogramm–Familien berücksichtigt, für die die meisten Beispiele vorliegen.

Zur Evaluierung gelernter Modelle auf "neuen" Daten wurden für jede der 15 Familien 5 Berichte zufällig ausgewählt und in der Lernphase nicht berücksichtigt. Die ausgewählten Berichte bilden somit einen *Testdatensatz*, der 75 Logdateien enthält. Der *Lerndatensatz* umfasst 1.769 Berichte mit der in Abbildung 5.2 dargestellten Verteilung auf die 15 ausgewählten Schadprogramm–Familien.

[17] http://www.avira.com/

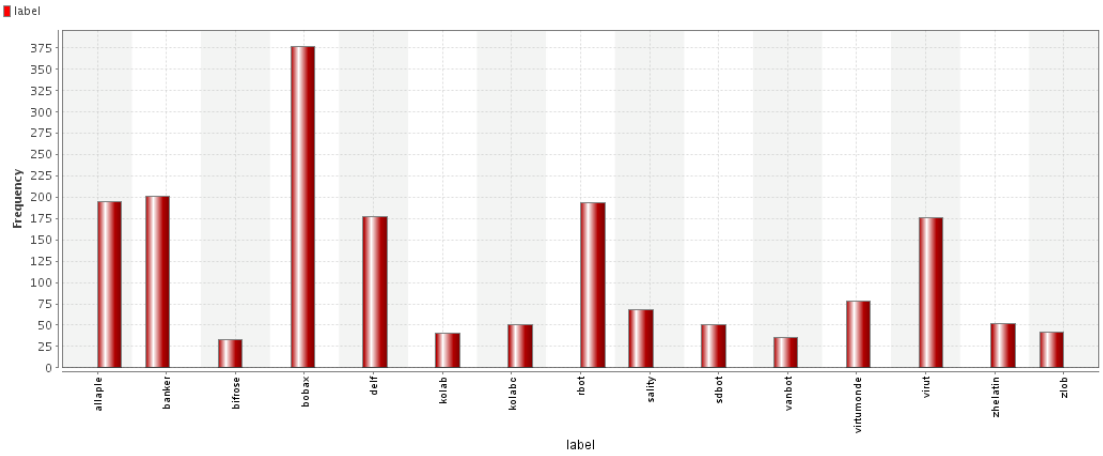



Abbildung 5.2.: Überblick über die Verteilung des Lerndatensatzes bestehend aus 1.769 Berichten der 15 Schadprogramm–Familien des LS6–Datensatzes, für die die meisten Beispiele vorliegen.

## 5.2. Konvertierung von Systemcall–Logdateien

Zur Anwendung maschineller Lernverfahren müssen die Daten in einer passenden Darstellungsform vorliegen. Bei einigen der in Kapitel 3 beschriebenen Ansätze wurde die Vektor–Repräsentation von Logdateien genutzt und erwies sich als geeignet. Die Vorteile der Vektordarstellung, auf die bereits in Abschnitt 3.4 hingewiesen wurde, liegen in der Möglichkeit zur Nutzung der vielfältigen Methoden aus dem Bereich der Textanalyse. In dieser Diplomarbeit wird die Vektordarstellung daher ebenfalls verwendet.

Der zur Transformation von Systemcall–Logdateien in eine Vektordarstellung durchzuführende Prozess ist in Abbildung 5.3 schematisch dargestellt. Der erste Schritt besteht

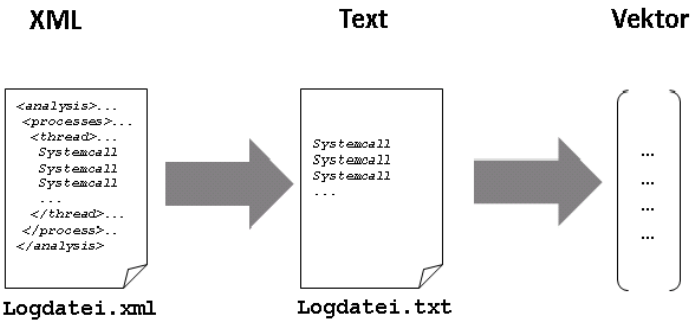



Abbildung 5.3.: Prozess der Konvertierung einer XML–Datei in eine Vektordarstellung

demnach in der Konvertierung von XML–Logdateien (Beispiel siehe Abbildung A.1) in eine Textdarstellung. Dieser Schritt ist notwendig, damit das in Abschnitt 3.4 erläuterte Vector Space Modells zur Erstellung eines Vektors angewendet werden kann. Die Konvertierung der Logdateien wird in dem vorliegenden Abschnitt beschrieben. Die Schilderung

des zweiten Schrittes, der die Vektor–Erstellung auf der Grundlage der Textdateien beinhaltet, findet sich in Abschnitt 5.3.

### Verschiedene Detaillierungsgrade der Textdateien

In den XML–Logdateien sind neben den aufgezeichneten Systemcalls auch deren Parameter und Parameterwerte enthalten. Um die Bedeutung von Parameternamen und/oder Parameterwerten für die Klassifikation von Schadprogramm–Familien bewerten zu können, wurden diverse Parser in Java implementiert, die unterschiedlich detaillierte Textdarstellungen der Logdatei–Beispiele erzeugen. Eine Zeile der erzeugten Textdatei enthält *einen* Systemcall mit oder ohne seine Parameternamen und/oder –werte. Die Reihenfolge der Systemcalls wird aus dem XML–Bericht übernommen.

Abbildung 5.4 zeigt den Ausschnitt einer erstellten Textdatei wenn *keine* Parameternamen und *keine* Parameterwerte berücksichtigt werden. Der in einigen Systemcalls ursprünglich vorhandene "_" wurde entfernt und der auf den Unterstrich folgende Buchstabe wurde zur besseren Lesbarkeit groß geschrieben. So wird zum Beispiel aus dem Systemcall "`load_dll`" der Term "`loadDll`". Abbildung 5.5 zeigt denselben Ausschnitt einer Textdatei *mit* Parameter*werten*, jedoch *ohne* die Parameter*namen*.

```
...
loadDll
openKey
queryValue
openFile
...
```

Abbildung 5.4.: Auszug aus der Textdarstellung einer Systemcall–Logdatei ohne Parameternamen und Parameterwerte.

```
...
loadDll(WS2_32.dll,1,$71A10000,94208,hash_error)
openKey(HKEY_LOCAL_MACHINE\Software\Microsoft\Advanced INF Setup)
queryValue(HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Advanced INF Setup,
 AdvpackLogFile)
openFile(file,c:\000b9fd156bdb01f38941e7dee6fd4c9,
 000b9fd156bdb01f38941e7dee6fd4c9,OPEN_EXISTING,FILE_ANY_ACCESS,
 SHARE_READ,FILE_ATTRIBUTE_NORMAL,SECURITY_ANONYMOUS,FileBasicInformation)
...
```

Abbildung 5.5.: Auszug aus der Textdarstellung einer Systemcall–Logdatei mit Parameterwerten. Aus Darstellungsgründen wurden zusätzliche Zeilenumbrüche eingefügt.

Um bewerten zu können, ob die Berücksichtigung *einzelner* Parameterwerte oder einer *Teilmenge* von Parameterwerten die Ergebnisse eines Lernverfahrens zur Klassifikation von Schadprogramm–Familien verbessern kann, besteht zudem die Möglichkeit der selektiven Angabe bestimmter Systemcalls und Parameter, die bei der Konvertierung der Logdatei in eine Textdarstellung berücksichtigt werden sollen. Der in den vorangegangenen Beispielen bereits verwendete Auszug ist für diesen Fall in Abbildung 5.6 dargestellt.

Dort wird lediglich der Parameter `srcfile` des Systemcalls `openFile` berücksichtigt. Eine detailliertere Beschreibung der implementierten Parser ist in Anhang A zu finden.

```
...
loadDll
openKey
queryValue
openFile(srcfile=c:\000b9fd156bdb01f38941e7dee6fd4c9)
...
```

Abbildung 5.6.: Auszug aus der Textdarstellung einer Systemcall-Logdatei bei selektiver Auswahl der zu übernehmenden Parameterwerte.

**Parameternamen** Jeder Systemcall kann eine fest vorgegebene Menge an Parametern spezifizieren. Dabei gibt es neben den Parameterwerten, die bei dem Aufruf des Systemcalls angegeben werden *müssen*, einige optionale Parameter. Die Namen der zu einem Systemcall gehörenden Parameter sind immer identisch. Nur die *Anzahl* variiert durch das Auslassen optionaler Angaben. Daher ist die Berücksichtigung der Parameter*namen* beim maschinellen Lernen eigentlich überflüssig. Ein Algorithmus kann aus diesen Angaben keine zusätzlichen Informationen gewinnen. Allerdings können Parameternamen zum Verständnis der Modelle sehr hilfreich sein und werden daher hier trotzdem stellenweise mit in die Textdateien übernommen.

### Extraktion einzelner Threads

Die zu konvertierenden XML–Logdateien sind hierarchisch strukturiert. Der schematische Aufbau einer Logdatei ist in Abbildung 5.7 zu sehen. Die Dateien enthalten einen oder mehrere Prozesse. Ein Prozess kann in einige *Threads*[18] (Ausführungsstränge) verzweigen, deren Systemcalls mit ihren Parametern und Parameterwerten in der Logdatei protokolliert sind.

Bei der bereits beschriebenen Konvertierung ganzer Berichte in die Textdarstellung wird die Dateistruktur nicht berücksichtigt. Die Systemcalls werden einfach in der Reihenfolge ihres Vorkommens in der XML–Datei in die Textdarstellung übernommen. Der Beginn oder das Ende eines Prozesses oder eines Threads werden nicht beachtet.

Zu Beginn dieser Diplomarbeit stellte sich die Frage, ob es sinnvoller ist einzelne Threads oder ganze Berichte zu klassifizieren. Für die Klassifikation einzelner Threads spricht, dass ein Angriff nicht zwingend ausschließlich maliziöse Muster enthalten muss. Teile der Ausführungen können tatsächlich normales Verhalten darstellen. Zudem kann bei der Klassifikation von Teilen eines Berichtes unter Umständen bereits vor Beendigung eines Angriffs die Zuordnung zu einer Schadprogramm–Familie erfolgen. Durch die Kenntnis der Malware kann das weitere Verhalten des Programms dann eingeschätzt werden und es ist möglich, passende Gegenmaßnahmen bereits frühzeitig zu ergreifen.

[18] Threads sind Teil–Prozesse, die sich Ressourcen des erzeugenden Prozesse wie zum Beispiel den Speicher teilen.

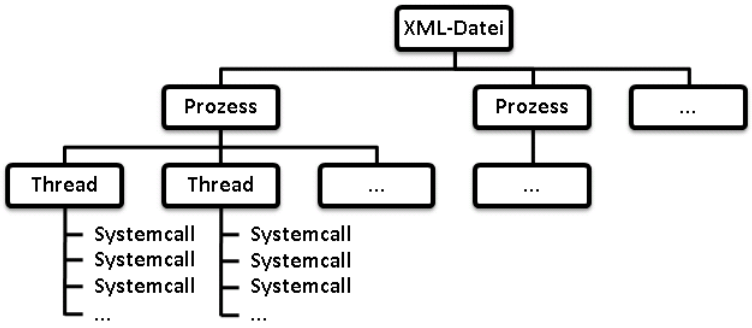



Abbildung 5.7.: Schematischer Aufbau einer XML–Datei

Aufgrund dieser Vorteile werden in den folgenden Abschnitten einzelne *Threads* der Berichte klassifiziert.

Anhand der Logdateistruktur können die Threads als Teile der Prozessausführung eines Schadprogramms aus der XML–Datei extrahiert werden. Die zur Konvertierung einer XML–Logdatei in eine Vektordarstellung einzelner Threads notwendigen Schritte sind basierend auf der schematischen Darstellung aus Abbildung 5.3 in Abbildung 5.8 skizziert. Es werden zunächst die Threads extrahiert. Anschließend wird das bereits für ganze Berichte beschriebene Verfahren auf diese Threads angewendet. Folglich können die verschiedenen Detaillierungsgrade der Textdateien bezüglich der Parameternamen und der Parameterwerte bei dieser Darstellung der Logdateien gleichermaßen erzeugt werden.

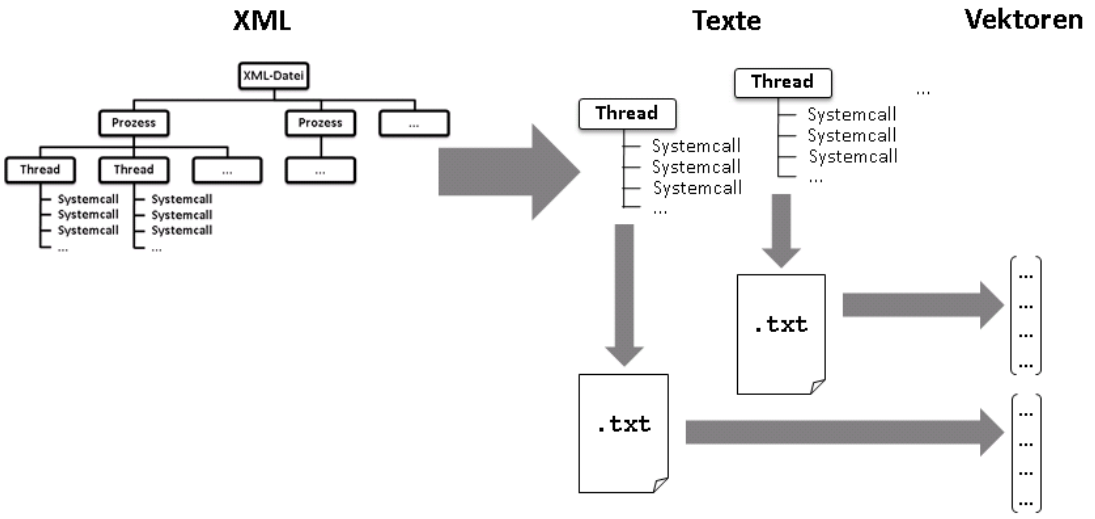



Abbildung 5.8.: Prozess der Konvertierung einer XML–Logdatei in eine Vektordarstellung einzelner Threads.

**Dateinamen der Thread–Textdateien** Zur Nachvollziehbarkeit der Zugehörigkeit einer Thread–Datei zu einer Logdatei setzen sich die Namen der Thread–Textdateien folgendermaßen zusammen:

```
<Name XML-Logdatei>_<process-id>_<thread-id>.txt
```

Auch hier enthält *eine* Zeile der Datei *einen* Systemcall (mit Parameternamen und/oder Parameterwerten) und die Reihenfolge der Systemcalls wird aus dem XML–Bericht übernommen.

### Automatische Sortierung der Textdateien nach Schadprogramm–Familien

Wie in dem vorangegangenen Abschnitt 5.1 beschrieben wurde, ist zu jeder XML–Datei eine Schadprogramm–Familie gegeben. Die Zuordnung erfolgt über den Namen der XML–Datei und wurde in Form einer separaten "Label–Datei" zur Verfügung gestellt. Diese Datei kann an einen der implementierten Parser übergeben werden, der die erstellten Textdateien dann automatisch nach Familien sortiert und in entsprechend benannten Verzeichnissen speichert. Dadurch wird das Einlesen der Daten in den später vorgestellten Experimenten wesentlich erleichtert.

**Verteilung der Textdateien auf die Klassen** Die Verteilung der XML–Berichte auf die 15 Klassen des in Abschnitt 5.1 beschriebenen Lerndatensatzes wurde bereits in Abbildung 5.2 dargestellt. Bei der Erstellung einer Textdatei pro Logdatei erfolgt eine 1:1–Abbildung der Beispiele, so dass sich diese Verteilung nicht ändert.

Allerdings weicht die Verteilung der *Threads* von der Verteilung ganzer Traces stark ab. Ein Thread wird mit der Schadprogramm–Familie des XML–Traces gelabelt, aus dem er extrahiert wurde. Die Anzahl der pro Trace vorhandenen Threads variiert von Familie zu Familie stark. Die durchschnittliche Anzahl an Threads in einem Trace ist in Abbildung 5.9 für jede Familie dargestellt.

Bei der Konvertierung der XML–Berichte des *Lerndatensatzes* durch die Extraktion einzelner Threads entsteht ein Datensatz aus 129.088 Beispielen. Die Verteilung der Beispiele auf die 15 ausgewählten Familien ist in Abbildung 5.10 zu sehen.

Der konvertierte Testdatensatz enthält 3.920 Threads mit der in Abbildung 5.11 dargestellten Verteilung auf die bereits bekannten Schadprogramm–Familien.

### Bereinigter Datensatz

Nach der Durchführung der ersten Experimente fiel auf, dass in den XML–Dateien einige Threads existieren, für die keine Systemcallaufrufe aufgezeichnet wurden. Das hat zur Folge, dass der Lerndatensatz 470 und der Testdatensatz 12 leere Dateien enthält. Nach Löschung dieser Dateien enthält der Lerndatensatz folglich noch 128.618 Beispiele und der Testdatensatz 3.908 Beispiele. Einige der bereits beendeten Experimente wurden auf diesem leicht veränderten Datensatz erneut durchgeführt. Die Ergebnisse *verbesserten* sich geringfügig. Aufgrund zeitlicher Restriktionen und der Tatsache, dass dieser bereinigte Datensatz keinen signifikanten Einfluss auf das Lernergebnis hatte, wurden nicht alle der bereits durchgeführten Experimente wiederholt. Das bedeutet, dass die Ergebnisse der Experimente, die zu Vergleichs–Zwecken an verschiedenen Stellen wiederholt

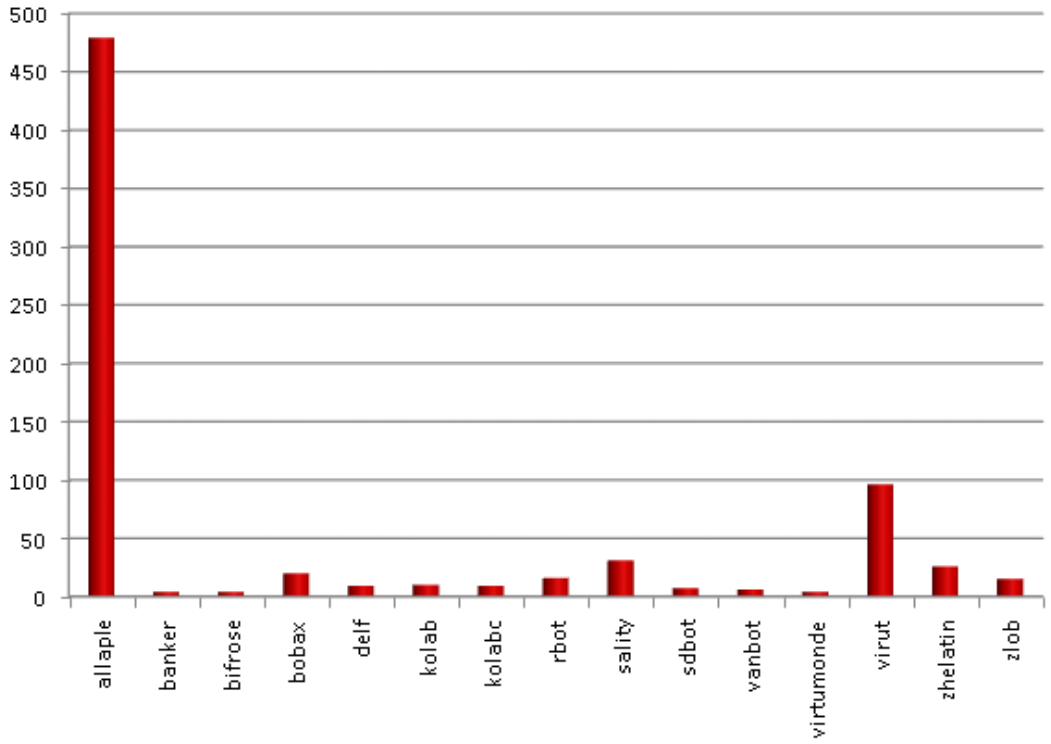



Abbildung 5.9.: Die Durchschnittswerte der Anzahl an Threads in einem XML–Bericht pro Familie für die 15 Schadprogramm–Familien des Lerndatensatzes.

angegeben werden, stellenweise leicht voneinander abweichen. An den betroffenen Stellen wird auf diesen Umstand jedoch noch einmal hingewiesen, indem die verwendete Datengrundlage mit einem Stern (“*”) gekennzeichnet ist.

## 5.3. Merkmalsextraktion

In dem vorangegangenen Abschnitt 5.2 wurde die Konvertierung der Systemcall–Logdateien in ein Textformat beschrieben. Wie in den Abbildungen 5.3 und 5.8 schematisch dargestellt wurde, müssen diese Textdateien zur Anwendung maschineller Lernverfahren nun in eine Vektordarstellung transformiert werden. Dazu muss zunächst eine Merkmalsextraktion aus den Strings der Textdateien erfolgen. Die Merkmale bilden dann die Dimensionen der Vektoren. Es werden verschiedene Merkmalsmengen extrahiert, um später die Abhängigkeiten zwischen der Lernbarkeit eines Modells und den verschiedenen Datenrepräsentationen bewerten zu können.

### 5.3.1. Transformation der Textdateien

Wie bereits beschrieben, wird die Transformation der Textdateien in eine Vektordarstellung durch die Anwendung des *Vector Space Modells* aus dem Bereich der Textanalyse

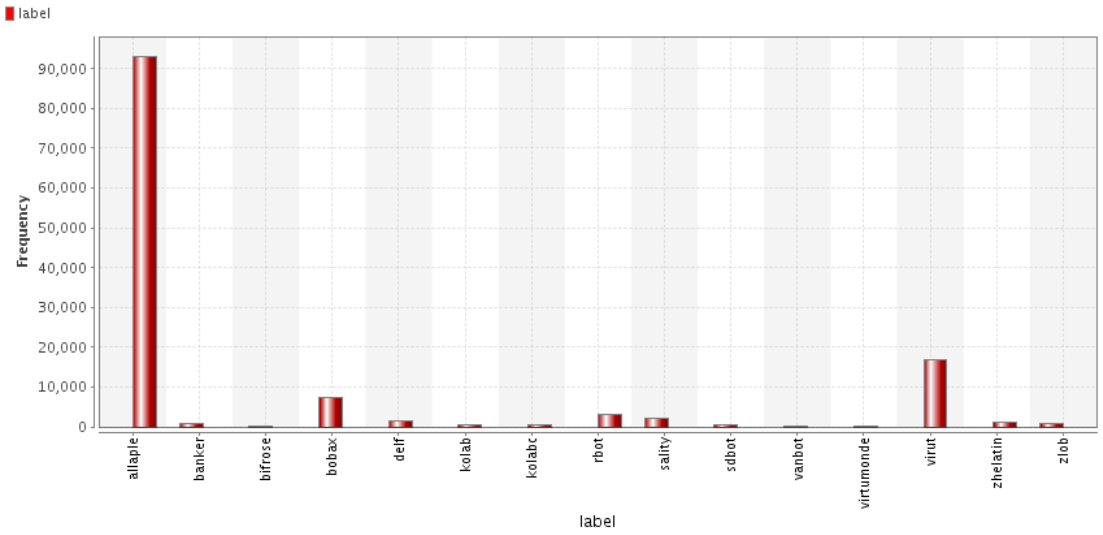



Abbildung 5.10.: Überblick über den Lerndatensatz aus 129.088 Threads, verteilt auf die 15 Schadprogramm–Familien des Lerndatensatzes.

vorgenommen. Die grundlegenden Ideen des Modells, einige elementare Definitionen in diesem Zusammenhang und die Übertragung des Vector Space Modells auf Systemcall–Logdateien wurden bereits in Abschnitt 3.4 aufgezeigt. Neben der dort verwendeten Merkmalsmenge einzelner Systemcalls werden im Folgenden auch die Parameternamen und/oder Werte der Systemcalls berücksichtigt und es werden Merkmalsmengen aus Systemcall–Sequenzen und Systemcall–Mengen aus den Textdateien extrahiert.

Systemcall–Sequenzen wurden bereits in [FHSL96], [LS98] und [ESL01] (siehe Abschnitt 3.3) zur Erkennung von Anomalien erfolgreich verwendet. Ähnlich wie bei diesen Ansätzen werden die Sequenzen der Länge $k$ auch hier durch ein gleitendes Fenster der Größe $k$ erzeugt, das über die Systemcall–Folgen der Textdateien geschoben wird. Mengen der Größe $k$ werden extrahiert, indem die in einem Fenster enthaltenen Sequenzen sortiert werden, so dass die Reihenfolge keine Rolle mehr spielt. Mengen wurden bereits in [KFH05] (Abschnitt 3.5.1) erfolgreich zur Anomalieerkennung genutzt.

Im Folgenden bildet demnach ein Systemcall mit oder ohne Parameternamen und/oder Parameterwerten, eine Systemcall–Menge oder eine Systemcall–Sequenz ein Merkmal der geordneten Merkmalsmenge $\mathcal{F}$. Die Elemente der Menge $\mathcal{F}$ bilden die Vektordimensionen. Zur Berechnung der Vektorkomponenten, also der Merkmalswerte eines Beispiels, werden die $TF$–, $BO$– und $TFIDF$–Werte genutzt, die in Abschnitt 3.4 (Definitionen 3.1, 3.2 und 3.5) definiert wurden. Die Erzeugung der Vektordarstellung $\vec{x}$ einer Textdatei $d$ aus der Menge aller in die Textdarstellung konvertierten Logdateien $\mathcal{D}$ kann folglich entsprechend der in [RHW+08] (Abschnitt 3.7.2) verwendeten Notation als eine Abbildung

$$\begin{aligned} \phi : \mathcal{D} &\rightarrow \mathbb{R}^{|\mathcal{F}|}, \\ d &\mapsto (f(d,s))_{s \in \mathcal{F}} \end{aligned} \tag{5.1}$$

definiert werden.

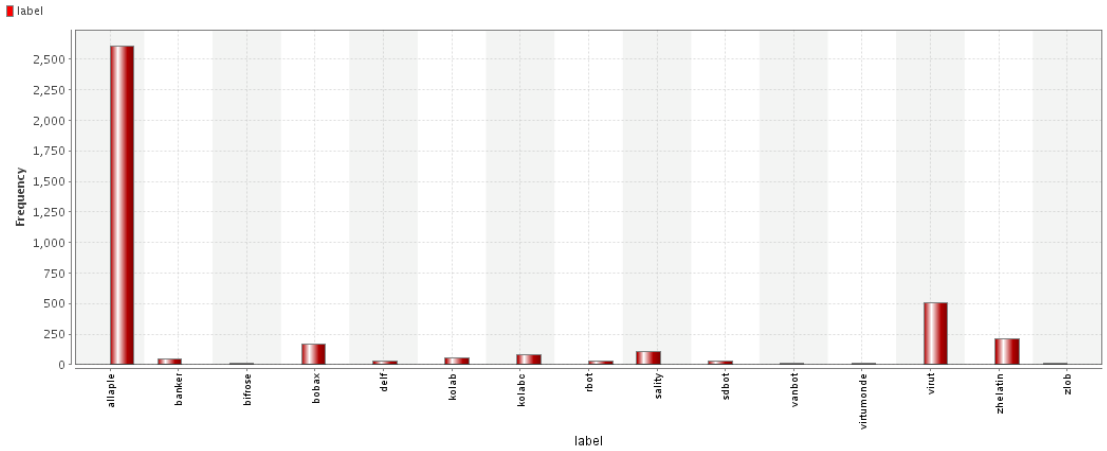



Abbildung 5.11.: Überblick über den Testdatensatz aus 3.920 Threads, verteilt auf 15 Schadprogramm–Klassen.

In Abschnitt 3.4 wurde bereits ein Beispiel vorgestellt, das die Abbildung einer Textdatei in eine Vektordarstellung aufzeigt, in der eine Vektordimension einem Systemcall entspricht. Dieses Beispiel soll hier zur Verdeutlichung auf Systemcall–*Sequenzen* übertragen werden. Sei dazu eine bereits in die Textdarstellung konvertierte Logdatei $d$ wieder gegeben durch

$$d = \texttt{loadDll loadDll openKey queryValue openFile}.$$

Die Merkmalsmenge aus 2er–Sequenzen enthält für diese Datei somit die Merkmale

(`loadDll,loadDll`),
(`loadDll,openKey`),
(`openKey,queryValue`),
(`queryValue,openFile`).

Die vollständige, geordnete Merkmalsmenge $\mathcal{F}$ aus den 2er–Sequenzen aller Logdateien $\mathcal{D}$ des Datensatzes sei gegeben durch

$s_1$ =(`loadDll,loadDll`),
$s_2$ =(`loadDll,openKey`),
$s_3$ =(`openKey,loadDll`),
$s_4$ =(`openKey,queryValue`),
$s_5$ =(`loadDll,openFile`),
$s_6$ =(`queryValue,openFile`),
$s_7$ =(`openFile,sleep`),
$s_8$ =(`sleep,writeValue`).

Es gilt also $\mathcal{F} = \{s_1, \ldots, s_8\}$ und die Vektordimension beträgt somit $m = 8$. Bei der Berechnung der Komponentenwerte des Vektors durch die Termfrequenz $TF$ wird die

Datei $d$ folglich abgebildet auf den Vektor

$$\vec{x} = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \begin{matrix} (\texttt{loadDll,loadDll}) \\ (\texttt{loadDll,openKey}) \\ (\texttt{openKey,loadDll}) \\ (\texttt{openKey,queryValue}) \\ (\texttt{loadDll,openFile}) \\ (\texttt{queryValue,openFile}) \\ (\texttt{openFile,sleep}) \\ (\texttt{sleep,writeValue}). \end{matrix}$$

### 5.3.2. Merkmalsextraktion mit RapidMiner

Als Lernumgebung zur Durchführung verschiedener Experimente wird in dieser Diplomarbeit die Open–Source–Variante der Data Mining Anwendung RapidMiner[19] genutzt. Die Umgebung wurde 2001 unter dem Namen YALE ("Yet Another Learning Environment") vom Lehrstuhl für Künstliche Intelligenz an der Universität Dortmund entwickelt und gehört mittlerweile zu den weltweit führenden Open–Source–Anwendungen im Bereich des Data Minings. Aus einer Vielzahl an Operatoren für die Ein– und Ausgabe von Daten, die Datenvorverarbeitung und verschiedene maschinelle Lernverfahren bis hin zur Datenvisualisierung und der Aufbereitung von Ergebnissen kann der Nutzer über eine graphische Oberfläche Experimente erstellen, die in einem XML–Format gespeichert werden. Im Rahmen des verfügbaren Text–Plugins stellt RapidMiner auch eine Vielzahl an Operatoren zur Verarbeitung von Texten zur Verfügung. Unter anderem ist die Erstellung von Vektoren möglich.

In diesem Abschnitt werden die genutzten Operatoren des Text–Plugins und die notwendigen Anpassungen der darin implementierten Funktionen beschrieben. Eine für die im folgenden durchgeführten Experimente beispielhafte Prozesskette in RapidMiner ist in Anhang B, Abbildung B.1 dargestellt.

#### Einlesen der Daten

Der *TextInput–Operator* des RapidMiner Text–Plugins nutzt das *Wordvektor–Tool*, welches wiederum den Kern des Text–Plugins bildet. Das *Wordvektor–Tool* ist eine flexible Java–Bibliothek zur statistischen Sprachmodellierung.

Der Parameter `texts` des *TextInput*–Operators bietet die Möglichkeit, Schlüssel–Wert–Paare von Klassen und Verzeichnissen zu definieren. Wurde einem der in Abschnitt 5.2 beschriebenen Parser eine Label–Datei mit den Zuordnungen der Logdateien zu den Schadprogramm–Familien übergeben, liegen die erstellten Textdateien nach Schadprogramm–Familien sortiert in entsprechend benannten Verzeichnissen vor. Diese Verzeichnisse können nun als "Wert" des `texts`–Parameters angegeben werden. Alle Textdokumente eines Verzeichnisses werden mit der Klasse gelabelt, die dem `texts`–Parameter als Schlüssel für dieses Verzeichnis übergeben wird. Für den Fall der Schadprogramm–Familien wird der Name der in dem Verzeichnis vorliegenden Familie als Schlüssel gewählt.

[19] http://www.rapid-i.com

### Extraktion der Merkmalsmenge

Der in RapidMiner zu erzeugende Datensatz (das *ExampleSet*) soll eine Zeile für jedes Textdokument und eine Spalte für jedes Wort oder jeden Term enthalten. Zur Extraktion der Wörter/Terme aus den in den Textdateien vorhandenen Strings benötigt der TextInput–Operator mindestens einen inneren Operator. Dieser Operator gibt die Merkmalsmenge (die Term–Menge) zur Erstellung der Vektoren zurück. Es stehen diverse Operatoren zur Verfügung. Im Folgenden wird der *StringTokenizer* verwendet.

**Extraktion einzelner Systemcalls** Der *StringTokenizer* des RapidMiner Text–Plugins unterteilt den in Form eines einzigen Strings vorliegenden Text in individuelle Einheiten. Der Operator verwendet die Unicode–Spezifikation[20] um zu entscheiden, ob ein Zeichen ein Buchstabe ist. Von allen Zeichen, die nicht als Buchstaben betrachtet werden wird angenommen, dass sie Trennelemente sind. Somit beinhalten die entstehenden Token nur Buchstaben.

In den übersetzten XML–Logdateien sind bei der Berücksichtigung von Parameterwerten Symbole und Zahlen vorhanden, bei denen der StringTokenizer eine Trennung vornimmt. Diese Trennung ist hier aber gerade *nicht* erwünscht. Jeder Systemcall soll zusammen mit seinen Parameternamen und –werten *ein* Merkmal bilden und somit als *ein* Term dargestellt werden. Da die Löschung der trennenden Zeichen einen Informationsverlust für die zu lösende Lernaufgabe bedeuten würde, wurde der StringTokenizer stattdessen angepasst. Der neue Operator heißt *CustomTokenizer* und bietet dem Nutzer die Möglichkeit zur Nennung aller Zeichen, die neben den gemäß der Unicode–Spezifikation definierten Buchstaben ebenfalls als Buchstaben betrachtet werden sollen. Bei diesen Zeichen erfolgt demnach *keine* Trennung der Strings. So wurde erreicht, dass die Merkmale $\mathcal{F}$ aus den Textbeispielen extrahiert werden, indem lediglich Leerstellen den Beginn eines neuen Terms anzeigen. Ein Merkmal $s_j$ der Merkmalsmenge $\mathcal{F}$ stellt somit einen Systemcall mit oder ohne Parameternamen und/oder –werte dar.

**Generierung von Sequenzen und Mengen** Das Text–Plugin bietet auch einen Operator an, der nach kleineren Anpassungen für die Erstellung von Sequenzen und Mengen verwendet werden kann. Der *TermNGramGenerator* erzeugt in seiner ursprünglichen Implementierung Term–N–Gramme aus Termfolgen, die ihm durch einen Operator wie den *StringTokenizer* übergeben werden. Ein *N–Gramm* ist eine Teilfolge bestehend aus $n \leq N$ aufeinanderfolgenden Elementen einer gegebenen Sequenz. N–Gramme sind eine Art statistisches Modell zur Vorhersage des nächsten Elements in einer Sequenz. Die maximale Länge $N$ der N–Gramme kann bei dem Operator durch den Parameter `max_length` festgelegt werden. Der Operator erzeugt dann 1-Gramme (gleichbedeutend mit einem Term), 2-Gramme, . . . , bis `max_length`-Gramme für jede übergebene Termfolge eines Textes und alle Fenster darüber.

Zur Bildung von Systemcall–Sequenzen sollen bei der Analyse von Logdateien lediglich Sequenzen der Länge $N$ erzeugt werden. Untersequenzen der $N$–Sequenzen sollen nicht betrachtet werden. Der *TermNGramGenerator* wurde folglich so angepasst, dass die erstellten Sequenzen dem Inhalt eines gleitenden Fensters der Größe $N$ entsprechen, das

[20] http://www.unicode.org/

über die Termfolgen der Beispiele geschoben wird.

Zur Erzeugung von Systemcall–Mengen wurde der TermNGramGenerator durch eine Sortierfunktion erweitert, die über eine Checkbox (`order`) aktiviert werden kann. Die erzeugten N–Sequenzen werden dann alphabetisch sortiert, so dass sie nun Mengen darstellen.

### Transformation in die Vektordarstellung

Nachdem die Merkmale extrahiert wurden, können auf dieser Basis die Vektorkomponenten für ein Beispiel berechnet werden. Dazu verlangt der TextInput–Operator die Angabe eines Schemas. Der Nutzer kann wählen zwischen den Schemata `TermFrequency`, `TermOccurences`, `TFIDF` und `BinaryOccurences`.

`TermFrequency` Die `TermFrequency`–Darstellung wird durch $\frac{TF(w_i,d)}{f_d}$ berechnet. Dabei ist $f_d$ die Gesamtanzahl der Terme in einer Textdatei $d$. Der resultierende Vektor ist auf euklidische Länge normalisiert.

`TermOccurences` Die `TermOccurences`–Darstellung entspricht den $TF(w_i, d)$–Werten aus Definition 3.1, also der absoluten Anzahl der Vorkommen eines extrahierten Merkmals in einem Dokument. Der resultierende Vektor ist *nicht* normalisiert.

`TFIDF` Die `TFIDF`–Darstellung wird gemäß der Definition 3.5 berechnet. Der für jedes Dokument resultierende Vektor ist auf euklidische Länge normalisiert.

`BinaryOccurences` Die `BinaryOccurences`–Darstellung wird gemäß der Definition 3.2 berechnet. Der resultierende Vektor ist *nicht* normalisiert.

Für die in Gleichung (5.1) definierte Abbildung gilt folglich

$$f = \begin{cases} \texttt{TermFrequency} & \text{oder} \\ \texttt{TermOccurences} & \text{oder} \\ \texttt{TFIDF} & \text{oder} \\ \texttt{BinaryOccurences}. & \end{cases}$$

Die Terme, die hier mit $w_i$ notiert werden, entsprechen einem Merkmal der geordneten Merkmalsmenge $\mathcal{F}$. Die Menge $\mathcal{F}$ besteht je nach Konvertierung der Daten und der gewählten Merkmalsextraktion aus einzelnen Systemcalls mit oder ohne Parameternamen und/oder –werten, aus Systemcall–Sequenzen oder aus Systemcall–Mengen.

## 5.4. Ein erster Überblick über die Systemcallvektoren

Zur Gewinnung eines ersten Überblicks über die zu klassifizierenden Threads des LS6–Datensatzes wurden Vektoren auf der Basis einer Merkmalsmenge bestehend aus einzelnen Systemcalls ohne Parameternamen und –werte erstellt. Um herauszufinden, ob es

Systemcalls gibt, deren Häufigkeiten eine Klasse identifizieren können, wurden die Vektorkomponenten dieser Merkmalsmenge durch $TF$–Werte berechnet. Die *durchschnittlichen $TF$–Werte* jedes Systemcalls pro Schadprogramm–Familie sind in Diagramm 5.12 dargestellt. Wie zu sehen ist, gibt es zwar unterschiedliche Ausprägungen der Häufigkeiten für verschiedene Systemcalls und Klassen, allerdings können aufgrund dieser Darstellung keine Werte festgemacht werden, die als charakteristisch für eine Schadprogramm–Familie betrachtet werden können.

Auch die *durchschnittlichen $TFIDF$–Werte* pro Familie, dargestellt in Diagramm 5.13, liefern keine eindeutigen Hinweise auf eine Klasse.

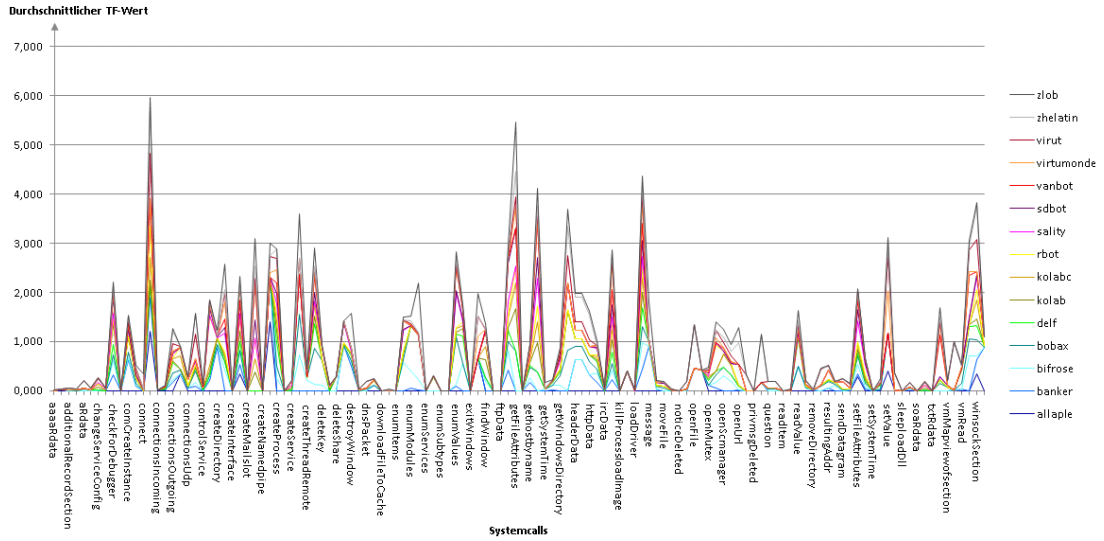



Diagramm 5.12.: Durchschnittliche $TF$–Werte jedes Systemcalls pro Familie im Lerndatensatz. Eine vergrößerte Darstellung dieses Diagramms ist in Anhang B zu finden. Aus Darstellungsgründen werden auf der x–Achse nicht alle Beschriftungen angezeigt.

Zur Nachvollziehbarkeit eventueller Fehlklassifikationen aufgrund von Ähnlichkeiten der Systemcallvektoren unterschiedlicher Klassen zueinander wurde zudem ein Clusteringalgorithmus auf die Vektoren angewendet. Auch dieser lieferte keine aufschlussreichen Informationen.

Charakteristiken von Schadprogramm–Familien sind folglich aufgrund dieser einfachen Darstellungen auf Anhieb nicht zu ermitteln. Die folgenden Experimente werden jedoch zeigen, dass eine Klassifikation der Vektoren durch maschinelle Lernverfahren durchaus möglich ist.

## 5.5. Experimente

Nachdem in Abschnitt 5.1 die vorliegenden Daten beschrieben, in Abschnitt 5.2 deren Konvertierung in eine Textdarstellung durch die Extraktion der Threads dargestellt und in Abschnitt 5.3 die Transformation der Textdateien in eine Vektordarstellung durchgeführt wurden, werden in diesem Abschnitt Ansätze zur Klassifikation der

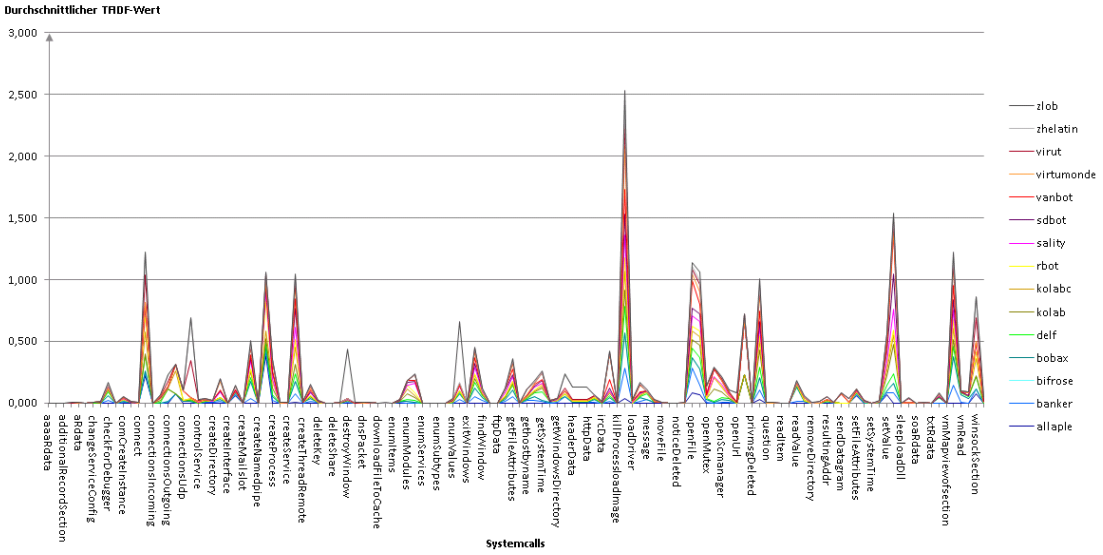



Diagramm 5.13.: Durchschnittliche $TFIDF$–Werte jedes Systemcalls pro Familie im Lerndatensatz. Eine vergrößerte Darstellung dieses Diagramms ist in Anhang B zu finden. Aus Darstellungsgründen werden auf der x–Achse nicht alle Beschriftungen angezeigt.

Schadprogramm–Familien präsentiert. Dazu werden verschiedene Lernverfahren, Vektordarstellungen der Beispiele und Merkmalsmengen bezüglich ihrer Eignung zur Lösung des Mehrklassenproblems bewertet.

Die Thread–Dateien sind gelabelt. Das heißt, dass die Schadprogramm–Familie eines jeden Beispiels bekannt ist. Daher werden *überwachte* Lernverfahren genutzt. Die verwendeten Verfahren und einige Implementierungsdetails wurden in Kapitel 4 erläutert und vielfach bereits in den verwandten Arbeiten aus Kapitel 3 eingesetzt.

In dem folgenden Abschnitt 5.5.1 wird zunächst die Lernbarkeit von Schadprogramm–Familien auf Systemcallvektoren mit der SVM bewertet. Anschließend wird dieses Verfahren in Abschnitt 5.5.2 auf Sequenzvektoren und Mengenvektoren angewendet. In Abschnitt 5.5.3 folgt die Evaluation einer Auswahl anderer Lernverfahren. Auf der Grundlage einiger der dort erstellten Modelle und der Anwendung verschiedener Gewichtungsverfahren wird in Abschnitt 5.5.4 die Bedeutung einzelner Systemcalls für die Klassifikation von Schadprogramm–Familien beurteilt. Die Ergebnisse werden in Abschnitt 5.5.5 teilweise dazu genutzt, eine Auswahl von Systemcalls zu bestimmen, deren Parameterwerte bei der Klassifikation der Malware berücksichtigt werden sollen. Schließlich präsentiert Abschnitt 5.5.6 einen Ansatz zur Verbesserung der Klassifikation, der zugleich die Basis für die Zuordnung eines Traces auf der Grundlage der klassifizierten Threads liefert.

**Notationen** Eine erzeugte Textdatei wird im Folgenden als Beispiel betrachtet und mit $d$ bezeichnet, der zugehörige Vektor wird mit $\vec{x}$ notiert und $\|\vec{x}\|$ beschreibt die euklidische Länge von $\vec{x}$. Die Menge aller Beispiele eines Datensatzes ist $\mathcal{X} = \{(\vec{x}_i, y_i) | 1 \leq i \leq n\}$ mit $y_i$ der Schadprogramm–Familie des Beispiels $\vec{x}_i$. Die geordnete Merkmalsmenge wird in

Anlehnung an [RHW+08] wie bereits zuvor mit $\mathcal{F} = \{s_j | 1 \leq j \leq m\}$ bezeichnet. Hierbei deklariert $s_j$ das $j$–te Merkmal der geordneten Menge und kann je nach Darstellungsform einen Systemcall mit oder ohne Parameternamen und/oder –werte, eine Systemcall–Menge oder eine Systemcall–Sequenz repräsentieren. Die Menge der Schadprogramm–Familien wird im Folgenden auch mit der Menge der "Klassen" oder "Label" bezeichnet und mit $\mathcal{Y}$ notiert. In dem Fall der verwendeten Lern– und Testdatensätze gilt also $|\mathcal{Y}| = 15$.

Die Mengen der richtig positiven Beispiele *TP* (True Positive) und der falsch positiven Beispiele *FP* (False Positive), die für jedes Experiment angegeben werden, wurden in Abschnitt 4.6 bereits bezüglich eines Zweiklassenproblems definiert und in den Gleichungen (4.39) und (4.40) für den Fall des hier vorliegenden Mehrklassenproblems formuliert. Gleiches gilt für die Definitionen 4.8 und 4.10 des *Recalls* und der *Accuracy*.

Die Accuracy der im Folgenden verwendeten Lernverfahren wird als Durchschnitt der Ergebnisse jedes einzelnen Lerndurchgangs bei der Anwendung des in Kapitel 4.6 beschriebenen Verfahrens der 10–fachen, stratifizierten Kreuzvalidierung berechnet.

### 5.5.1. Klassifikation von Systemcallvektoren mit der SVM

In dem vorliegenden Abschnitt erfolgt die Auswahl einer geeigneten Systemcallvektor–Darstellung der Logdateien. Es wird bewertet, welche der in Abschnitt 3.4 eingeführten und in Abschnitt 5.3.2 in Verbindung mit RapidMiner aufgegriffenen Komponentenberechnungen die beste Accuracy (siehe Definition 4.10) bei der Klassifikation mit der Stützvektormethode liefert. Dabei besteht die Merkmalsmenge aus einzelnen Systemcalls ohne Parameternamen und –werte.

Die in Kapitel 4.4 beschriebene Stützvektormethode passt eine Hyperebene in den Merkmalsraum ein, die die in einer Vektordarstellung gegebenen Beispiele bestmöglich voneinander trennt. Das Verfahren wird aufgrund seiner Generalisierungsfähigkeit in vielen Bereichen des Data Minings und ebenso zur Analyse von Systemcall–Berichten häufig eingesetzt. In [YZF06] (Abschnitt 3.4.2) wird die SVM zur Erkennung von Anomalien verwendet. Dabei wird vorab eine Merkmalsgewichtung durchgeführt. In [KFH05] (siehe Abschnitt 3.5.1) wird neben der Güte anderer Lernverfahren auch die Güte der SVM bei der Erkennung von Angriffen auf der Basis von Systemcall–Mengenvektoren evaluiert und liefert eine perfekte Klassifikation. In Abschnitt 3.7 wurden die Ansätze aus [KAT07] und [RHW+08] zur Klassifikation von Schadprogramm–Familien durch ein SVM–Modell beschrieben. Auch hier waren die Ergebnisse mit 69,80% und 88% Genauigkeit zufriedenstellend. Neben diesen Arbeiten gibt es eine Vielzahl weiterer Ansätze, die die SVM im Bereich der Intrusion Detection nutzen. Allerdings wurde von der detaillierten Vorstellung anderer Ansätze im Rahmen dieser Diplomarbeit abgesehen und es wurden nur einige grundlegende Arbeiten beschrieben.

**SVM–Parameter** Die in den folgenden Experimenten verwendeten Parameterwerte der SVM wurden mit Hilfe einer Parameteroptimierung bestimmt. Aufgrund dieser Optimierung wurde ein RBF–Kern (Gleichung (4.15)) mit $\gamma = 0,008$ genutzt. Der Parameter $C$, der den Einfluss des Trainingsfehlers bestimmt, wurde auf 120 gesetzt. Die Toleranz des Terminierungskriteriums beträgt 0,176.

**Auswertung der Ergebnisse** Tabelle 5.1 zeigt die Ergebnisse der SVM bei der Anwendung auf Systemcallvektoren der Threads, die sich durch die Berechnung ihrer Komponentenwerte unterscheiden. Die Genauigkeit der Vorhersage ist bei den $TF$–Vektoren am höchsten. Wie in Tabelle 5.2 zu erkennen, ist diese Darstellung der Vektoren zudem die

| Komponentenberechnung | Accuracy (%) |
|---|---|
| TFIDF $= TFIDF(s_j, d)$ | 83,29 ± 0,41 |
| Term Frequency $= \frac{TF(s_j,d)}{f_d}$ | 83,20 ± 0,26 |
| Term Ocuurence $= TF(s_j, d)$ | **87,20 ± 0,27** |
| Binary Occurrences $= BO(s_j, d)$ | 86,28 ± 0,33 |

Tabelle 5.1.: Accuracy der SVM bei unterschiedlicher Berechnung der Vektorkomponenten. Die Vektordimensionen entsprechen dabei einzelnen Systemcalls ohne Parameternamen und –werte. Die Accuracy–Werte wurden als Durchschnitt der Werte aller Klassen bei 10–facher Kreuzvalidierung ermittelt. Das beste Ergebnis ist fett markiert.

einzige, bei der der Class Recall der SVM (auch True Positive Rate, siehe Abschnitt 4.6) bei allen Schadprogramm–Familien größer als 0 ist. Bei dem Lernen auf $BO$–Vektoren ist die Accuracy zwar ebenfalls gut, der Class Recall beträgt bei 2 Schadprogramm–Familien jedoch 0. Bei der Darstellung als $TFIDF$–Systemcallvektoren sind es bereits 6 und bei der Verwendung von TermFrequency–Vektoren sogar 7 Schadprogramm–Familien mit einem Recall von 0.

Die besseren Trefferquoten einiger Familien wenn *nicht* die $TF$–Darstellung gewählt wird, liegen mit Ausnahme des Virus Sality unter 3,5% über dem Wert der $TF$–Darstellung. Die Trefferquote der Familie Sality liegt bei der Berechnung der Vektorkomponenten durch $TFIDF$–Werte um 16% über der Trefferquote bei der Verwendung von $TF$–Vektoren.

**Zusammenfassung der Ergebnisse** Es kann festgehalten werden, dass die SVM bei der Klassifikation auf der Grundlage von $TF$–Systemcallvektoren einzelner Threads mit 87,20% ± 0,27% die beste Accuracy liefert. Die relativ niedrige Accuracy–Varianz lässt auf die Robustheit das Modells schließen.

Bei der Mehrheit der Schadprogramm–Familien liefert die $TF$–Darstellung auch den besten Class Recall. Ein Vorteil der Verwendung von $TF$–Vektoren gegenüber der Verwendung von $TFIDF$–Vektoren besteht zudem darin, dass die Komponentenwerte der Vektoren nicht von den übrigen Beispielen abhängen. So kann ein neues Trainingsbeispiel zur Lernmenge hinzugefügt werden, ohne die Gewichte der existierenden Beispiele ändern zu müssen.

Die bei den vorgestellten Experimenten auf $TF$–Vektoren erzielte Genauigkeit entspricht ungefähr den Ergebnissen aus [RHW+08] (88% Accuracy). Sie liegt über dem Ergebnis aus [KAT07], obwohl dort lediglich ein 5–Klassenproblem gelöst wurde.

| Komponenten-Berechnung | Class Recall pro Schadprogramm–Familie (%) | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ALLAPLE | BANKER | BIFROSE | BOBAX | DELF | KOLAB | KOLABC | RBOT | SALITY | SDBOT | VANBOT | VIRTUMONDE | VIRUT | ZHELATIN | ZLOB |
| TFIDF | 98,45 | 0,00 | 0,00 | 13,36 | **24,77** | 0,00 | 0,24 | 0,09 | **42,58** | 0,00 | 0,00 | 0,00 | 77,41 | 2,65 | 68,61 |
| TermFrequency | 98,51 | 0,00 | 0,00 | 9,63 | 6,32 | 0,00 | 0,96 | 0,22 | 15,12 | 0,00 | 0,00 | 0,00 | 83,30 | 0,00 | **72,94** |
| TF | **99,36** | **16,20** | **6,15** | **49,75** | 24,20 | **11,14** | **9,88** | 16,05 | 29,58 | **4,52** | **5,32** | **16,7** | 81,73 | **17,93** | 69,51 |
| BO | 99,11 | 3,73 | 1,54 | 40,93 | 11,07 | 0,48 | 0,00 | **17,00** | 24,74 | 1,33 | 0,00 | 8,39 | **84,19** | 3,35 | 71,00 |

Tabelle 5.2.: Class Recall der SVM auf den Threads des Lerndatensatzes bei der Verwendung unterschiedlicher Komponentenberechnungen der Systemcallvektoren. Der beste Wert pro Klasse ist fett markiert.

### 5.5.2. Klassifikation von Sequenz– und Mengenvektoren mit der SVM

In diesem Abschnitt wird evaluiert, ob sich die Klassifikationsergebnisse der SVM für das vorliegende Problem verbessern wenn Sequenzen und Mengen als Merkmale genutzt werden und welche Sequenzlängen und Mengengrößen sich eignen.

Wie in Abschnitt 3.3.1 beschrieben, haben Forrest et. al. in [FHSL96] zur Unterscheidung maliziöser und normaler Programmausführungen *Systemcall–Sequenzen* der Längen 5,6 und 11 verwendet. Ein ähnlicher Ansatz wurde in [LS98] (Abschnitt 3.3.2) verfolgt. In beiden Arbeiten konnten auf dieser Datengrundlage gute Ergebnisse erzielt werden. Aufbauend auf der Idee in [KFH05], die bereits in Abschnitt 3.5.1 beschrieben wurde, wird zudem die Lernbarkeit auf *Systemcall–Mengen* betrachtet. Durch Mengen statt Sequenzen kann die Anzahl der Merkmale reduziert und der Aufrufzusammenhang dennoch berücksichtigt werden.

Die Sequenzen und die Mengen werden mit Hilfe des in Abschnitt 5.3.2 beschriebenen, angepassten *TermNGramGenerator*–Operators in RapidMiner extrahiert und als Merkmale zur Erzeugung einer Vektordarstellung genutzt. Die Mengen werden also nicht wie in [KFH05] in der Form von Systemcallvektoren über alle im Datensatz vorhandenen Systemcalls dargestellt, sondern dienen stattdessen wie die Sequenzen direkt als Merkmalsmenge zur Erzeugung einer Vektordarstellung.

Aufgrund der Ergebnisse des vorigen Abschnitts 5.5.1 werden bei den folgenden Experimenten $TF$–Merkmalsvektoren genutzt. Jede Komponente des Vektors gibt folglich die Häufigkeit einer Sequenz oder einer Menge in dem vorliegenden Thread an.

**Überblick über die Sequenz– und Mengenvektoren des Lerndatensatzes** Tabelle 5.3 gibt einen ersten Überblick über die als $TF$–Vektoren verschiedener Merkmalsmengen vorliegenden Trainingsdaten. Neben der Anzahl der Merkmale $|\mathcal{F}|$ und der maximalen euklidischen Länge eines Beispiels ($\max ||\vec{x}||$) wird auch die maximale Anzahl der Merkmale angegeben, die in einem Beispiel vorhanden sind, also

$$\max_{\vec{x}_i \in \mathcal{X}} \sum_{j=1}^{|\mathcal{F}|} [\![s_{ji}]\!] \quad \text{mit } [\![s_{ji}]\!] = \begin{cases} 1 & \text{wenn der Wert von } s_j \text{ in dem Beispiel } \vec{x}_i > 0 \text{ ist und} \\ 0 & \text{sonst.} \end{cases}$$

Zu Vergleichszwecken werden die Angaben auch für den Fall der Verwendung einzelner Systemcalls als Merkmalsmenge aufgeführt.

Wie zu erkennen ist, steigt die Anzahl unterschiedlicher Merkmale $m = |\mathcal{F}|$ mit jeder Mengenvergrößerung und jeder Sequenzverlängerung stark an. Die Beispielvektoren sind offensichtlich sehr spärlich besetzt. Je länger die Sequenz beziehungsweise je größer die Menge ist, die ein Merkmal bildet, desto seltener kommen die Merkmale in einem Thread vor. Bei einer Merkmalsmenge aus 6er–Sequenzen sind gerade noch maximal 1,75% des Vektors mit Werten $> 0$ belegt, bei 6er–Mengen noch maximal 1,60%. Die maximale euklidische Länge der Mengenvektoren scheint bei Mengen mit einer geraden Anzahl an Elementen (2,4,6) etwas höher zu sein als bei den übrigen Merkmalsmengen. Diese Mengen scheinen demnach häufiger in einem Thread enthalten zu sein. Darüber hinaus weist die euklidische Länge jedoch keine signifikanten Werte bezüglich einer bestimmten Darstellung auf.

| Merkmale | $\lvert\mathcal{F}\rvert$ | $\max \lvert\lvert\vec{x}\rvert\rvert$ | $\max_{\vec{x}_i \in \mathcal{X}} \sum_{j=1}^{\lvert\mathcal{F}\rvert} [\![s_{ji}]\!]$ |
|---|---|---|---|
| einzelne Systemcalls | 120 | 67.612,38 | 49 |
| 2er-Sequenzen | 1.705 | 58.908,58 | 140 |
| 3er-Sequenzen | 7.612 | 58.032,74 | 277 |
| 4er-Sequenzen | 15.635 | 57.175,48 | 370 |
| 5er-Sequenzen | 23.387 | 56.322,07 | 457 |
| 6er-Sequenzen | 30.453 | 55.488,48 | 532 |
| 2er-Menge | 1.216 | 83.040,65 | 112 |
| 3er-Menge | 4.757 | 58.549,06 | 198 |
| 4er-Menge | 9.984 | 80.626,75 | 254 |
| 5er-Menge | 15.728 | 56.338,26 | 313 |
| 6er-Menge | 21.279 | 78.443,50 | 349 |

Tabelle 5.3.: Anzahl der Merkmale $\lvert\mathcal{F}\rvert$, maximale euklidische Länge $\max \lvert\lvert\vec{x}\rvert\rvert$ gerundet auf 2 Dezimalstellen und maximale Anzahl der in einem Thread $\vec{x}_i$ des Lerndatensatzes $\mathcal{X}$ vorkommenden Merkmale $s_j \in \mathcal{F}$ bei der Verwendung von Sequenzen und Mengen.

**Auswertung der Ergebnisse der Lernphase** Die SVM–Parameter entsprechen den Werten, die bei den Experimenten aus Abschnitt 5.5.1 genutzt wurden. Tabelle 5.4 zeigt die Ergebnisse bei der Anwendung des Verfahrens auf die $TF$–Vektoren aus Sequenzen und Mengen der Thread–Dateien des Lerndatensatzes. Die Accuracy wird erneut als Durchschnitt bei 10–facher, stratifizierter Kreuzvalidierung berechnet. Neben der Accuracy wird die Anzahl der richtig und der falsch klassifizierten Beispiele angegeben, die relative Änderung der Laufzeit bezüglich der Laufzeit des Verfahrens auf einzelnen Systemcalls, die Anzahl der Merkmale $\lvert\mathcal{F}\rvert$ bei der jeweiligen Darstellung und der relative Zuwachs an Stützvektoren (SV) verglichen mit der Anzahl an Stützvektoren bei der Anwendung des Verfahrens auf einzelne Systemcalls.

Die Berücksichtigung der euklidischen Länge eines Vektors und der Anzahl vorhandener Merkmale als zwei zusätzliche Merkmale eines jeden Beispiels ergab keine Verbesserung des Lernergebnisses.

Die Genauigkeit des Modells erhöht sich bei der Verwendung von 2er–Sequenzen gegenüber der Betrachtung einzelner Systemcalls unter Berücksichtigung der Varianzen um mindestens 2,58%, die Laufzeit steigt um ca. 19%. Bei dem Lernen auf 2–elementigen Mengen lässt sich ebenfalls eine Verbesserung der Accuracy von mindestens 2,22% gegenüber dem Ergebnis des Verfahrens bei der Betrachtung einzelner Systemcalls erzielen. Bei dieser Darstellung der Beispiele *verringert* sich die Laufzeit sogar um ca. 8%, obwohl sich die Anzahl der Merkmale $\lvert\mathcal{F}\rvert$ ungefähr verzehnfacht. Die Laufzeit des Lernverfahrens bei der Nutzung von 3er–Mengen entspricht ungefähr der Laufzeit bei der Nutzung von $TF$–Vektoren einzelner Systemcalls und liefert eine um mindestens 2,66% höhere Accuracy.

Bei einer weiteren Fenstervergrößerung zur Erstellung der Sequenzen und Mengen wird

| Merkmale | Ø Accuracy (%) | TP | FP | rel. Laufzeit-Änderung (%) | $\|\mathcal{F}\|$ | rel. Zuwachs SV |
|---|---|---|---|---|---|---|
| einzelne Systemcalls | 87,20 ± 0,27 | 112.563 | 16.525 | - | 120 | - |
| 2er-Sequenzen | 90,35 ± 0,30 | 116.636 | 12.452 | +18,97 | 1.705 | · 7,8 |
| 3er-Sequenzen | 90,51 ± 0,30 | 116.834 | 12.254 | +40,49 | 7.612 | · 7,5 |
| 4er-Sequenzen | 90,76 ± 0,26 | 117.109 | 11.979 | +72,77 | 15.635 | · 7,6 |
| 5er-Sequenzen | 90,91 ± 0,33 | 117.555 | 11.533 | +108,36 | 23.387 | · 7,6 |
| 6er-Sequenzen | 90,92 ± 0,24 | 117.362 | 11.726 | +127,78 | 30.453 | · 7,6 |
| 2er-Menge | 89,89 ± 0,20 | 116.043 | 13.045 | -7,78 | 1.216 | · 7,9 |
| 3er-Menge | 90,47 ± 0,34 | 116.792 | 12.296 | -0,72 | 4.757 | · 7,5 |
| 4er-Menge | 90,74 ± 0,25 | 117.133 | 11.955 | +17,87 | 9.984 | · 7,6 |
| 5er-Menge | 90,79 ± 0,24 | 117.196 | 11.892 | +108,36 | 15.728 | · 7,6 |
| 6er-Menge | 90,89 ± 0,27 | 117.331 | 11.757 | +100,74 | 21.279 | · 7,6 |

Tabelle 5.4.: Accuracy der SVM auf den Threads des Lerndatensatzes, dargestellt durch $TF$–Vektoren aus Sequenzen und Mengen. Berechnung der Accuracy als Durchschnitt der Werte aller Klassen bei 10–facher Kreuzvalidierung. Angabe der Anzahl an Merkmalen $|\mathcal{F}|$ bei den entsprechenden Darstellungen, der relativen Laufzeitänderung und dem relativen Zuwachs an Stützvektoren (SV) bezüglich des Lernverfahrens auf $TF$–Vektoren einzelner Systemcalls.

die Accuracy nur noch minimal besser, die Laufzeit verlängert sich hingegen beträchtlich. Dieser Trend ist bei den Ergebnissen aus Tabelle 5.4 deutlich zu erkennen und in Abbildung 5.14 noch einmal graphisch dargestellt. Dabei bezeichnen die Zahlen neben den Datenpunkten in der Abbildung jeweils die Länge der Sequenzen beziehungsweise die Größe der Mengen, die zu den entsprechenden Ergebnissen geführt haben.

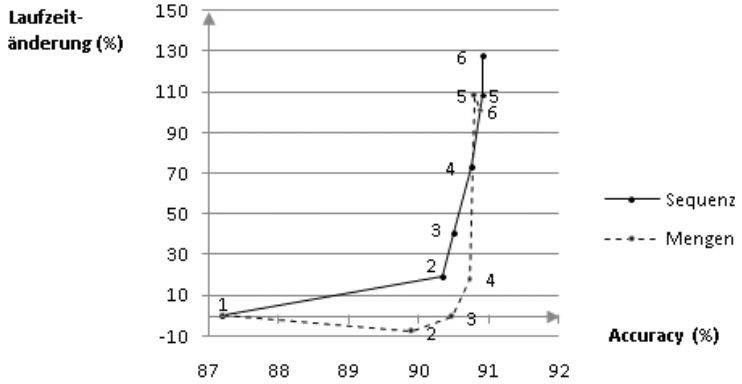



Abbildung 5.14.: Verhältnis der Accuracy zur Laufzeit der SVM bei unterschiedlicher Länge der Sequenzen und Größe der Mengen. Die Zahlen neben den Datenpunkten geben die Länge der Sequenzen beziehungsweise die Anzahl der Elemente der Mengen an, die zu den entsprechenden Ergebnissen geführt haben.

Verglichen mit der Anwendung des Lernverfahrens auf Vektoren einzelner Systemcalls werden bei allen Mengen– und Sequenzdarstellungen 7,5 bis 7,9 Mal so viele Stützvek-

toren zur Trennung der Klassen herangezogen. Da mit den Stützvektoren die Beispiele bezeichnet werden, die der Hyperebene am nächsten sind, scheinen folglich viele Beispiele an den Klassengrenzen zu liegen. Ein Anstieg der Anzahl an Stützvektoren aufgrund längerer Sequenzen oder größerer Mengen liegt *nicht* vor.

Der Class Recall der SVM–Modelle ist in Tabelle 5.5 dargestellt. Die Werte variieren je nach Schadprogramm–Familie und je nach Merkmalsmenge teilweise stark. Der Wurm *Allaple* und der Virus *Virut* weisen für alle Sequenzen, Mengen und einzelne Systemcalls einen hohen Class Recall von durchschnittlich 99,68% und 86,55% auf. Dies sind die Familien, für die im Lerndatensatz die meisten Beispiele vorliegen (siehe Abbildung 5.10). Trotz sehr wenig vorhandener Threads des Trojaners *Zlob* liegt die Trefferquote bei dieser Malware für alle Merkmalsmengen noch bei durchschnittlich 73,56%.

Die Trefferquoten der Schadprogramm–Familien *Kolabc* und *Virtumonde* sind stark abhängig von der Größe der verwendeten Mengen bzw. der Länge der Sequenzen. Beide Familien werden bei 5er–Mengen am besten erkannt. Sie scheinen bei dieser Darstellung folglich charakteristischere Strukturen aufzuweisen. Die Familien *Zhelatin*, *Sdbot* und *Vanbot* und werden durch die Verwendung von Sequenzen und Mengen nicht viel besser erkannt als bei der Betrachtung einzelner Systemcalls.

Alle übrigen Schadprogramm–Familien liefern bei den Sequenz– und Mengenvektoren vergleichbare Werte, die bis auf einige wenige Ausnahmen über dem Class Recall der SVM bei der Darstellung der Threads als Vektoren einzelner Systemcalls liegen. Die Differenz ist dabei in einigen Fällen erheblich. Die SVM liefert zum Beispiel für den Wurm *Rbot* beim Lernen auf einzelnen Systemcalls lediglich einen Class Recall von 16,05%, während der durchschnittliche Recall beim Lernen auf Sequenzen und Mengen bei 70,02% liegt. Ähnlich stark ist der Unterschied bei dem Trojaner *Virtumonde*, bei dem der Class Recall beim Lernen auf der Basis einzelner Systemcalls 16,78% gegenüber dem Durchschnitt von 50,10% bei Sequenzen und Mengen beträgt.

Auf der Basis von Sequenzvektoren und Mengenvektoren können demnach genauere Modelle erstellt werden als auf der Basis von Systemcallvektoren. Leider geben die Ergebnisse keinen eindeutigen Hinweis auf eine zur Erkennung aller vorliegenden Schadprogramme zu bevorzugende Fenstergröße. Auch innerhalb einer Schadprogramm–Familie hängt die Erkennungsrate größtenteils nur wenig von der Größe der Mengen beziehungsweise der Länge der Sequenzen ab. Es gibt lediglich einige Tendenzen. Die Familie *Banker* wird zum Beispiel besser erkannt, wenn *Mengen* verwendet werden, wohingegen die Familie *Rbot* bei *Sequenzen* besser erkannt wird.

**Auswertung der Ergebnisse der Testphase** In Tabelle 5.6 sind die Ergebnisse bei der Anwendung der gelernten Modelle auf die neuen, bis dahin ungesehenen Daten der Testmenge dargestellt. Sie bestätigen die Überlegenheit der auf Sequenz– und Mengenvektoren gelernten Modelle gegenüber dem auf Systemcallvektoren gelernten Modell. Die Genauigkeit ist um mindestens 4,31% höher. Die Accuracy nimmt mit der Länge der Sequenzen beziehungsweise der Anzahl der Elemente der Mengen zunächst zu. Beim Übergang von 3er–Mengen zu 4er–Mengen verbessert sich die Accuracy mit 1,84% noch verhältnismäßig stark. Bei der Verwendung von Sequenzen der Länge 6 und Mengen der Größe 5 nimmt sie dann wieder ein wenig ab. Die relative Zunahme der Laufzeiten bei der Vergrößerung der Mengen und der Verlängerung der Sequenzen ist leider enorm.

| Merkmale | Class–Recall pro Schadprogramm–Familie (%) | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ALLAPLE | BANKER | BIFROSE | BOBAX | DELF | KOLAB | KOLABC | RBOT | SALITY | SDBOT | VANBOT | VIRTUMONDE | VIRUT | ZHELATIN | ZLOB |
| einzelne Systemcalls | 99,36 | 16,20 | 6,15 | 49,75 | 24,20 | 11,14 | 9,88 | 16,05 | 29,58 | 4,52 | 5,32 | 16,78 | 81,73 | 17,93 | 69,51 |
| 2er-Sequenzen | 99,72 | 32,17 | 12,31 | 57,70 | 29,21 | 13,32 | 10,60 | 68,05 | 40,80 | 3,19 | 6,08 | 37,58 | 86,93 | 22,84 | 71,15 |
| 3er-Sequenzen | 99,72 | 31,35 | 14,62 | 55,83 | 27,58 | 13,32 | 13,73 | 71,01 | 46,90 | 5,32 | 5,70 | 58,39 | 87,21 | 22,37 | 74,14 |
| 4er-Sequenzen | **99,77** | 31,82 | 13,85 | 56,06 | 29,96 | 14,53 | 27,95 | 71,04 | **55,40** | 4,79 | 5,70 | 57,05 | 87,09 | 22,84 | 74,29 |
| 5er-Sequenzen | 99,76 | 31,70 | 16,15 | 58,33 | 30,89 | 14,53 | 25,06 | **71,64** | 52,77 | 7,71 | 6,08 | 56,38 | 87,14 | **25,95** | 74,29 |
| 6er-Sequenzen | **99,77** | 32,17 | 15,38 | 57,10 | **32,08** | **15,74** | 30,36 | 71,61 | 53,52 | 6,65 | 4,56 | 58,72 | **87,36** | 25,02 | 75,64 |
| 2er-Menge | 99,58 | 22,61 | 13,85 | 52,75 | 30,58 | 10,65 | 12,77 | 66,35 | 44,46 | **10,11** | **9,89** | 39,60 | 86,29 | 23,54 | 71,90 |
| 3er-Menge | 99,72 | 34,15 | 13,85 | 55,86 | 31,14 | 14,53 | 10,84 | 70,19 | 45,87 | 4,79 | **9,89** | 36,91 | 87,14 | 22,37 | 73,84 |
| 4er-Menge | 99,72 | 34,15 | 13,85 | 55,86 | 31,14 | 14,53 | 10,84 | 70,19 | 45,87 | 4,79 | **9,89** | 36,91 | 87,14 | 22,37 | 73,84 |
| 5er-Menge | 99,72 | 34,50 | 12,31 | 56,30 | 31,77 | 15,01 | **31,81** | 69,69 | 55,31 | 6,12 | 8,37 | **62,42** | 87,08 | 23,62 | 74,44 |
| 6er-Menge | 99,72 | **34,85** | **17,69** | **59,49** | 31,77 | 15,25 | 29,40 | 70,38 | 50,61 | 5,32 | 6,46 | 57,05 | 86,96 | 24,08 | **76,08** |

Tabelle 5.5.: Class Recall der SVM auf den Threads des Lerndatensatzes, dargestellt durch $TF$–Vektoren erstellt auf der Grundlage von Merkmalsmengen aus Sequenzen und Mengen. Der jeweils beste Wert pro Schadprogramm–Familie ist fett markiert.

| Modell | Accuracy (%) | TP | FP | rel. Laufzeit-Änderung (%) |
|---|---|---|---|---|
| einzelne Systemcalls | 82,02 | 3.215 | 705 | - |
| 2er-Sequenzen | 86,40 | 3.387 | 533 | +24,58 |
| 3er-Sequenzen | 87,27 | 3.421 | 499 | +185,59 |
| 4er-Sequenzen | 88,57 | 3.472 | 448 | +514,83 |
| 5er-Sequenzen | 88,65 | 3.475 | 445 | +980,93 |
| 6er-Sequenzen | **88,60** | 3,473 | 447 | +1.577,97 |
| 2er-Menge | 86,33 | 3.384 | 536 | +21,61 |
| 3er-Menge | 86,86 | 3.405 | 515 | +95,76 |
| 4er-Menge | 88,70 | 3.477 | 443 | +302,54 |
| 5er-Menge | **88,60** | 3.473 | 447 | +537,29 |
| 6er-Menge | 88,90 | 3.485 | 435 | +815,68 |

Tabelle 5.6.: Ergebnisse der SVM–Klassifikation der Threads des Testdatensatzes durch die gelernten Modelle. *Abnehmende* Accuracy bei steigender Länge der Sequenzen und Größe der Mengen ist fett markiert.

Der Class Recall der gelernten Modelle auf den Testdaten ist in Tabelle 5.7 dargestellt. Die Werte variieren auch hier je nach Schadprogramm–Familie stark. *Allaple* und *Virut* weisen wie schon in Tabelle 5.5 für alle Sequenzen, Mengen und einzelne Systemcalls einen recht hohen Class Recall auf. Die Trefferquote des Wurms *Kolabc* ist auch hier wieder stark abhängig von der Größe der verwendeten Mengen beziehungsweise der Länge der Sequenzen. Die Ergebnisse der Familie *Zhelatin* liegen bezüglich der Sequenz– und Mengenvektoren eng beieinander und sind dabei wesentlich besser als bei den Systemcallvektoren.

Der Class Recall der Schadprogramm–Familien *Bifrose* und *Vanbot* beträgt bei allen Merkmalsmengen 0. Mit den Beispielen der Familie *Sdbot* verhält es sich ähnlich. Dort wird lediglich beim Lernen auf 2er–Mengen ein einziger von insgesamt 26 Threads erkannt. Die Recall–Werte dieser Familien waren bereits in der Lernphase niedrig.

Die Werte der Familien *Delf*, *Kolab* und *Virtumonde* entsprechen in etwa den Werten in der Lernphase. Der Recall der Familie *Rbot* ist bei der Klassifikation der Testdaten durch das auf Vektoren einzelner Systemcalls erstellte Modell gegenüber der Lernphase um rund 30% gestiegen. Der Class Recall der Familien *Kolabc* und *Zhelatin* liegt in einigen Fällen ebenfalls erheblich *über* dem Class Recall dieser Schadprogramm–Familien in der Lernphase.

**Zusammenfassung der Ergebnisse** Zusammenfassend kann aufgrund der durchgeführten Experimente festgehalten werden, dass die Genauigkeit des SVM–Modells bei der Verwendung von $TF$–Vektoren der Merkmalsmengen aus Systemcall–Sequenzen und Systemcall–Mengen steigt. Dabei scheinen Mengenvektoren besser geeignet zu sein. Sie liefern in den meisten Fällen verglichen mit den Sequenzen ähnliche oder bessere Ergebnisse bezüglich der Accuracy und des Class Recalls und die SVM benötigt weniger Zeit zur Erstellung dieses Modells.

| Modell | Class–Recall pro Schadprogramm–Familie (%) | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ALLAPLE | BANKER | BIFROSE | BOBAX | DELF | KOLAB | KOLABC | RBOT | SALITY | SDBOT | VANBOT | VIRTUMONDE | VIRUT | ZHELATIN | ZLOB |
| einzelne Systemcalls | 99,42 | 5,88 | 0,00 | 20,24 | 21,43 | 9,43 | 8,86 | 46,43 | 17,86 | 0,00 | 0,00 | 13,33 | 87,97 | 41,43 | 13,33 |
| 2er-Sequenzen | **99,73** | 17,65 | 0,00 | 30,36 | 21,43 | 20,75 | 5,06 | 53,57 | 33,04 | 0,00 | 0,00 | 46,67 | 91,72 | **85,71** | 26,67 |
| 3er-Sequenzen | **99,73** | **21,57** | 0,00 | 39,29 | 32,14 | 11,32 | 8,86 | 60,71 | 35,71 | 0,00 | 0,00 | 73,33 | 92,90 | 85,24 | 40,00 |
| 4er-Sequenzen | **99,73** | 19,61 | 0,00 | 37,50 | 28,57 | 13,21 | **74,68** | 60,71 | 41,96 | 0,00 | 0,00 | 53,33 | 92,50 | 85,24 | 46,67 |
| 5er-Sequenzen | **99,73** | 19,61 | 0,00 | 29,76 | 28,57 | 13,21 | 68,35 | 64,29 | 56,25 | 0,00 | 0,00 | **60,00** | **93,10** | 85,24 | 46,67 |
| 6er-Sequenzen | **99,73** | 19,61 | 0,00 | 30,95 | **35,71** | 13,21 | 68,35 | **71,43** | 50,00 | 0,00 | 0,00 | **60,00** | 92,90 | 85,24 | 46,67 |
| 2er-Menge | 99,58 | 17,65 | 0,00 | 31,55 | 21,43 | 9,43 | 5,06 | 50,00 | 35,71 | **3,85** | 0,00 | 46,67 | 91,72 | 85,24 | 46,67 |
| 3er-Menge | **99,73** | **21,57** | 0,00 | 29,17 | 28,57 | 11,32 | 6,33 | 53,57 | 40,18 | 0,00 | 0,00 | 53,33 | 92,90 | 85,24 | **66,67** |
| 4er-Menge | **99,73** | **21,57** | 0,00 | 31,55 | 28,57 | 13,21 | 69,62 | 57,14 | **57,14** | 0,00 | 0,00 | **60,00** | 92,70 | 85,24 | 46,67 |
| 5er-Menge | **99,73** | 19,61 | 0,00 | 29,76 | 32,14 | 16,98 | **74,68** | 57,14 | 49,11 | 0,00 | 0,00 | **60,00** | **93,10** | 85,24 | 46,67 |
| 6er-Menge | **99,73** | **21,57** | 0,00 | **39,88** | 28,57 | **28,30** | 70,89 | 64,29 | 41,96 | 0,00 | 0,00 | **60,00** | 92,70 | 85,24 | 46,67 |

Tabelle 5.7.: Class Recall der SVM bei der Klassifikation der Threads des Testdatensatzes durch die gelernten Modelle. Die Threads werden durch $TF$–Vektoren der Merkmalsmengen aus Sequenzen und Mengen dargestellt. Der jeweils beste Wert pro Schadprogramm–Familie ist fett markiert.

Die Experimente geben leider keinen eindeutigen Hinweis auf eine bestimmte Sequenzlänge oder eine Mengengröße, die sich zur Erkennung *aller* Schadprogramm–Familien besonders gut eignet. Auch eine Abhängigkeit der *familienspezifischen* Trefferquoten von der Fenstergröße liegt in den meisten Fällen nicht vor.

Vergleichsweise gute Ergebnisse für alle Schadprogramm–Familien unter Berücksichtigung der Accuracy und der Laufzeit während der Lernphase und der Ergebnisse in der Testphase liefert die SVM bei der Klassifikation von $TF$–Vektoren einer Merkmalsmenge aus 3er–Mengen. Allerdings ist die Zunahme der Laufzeit bei der Anwendung des Modells in der Testphase auch hier enorm.

Durch die Verwendung von Mengen– und Sequenzvektoren kann eine höhere Modell–Genauigkeit erzielt werden als die in [RHW+08] angegebenen 88%. Dort wurde die SVM zur Klassifikation von Schadprogramm–Familien auf eine Merkmalsmenge unterschiedlicher *Detaillierungsgrade* der Systemcalls und ihrer Parameterwerte angewendet.

### 5.5.3. Bewertung weiterer Klassifikationsverfahren

Zur Bewertung der allgemeinen Lernbarkeit von Schadprogramm–Familien auf Systemcallvektoren werden in diesem Abschnitt verschiedene gängige Klassifikationsverfahren getestet und mit den Ergebnissen der SVM aus den vorangegangenen Abschnitten 5.5.2 und 5.5.1 verglichen. Die ausgewählten Verfahren können mit großen Merkmalsmengen gar nicht oder nur auf Kosten hoher Laufzeiten und hohen Platzbedarfs umgehen. Daher wurden lediglich Vektordarstellungen der Merkmalsmenge einzelner Systemcalls ohne ihre Parameternamen und –werte verwendet. Die Menge der in einem Betriebssystem zur Verfügung stehenden Systemcalls ist begrenzt und beläuft sich in dem vorliegenden Datensatz auf 120. Folglich ist auch die Anzahl der Merkmale auf 120 begrenzt. Sequenzvektoren und Mengenvektoren wurden also nicht betrachtet, da bei diesen Darstellungen wesentlich größere Merkmalsmengen entstehen (siehe Tabelle 5.3).

#### Bewertete Lernverfahren

Um das Feld der überwachten Lernverfahren gut abzudecken und gleichzeitig eine übersichtliche Auswertung zu gewährleisten, werden die verschiedenen in Kapitel 4 beschriebenen Verfahren stellvertretend für eine Vielzahl weiterer, möglicher Klassifikatoren betrachtet. Die Parameterwerte aller hier verwendeten Algorithmen wurden wieder mit Hilfe einer Parameteroptimierung festgelegt. Die Accuracy gibt den Durchschnittswert aller Klassen bei 10–facher, stratifizierter Kreuzvalidierung an.

**Regellerner** Aus der Kategorie der Regellerner wurde der in RapidMiner zur Verfügung stehende Operator genutzt, der das in Kapitel 4.2 beschriebene Verfahren von Kohavi [Koh95] implementiert. Zur übersichtlichen Darstellung des komplexen Regelwerks aus mehreren Bedingungen und der jeweils zugehörigen Klasse wird folglich eine Entscheidungstabelle aufgestellt. Obwohl der Algorithmus bereits eine interne Kreuzvalidierung vornimmt um die optimale Teilmenge $\mathcal{F}^* \subseteq \mathcal{F}$ der Systemcalls zur Aufstellung des Tabellenschemas zu finden, wurde das Verfahren mit einer zusätzlichen, äußeren, unabhängigen Kreuzvalidierung bewertet.

Ein Regellerner (RIPPER) wurde in [LS98] zur Anomalieerkennung auf Sequenzen angewendet (siehe Abschnitt 3.3.2) und in [KFH05] auf Mengen (siehe Abschnitt 3.5.1). Beide Varianten lieferten gute Ergebnisse.

**Instanzbasierte Klassifikation** Der aus der Kategorie der instanzbasierten Verfahren benutzte Algorithmus basiert auf der in Abschnitt 4.1 beschriebenen, speicherreduzierenden Variante des $k$NN–Klassifikators aus [AK91]. Dabei werden nicht mehr alle Beispiele der Lernmenge gespeichert. Als Ähnlichkeitsmaß dient die euklidische Distanz und das Verfahren wurde mit $k$=2 angewendet.

In [LV02] lieferte ein $k$NN–Algorithmus auf Systemcallvektoren mit $k$=10 und dem Cosinus–Maß als Distanzfunktion gute Ergebnisse bei der Lösung des Zweiklassenproblems der Anomalieerkennung. Die Ergebnisse aus [LV02] wurden in Abschnitt 3.4.1 erläutert.

**Entscheidungsbaum** Des Weiteren wird ein Entscheidungsbaum–Lerner verwendet, der die in Kapitel 4.3 beschriebene Erweiterung des C4.5–Algorithmus implementiert. Entscheidungsbäume können als alternative Darstellung eines Regelwerks angesehen werden. Die Merkmale eines Klassifizierungsproblems bilden die Knoten des Baumes, die Merkmalsausprägungen die Kanten und die Klassen die Blätter. Der hier verwendete Konfidenz–Schwellwert für das Pruning des erstellten Baumes beträgt 0,326 und es wurde festgelegt, dass ein Blatt des Baumes mindestens 2 Beispiele abdecken muss.

Zur Anomalieerkennung wurde der C4.5–Baumlerner zum Beispiel in [KFH05] (siehe Abschnitt 3.5.1) auf Systemcall–Mengen erfolgreich angewendet.

**Bayes–Klassifikation mit einem Graphen** In Abschnitt 4.5.2 wurde das Multinomiale Bayes–Modell beschrieben. Dieses Modell wurde in [KFH05] auf Systemcall–Mengen angewendet und lieferte im Vergleich zu den anderen getesteten Verfahren schlechte Ergebnisse bei der Erkennung von Angriffen. Auch bei dem vorliegenden Problem der Klassifikation von Schadprogramm–Familien auf Systemcallvektoren sind die Ergebnisse des Naive Bayes Multinomial Lerners verhältnismäßig schlecht. Die durchschnittliche Accuracy des Verfahrens über alle Klassen, evaluiert mit 10–facher Kreuzvalidierung beträgt lediglich 82,90% bei niedrigen Class Recall–Werten. Daher wird dieses Lernverfahren im Folgenden nicht weiter betrachtet.

Aus dem Bereich der Bayes–Klassifikatoren wurde stattdessen ein Graph–Lerner zur Erstellung eines Bayes–Netzes gewählt. Bayes–Netze können die Wahrscheinlichkeitsverteilungen von Merkmalen mittels eines Graphen kompakt repräsentieren. Das Verfahren wurde in Abschnitt 4.5.1 erläutert. Das zu erstellende Netz soll in dem vorliegenden Fall 25 Knoten und 56 Kanten enthalten. Die Variablenkardinalität wurde auf 42 festgelegt.

Ähnliche Bayes–Modelle lieferten in anderen Arbeiten gute Ergebnisse für verschiedene Probleme. In [Zan06] (siehe Abschnitt 3.6.1) wurde ein Markov Modell der Ordnung 1 verwendet, um den Ausführungskontext eines Systemcalls berücksichtigen zu können. Ein Markov Modell kann als die einfachste Form eines dynamischen Bayes–Netzes verstanden werden. In [ESL01] wurden Wahrscheinlichkeitsbäume erfolgreich zur Bestimmung der optimalen Sequenzlänge eingesetzt (siehe Abschnitt 3.3.3). In [MVKV06] wurde ein

Bayes–Netz zur Klassifikation eines Systemcalls durch die Kombination der Ausgaben verschiedener Parametermodelle genutzt (siehe Abschnitt 3.6.2).

### Auswahl der Vektordarstellung

Wie schon in Abschnitt 5.5.1 wird auch in diesem Abschnitt zunächst evaluiert, welche Berechnung der Vektorkomponenten die beste Accuracy liefert. Die Experimente werden wieder auf den Threads des Lerndatensatzes durchgeführt. Die Accuracy–Werte sind in Tabelle 5.8 dargestellt. Sie sind auch hier wieder bei der Verwendung von $TF$–

| Komponenten-berechnung | Accuracy (%) | | | |
|---|---|---|---|---|
| | Nearest–Neighbor | Entscheidungstabelle | Entscheidungsbaum | Graph (Bayes-Netz) |
| TFIDF $= TFIDF(s_j, d)$ | 92,06 ± 0,17 | 90,57 ± 0,17 | 92,15 ± 0,18 | **90,47 ± 0,27** |
| Term Frequency $= \frac{TF(s_j,d)}{f_d}$ | 92,07 ± 0,17 | 90,56 ± 0,18 | 92,13 ± 0,18 | 89,99 ± 0,23 |
| Term Ocuurence $= TF(s_j, d)$ | **92,76 ± 0,15** | **91,68 ± 0,23** | **92,98 ± 0,18** | 86,25 ± 0,26 |
| Binary Occurrences $= BO(s_j, d)$ | 91,06 ± 0,14 | 90,58 ± 0,11 | 91,05 ± 0,14 | 81,38 ± 0,17 |

Tabelle 5.8.: Accuracy verschiedener Lernverfahren auf den Systemcallvektoren der Threads des Lerndatensatzes bei unterschiedlicher Berechnung der Vektorkomponenten, ermittelt als Durchschnitt der Werte aller Klassen bei 10–facher Kreuzvalidierung. Das beste Ergebnis jedes Lernverfahrens ist fett markiert.

Vektoren der Threads am höchsten. Eine Ausnahme bildet der Graph–Lerner, der unter Berücksichtigung der Varianzen ein um mindestens 3,69% genaueres Modell erstellt, wenn $TFIDF$-Vektoren verwendet werden.

| Lernverfahren | Accuracy (%) | TP | FP |
|---|---|---|---|
| SVM (nur Systemcalls) | 87,20 ± 0,27 | 112.563 | 16.525 |
| SVM (6er-Sequenzen) | 90,92 ± 0,24 | 117.362 | 11.726 |
| Nearest–Neighbor | 92,76 ± 0,15 | 119.747 | 9.341 |
| Entscheidungstabelle | 91,68 ± 0,23 | 118.350 | 10.738 |
| Entscheidungsbaum | 92,98 ± 0,18 | 120.029 | 9.059 |
| Graph (Bayes) | 90,47 ± 0,27 | 116.792 | 12.296 |

Tabelle 5.9.: Darstellung der jeweils besten Ergebnisse der Verfahren bezüglich der Wahl der Komponentenberechnung auf den Threads des Lerndatensatzes. Angabe der Anzahl richtig klassifizierter Threads (TP) und falsch klassifizierter Threads (FP). Berechnung der Accuracy als Durchschnitt der Werte aller Klassen bei 10–facher Kreuzvalidierung.

### Ergebnisse

Im Folgenden wird eine detaillierte Auswertung der besten Ergebnisse jedes Lernverfahren aus Tabelle 5.8 vorgenommen. Da es sich dabei mit Ausnahme des Bayes–Modells um die Ergebnisse bei der $TF$–Darstellung der Vektoren handelt, werden die Ergebnisse dieser Darstellung für das Bayes–Modell zusätzlich angegeben. Die in Abschnitt 5.5.1 durch die SVM erzielten Werte auf den $TF$–Vektoren der Merkmalsmenge einzelner Systemcalls und der Merkmalsmenge aus 6er–Sequenzen werden zur Vergleichbarkeit ebenfalls aufgeführt.

**Auswertung der Ergebnisse der Lernphase** Die detaillierte Darstellung der Ergebnisse erfolgt in Tabelle 5.9. Die Accuracy–Werte der in dem vorliegenden Abschnitt vorgestellten Verfahren liegen eng beieinander. Die Modelle sind genauer als das SVM–Modell bei der Anwendung auf Systemcallvektoren. Die besten Klassifikationen bezüglich der Accuracy und der Varianz liefern das Nearest–Neighbor–Verfahren mit 92,76% ± 0,15% und der Entscheidungsbaum–Lerner mit 92,98% ± 0,18%.

Der Class Recall, dargestellt in Tabelle 5.10, variiert wie bei den Ergebnissen aus Kapitel 5.5.1 auch hier wieder je nach Schadprogramm–Familie stark. Der Wurm *Allaple* und der Virus *Virut* weisen ähnlich wie bei den SVM–Experimenten bei allen Verfahren einen recht hohen Class Recall von durchschnittlich 99,30% auf. Für diese Familien sind die meisten Beispiele vorhanden (siehe Abbildung 5.10). Auch der Trojaner *Zlob* wird trotz sehr weniger vorhandener Threads wieder von allen Verfahren gut erkannt. Der Recall liegt bei durchschnittlich etwa 76,50%.

Die in diesem Abschnitt getesteten Verfahren erkennen zudem alle die Schadprogramm–Familie *Virtumonde* wesentlich besser als die SVM. Während die SVM auf Systemcallvektoren für diese Familie lediglich einen Recall von 16,78% und auf 6er–Sequenzvektoren von 53,52% erreicht, liegt der durchschnittliche Wert der übrigen Verfahren bei über 80%. Das Nearest–Neighbor–Verfahren und der Entscheidungsbaum–Lerner weisen zudem bei der Erkennung der Familien *Banker* und *Delf* auffallend höhere Recall–Werte auf als die anderen Verfahren. Die Auswertung des Class Recalls untermauert folglich die bereits auf der Grundlage der Accuracy festgestellte Überlegenheit dieser beiden Verfahren. Die SVM liefert für viele Familien das schlechteste Ergebnis.

**Auswertung der Ergebnisse der Testphase** Die Ergebnisse der Klassifikation des Testdatensatzes durch die gelernten Modelle werden in Tabelle 5.11 dargestellt.

Das $TFIDF$–Modell des Graph–Lerners liefert hier im Vergleich zu dem $TF$–Modell eine um 6,76% schlechtere Genauigkeit. Der Grund dafür liegt sehr wahrscheinlich in der veränderten Dokumentenkollektion der Testmenge. Diese wird im Gegensatz zu der Komponentenberechnung durch $TF$–Werte bei der Berechnung der $TFIDF$–Werte durch die inverse Dokumentenfrequenz berücksichtigt (siehe Definitionen 3.4 und 3.5).

Das Nearest–Neighbor–Verfahren und der Entscheidungsbaum–Lerner erzielen auch hier wieder die besten Ergebnisse. Allerdings müssen die Trainingsbeispiele bei dem $k$NN–Verfahren mit allen zur Klassifikation in dem Modell gespeicherten Instanzen verglichen werden. Obwohl bereits die in Abschnitt 4.1 beschriebene speicherreduzierende Erweiterung des Verfahrens genutzt wurde, ist die Laufzeit bei der Anwendung des Modells

| Lernverfahren | Class–Recall pro Schadprogramm–Familie (%) | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ALLAPLE | BANKER | BIFROSE | BOBAX | DELF | KOLAB | KOLABC | RBOT | SALITY | SDBOT | VANBOT | VIRTUMONDE | VIRUT | ZHELATIN | ZLOB |
| SVM (nur Systemcalls) | 99,36 | 16,20 | 6,15 | 49,75 | 24,20 | 11,14 | 9,88 | 16,05 | 29,58 | 4,52 | 5,32 | 16,78 | 81,73 | 17,93 | 69,51 |
| SVM (6er-Sequenzen) | **99,77** | 32,17 | 15,38 | 57,10 | 32,08 | 15,74 | 30,36 | 71,61 | 53,52 | 6,65 | 4,56 | 58,72 | 87,36 | 25,02 | 75,64 |
| Nearest–Neighbor | 99,74 | 63,75 | 40,77 | **63,38** | **63,85** | **30,99** | **45,54** | 73,02 | 47,51 | 12,23 | 8,75 | 86,91 | 92,41 | 31,72 | 80,12 |
| Entscheidungstabelle | 99,73 | 32,05 | 20,77 | 57,78 | 47,84 | 26,15 | 37,11 | 71,64 | 43,57 | 18,88 | 4,56 | 84,56 | 91,34 | 29,85 | 74,89 |
| Entscheidungsbaum | 99,75 | **63,87** | 41,54 | 62,72 | 62,85 | 30,51 | 45,06 | 75,92 | 50,80 | **25,00** | 11,79 | **87,92** | **92,98** | **32,89** | **81,61** |
| Graph (Bayes) — TFIDF | 98,25 | 47,32 | 33,85 | 48,33 | 57,22 | 30,02 | 42,17 | **79,23** | **55,87** | 21,01 | 18,63 | 86,58 | 89,04 | 30,48 | 79,37 |
| Graph (Bayes) — TF | 98,59 | 37,18 | **53,85** | 37,57 | 18,82 | 23,49 | 16,63 | 37,55 | 24,32 | 17,55 | **19,01** | 67,45 | 77,22 | 24,55 | 74,74 |

Tabelle 5.10.: Class Recall der verschiedenen Lernverfahren auf den Threads des Lerndatensatzes. Wenn keine abweichenden Angaben gemacht werden, handelt es sich um die $TF$–Darstellung der Vektoren einzelner Systemcalls. Der jeweils beste Wert pro Schadprogramm–Familie ist fett markiert.

| Modell | Accuracy (%) | TP | FP |
|---|---|---|---|
| SVM (nur Systemcalls) | 82,02 | 3.215 | 705 |
| SVM (6er–Sequenzen) | 88,60 | 3.473 | 447 |
| Nearest–Neighbor | 90,74 | 3.557 | 363 |
| Entscheidungstabelle | 89,36 | 3.503 | 417 |
| Entscheidungsbaum | **91,33** | 3.581 | 339 |
| Graph (Bayes)— TFIDF | 77,76 | 3.048 | 872 |
| Graph (Bayes)— TF | 84,52 | 3.313 | 607 |

Tabelle 5.11.: Accuracy, TP und FP bei der Klassifizierung der Threads des Testdatensatzes durch die in der Lernphase erstellten Modelle mit der jeweils höchsten Genauigkeit. Bis auf den gekennzeichneten Fall des Graph–Lerners handelt es sich dabei um die Klassifikation von $TF$–Vektoren.

im Vergleich zu den anderen hier bewerteten Verfahren sehr hoch. Die Entscheidungstabelle kann die Beispiele am schnellsten einer Klasse zuordnen. Der Entscheidungsbaum benötigt das 3–fache dieser Zeit und das $k$NN–Modell etwa das 17–fache.

Der Class Recall der Verfahren wird in Tabelle 5.12 angegeben und variiert je nach Schadprogramm–Familie und Verfahren auch hier teilweise stark. *Allaple* und *Virut* weisen wieder bei allen Verfahren einen hohen Class Recall auf. Verglichen mit dem durch die SVM auf Systemcallvektoren erstellten Modell klassifizieren die alternativen Lernverfahren die Familie *Virtumonde*, wie bereits in der Lernphase, wesentlich besser. Das Nearest–Neighbor–Modell und das Entscheidungsbaum–Modell liefern bei der Erkennung neuer Threads der Familien *Banker*, *Delf*, *Kolabc* und *Zlob* wesentlich höhere Recall–Werte als alle anderen Verfahren. Die Familien *Sdbot* und *Vanbot* werden lediglich von dem Graph–Modell auf $TF$–Vektoren in wenigen Fällen erkannt. Das SVM–Modell ist mit Ausnahme der Erkennung neuer Threads der Schadprogramm–Familien *Allaple* und *Rbot* mindestens einem der anderen Verfahren stark unterlegen.

**Zusammenfassung der Ergebnisse** Die Ergebnisse dieses Abschnitts und der vorangehenden Passagen 5.5.1 und 5.5.2 werden in der Tabelle 5.13 noch einmal zusammengefasst.

Die SVM ist sowohl bei Zweiklassenproblemen als auch bei Mehrklassenproblemen ein beliebtes Lernverfahren. Eine Vielzahl von Ergebnissen verschiedener Wissenschaftler in den letzten Jahren, von denen einige in Kapitel 3 beschrieben wurden, bestätigen die gute Performanz dieses Lerners auch im Bereich der Angriffserkennung und der Klassifikation von Schadprogramm–Familien. Daher verwundert es, dass die SVM bezüglich der erreichbaren Accuracy und des Class Recalls aus dem in dieser Diplomarbeit durchgeführten Vergleich als schlechtestes Klassifikationsverfahren herausgeht. Die Methode weist bei der Klassifikation von Schadprogramm–Familien auf Thread–Vektoren einzelner Systemcalls gegenüber dem besten hier bewerteten Verfahren (dem Entscheidungsbaum–Lerner) unter der Berücksichtigung von Varianzen eine um mindestens 5,33% schlechtere Genauigkeit auf. Bei der Modellanwendung auf neue Daten beträgt die Accuracy–Differenz zu der des Entscheidungsbaum–Modells sogar 9,31%.

| Modell | Class–Recall pro Schadprogramm–Familie (%) | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ALLAPLE | BANKER | BIFROSE | BOBAX | DELF | KOLAB | KOLABC | RBOT | SALITY | SDBOT | VANBOT | VIRTUMONDE | VIRUT | ZHELATIN | ZLOB |
| SVM (nur Systemcalls) | 99,42 | 5,88 | 0,00 | 20,24 | 21,43 | 9,43 | 8,86 | 46,43 | 17,86 | 0,00 | 0,00 | 13,33 | 87,97 | 41,43 | 13,33 |
| SVM (6er-Sequenzen) | **99,73** | 19,61 | 0,00 | 30,95 | 35,71 | 13,21 | 68,35 | **71,43** | 50,00 | 0,00 | 0,00 | 60,00 | 92,90 | 85,24 | 46,67 |
| Nearest–Neighbor | **99,73** | **78,43** | 25,00 | **35,71** | 46,43 | 35,85 | 77,22 | 60,71 | 45,54 | 0,00 | 0,00 | 86,67 | 96,45 | **87,62** | **60,00** |
| Entscheidungstabelle | **99,73** | 58,82 | 16,67 | 29,76 | 25,00 | 26,42 | 69,62 | 60,71 | 36,61 | 0,00 | 0,00 | 86,67 | 96,84 | 86,19 | 26,67 |
| Entscheidungsbaum | **99,73** | 78,43 | 25,00 | 34,52 | **53,57** | 30,19 | **78,48** | **71,43** | 55,36 | 0,00 | 0,00 | 86,67 | **98,62** | **87,62** | **60,00** |
| Graph (Bayes) — TFIDF | 82,19 | 27,45 | 41,67 | 20,24 | 39,29 | **45,28** | 72,15 | 35,71 | **61,61** | 0,00 | 9,09 | **86,67** | 94,67 | 87,14 | 40,00 |
| Graph (Bayes) — TF | 98,69 | 11,76 | **58,33** | 23,21 | 10,71 | 35,85 | 7,59 | 14,29 | 35,71 | **3,85** | **18,18** | 60,00 | 83,83 | 84,29 | 26,67 |

Tabelle 5.12.: Class Recall bei der Anwendung der gelernten Modelle auf die Threads des Testdatensatzes. Bis auf den gekennzeichneten Fall des Graph–Lerners handelt sich dabei um die Darstellung der Beispiele als $TF$–Vektoren. Das beste Ergebnis pro Schadprogramm–Familie ist fett markiert.

| Lernverfahren | Accuracy (%) | | | | | |
|---|---|---|---|---|---|---|
| | BO | $\frac{\text{TF}}{f_d}$ | TFIDF | TF | Mengen (bestes Erg.) | Sequenzen (bestes Erg.) |
| SVM | 86,28 ± 0,33 | 83,20 ± 0,26 | 83,29 ± 0,41 | **87,20** ± 0,27 | 90,79 ± 0,24 | **90,92 ± 0,24** |
| Nearest–Neighbor | 91,06 ± 0,14 | 92,07 ± 0,17 | 92,06 ± 0,17 | **92,76 ± 0,15** | * | * |
| Entscheidungstabelle | 90,58 ± 0,11 | 90,56 ± 0,18 | 90,57 ± 0,17 | **91,68 ± 0,23** | * | * |
| Entscheidungsbaum | 91,05 ± 0,14 | 92,13 ± 0,18 | 92,15 ± 0,18 | **92,98 ± 0,18** | * | * |
| Graph (Bayes) | 81,38 ± 0,17 | 89,99 ± 0,23 | **90,47 ± 0,27** | 86,25 ± 0,26 | ** | ** |

Tabelle 5.13.: Zusammenfassung der in diesem Abschnitt und in den Abschnitten 5.5.1 und 5.5.2 durchgeführten Experimente. Accuracy der verschiedenen Lernverfahren auf den Threads des Lerndatensatzes, ermittelt mit 10–facher Kreuzvalidierung bei unterschiedlicher Berechnung der Vektorkomponenten einzelner Threads. Die "*"–Einträge bedeuten einen Abbruch des Experiments nach einer Laufzeit von mehr als 1 Tag und 6 Stunden. "**"–Einträge bedeuten einen systemseitigen Abbruch des Experiments wegen zu hohem Speicherplatzbedarf.

Erst bei der Anwendung der SVM auf *Sequenzvektoren* nähern sich die Ergebnisse des Verfahrens denen der übrigen getesteten Lernverfahren an. Allerdings erhöht sich die Laufzeit dabei stark, wie bereits in Abbildung 5.14 dargestellt wurde.

Der Entscheidungsbaum–Lerner und das $k$NN–Verfahren gehen aus dem Vergleich als die besten Klassifikatoren heraus.

Unabhängig von der verwendeten Merkmalsmenge und dem angewandten Lernverfahren scheinen sich die $TF$–Vektoren am besten zur Klassifikation der Threads zu eignen.

Wie zu Beginn dieses Abschnitts erläutert, liegen für das Nearest–Neighbor–Verfahren, die Entscheidungstabelle, den Entscheidungsbaum–Lerner und den Graph–Lerner *keine* Lernergebnisse auf der Grundlage von Sequenzvektoren und Mengenvektoren vor, da diese Klassifikationsverfahren bei den dabei entstehenden Merkmalsmengen (siehe Tabelle 5.4) immens speicherplatz– und rechenzeitintensiv sind.

Zusammenfassend kann folglich festgehalten werden, dass diese Verfahren bei großen Merkmalsmengen wie Systemcall–Sequenzen und Systemcall–Mengen bei der Klassifikation von Schadprogramm–Familien zwar ineffizient sind, auf Vektoren einzelner Systemcalls aber sehr gute Ergebnisse liefern. Für die Lernphase gilt demnach grob

$$\begin{aligned} & \textit{acc(SVM, Systemcalls(TF))} \\ < \; & \textit{acc(Graph, Systemcalls(TF))} \\ < \; & \textit{acc(SVM, 6er–Sequenzen(TF))} \\ < \; & \textit{acc(V, Systemcalls(TF))} \end{aligned}$$

mit V = {Nearest Neighbor, Entscheidungstabelle, Entscheidungsbaum}.

### 5.5.4. Bedeutung einzelner Systemcalls für die Klassifikation

Um einen Überblick über die Bedeutung einzelner Systemcalls für die Klassifikation von Schadprogramm–Familien zu erlangen, werden in diesem Abschnitt zunächst das Modell der Entscheidungstabelle und das Modell des Graph–Lerners ausgewertet. Diese Modelle wurden im Rahmen der in Abschnitt 5.5.3 beschriebenen Experimente auf $TF$–Systemcallvektoren der Threads erstellt. Anschließend werden die Ergebnisse einiger Gewichtungsverfahren vorgestellt und die hochgewichteten Systemcalls aller Verfahren werden miteinander verglichen.

**Die optimale Merkmalsmenge der Entscheidungstabelle** Der Algorithmus zur Erstellung einer Entscheidungstabelle beginnt bei der Suche nach der optimalen Merkmalsmenge $\mathcal{F}^* \subseteq \mathcal{F}$ für das Tabellenschema mit der leeren Menge und führt eine Vorwärtssuche durch. Wie in Abschnitt 4.2 beschrieben, wird das Problem in eine Zustandsraumsuche transformiert, in der die Teilmengen der Merkmale die Zustände darstellen und Operatoren Merkmale hinzufügen können. Da es sich bei dem vorliegenden Problem durch die $TF$–Werte der Vektorkomponenten um kontinuierliche Merkmalswerte handelt, werden laut [Koh95] Merkmale mit wenigen Werten genutzt und diese Werte werden in Intervalle zerlegt. Die Suche nach der optimalen Merkmalsmenge terminiert nach 5 Knotenexpansionen, durch die keine Verbesserung der aktuell besten Accuracy erzielt werden konnte. Bis zur Erfüllung dieses Kriteriums wurden in der in Abschnitt 5.5.3 durchgeführten

Lernphase des Algorithmus 4.113 Teilmengen bewertet. Als optimale Merkmalsmenge $\mathcal{F}^*$ zur Bildung des Tabellenschemas werden die folgenden 35 Systemcalls aus der Menge aller 120 Systemcalls ausgewählt:

| | | |
|---|---|---|
| `createOpenFile` | `winsockSection` | `connectionsOutgoingBlocked` |
| `ping` | `setFileAttributes` | `openFile` |
| `openKey` | `queryValue` | `loadDll` |
| `sleep` | `deleteFile` | `killProcess` |
| `createThread` | `createProcess` | `changeServiceConfig` |
| `createMutex` | `checkForDebugger` | `enumWindow` |
| `getFileAttributes` | `getComputerName` | `createFile` |
| `connectionsOutgoing` | `comCreateInstance` | `gethostbyname` |
| `destroyWindow` | `readValue` | `getWindowsDirectory` |
| `message` | `enumProcesses` | `setFileTime` |
| `dnsData` | `sendDatagram` | `ircData` |
| `createProcessNt` | `enumShare` | |

In der Entscheidungstabelle werden 108.936 von 129.088 Beispielen der Lernmenge gespeichert, indem sie auf diese 35 Systemcalls abgebildet werden. Die Verteilung der ausgewählten Beispiele entspricht der Verteilung im Lerndatensatz. Tabelle 5.14 zeigt einen Ausschnitt aus der erstellten Entscheidungstabelle.

| `createOpenFile` | ... | `loadDll` | `sleep` | ... | `enumProcesses` | ... | Label |
|---|---|---|---|---|---|---|---|
| (0.5-1.5] | ... | (65.5-inf) | (0.5-1.5] | ... | (-inf-0.5] | ... | *Allaple* |
| ... | ... | ... | ... | ... | ... | ... | ... |
| (-inf-0.5] | ... | (65.5-inf) | (6.5-20.5] | ... | (-inf-0.5] | ... | *Delf* |
| (22.5-35.5] | ... | 65.5-inf) | (1.5-2.5] | ... | (-inf-0.5] | ... | *Banker* |
| ... | ... | ... | ... | ... | ... | ... | ... |
| (4.5-5.5] | ... | (48.5-54.5] | (-inf-0.5] | ... | (-inf-0.5] | ... | *Allaple* |
| (-inf-0.5] | ... | (48.5-54.5] | (-inf-0.5] | ... | (-inf-0.5] | ... | *Delf* |
| ... | ... | ... | ... | ... | ... | ... | ... |
| (8.5-9.5] | ... | (54.5-56.5] | (5.5-6.5] | ... | (0.5-1.5] | ... | *Rbot* |
| ... | ... | ... | ... | ... | ... | ... | ... |

Tabelle 5.14.: Ausschnitt aus der durch den Algorithmus von Kohavi [Koh95] erstellten Entscheidungstabelle. Wie zu erkennen ist, wurden die kontinuierlichen Merkmalswerte in Intervalle zerlegt, so dass sie wie diskrete Merkmalswerte gehandhabt werden können.

**Die Systemcalls des Entscheidungsbaums mit dem höchsten Informationsgewinn**
Der Entscheidungsbaum–Lerner erstellt einen Baum mit 9.105 Knoten, davon 4.553 Blätter. Jedes Blatt deckt mindestens 2 Beispiele ab. Die Merkmale mit dem größten Informationsgewinn sind die, die der Wurzel am nächsten sind, da für den nächsten Knoten immer das Merkmal gewählt wird, das den größten Informationsgewinn liefert. Ein Ausschnitt des erstellten Baumes ist in Abbildung 5.15 dargestellt. Auf den obersten 5

Ebenen werden die folgenden 22 Systemcalls abgefragt:

| | | |
|---|---|---|
| **comCreateInstance** | **connectionsOutgoing** | **connectionsOutgoingBlocked** |
| connectionsUnknown | **createMutex** | **createOpenFile** |
| **createThread** | **destroyWindow** | enumKeys |
| **enumProcesses** | enumServices | findWindow |
| **getFileAttributes** | **getWindowsDirectory** | **loadDll** |
| loadImage | openScmanager | **readValue** |
| resultingAddr | setThreadContext | setValue |
| **sleep** | | |

Die 13 fett markierten Systemcalls sind ebenfalls in der Menge der Systemcalls enthalten, die das Schema der Entscheidungstabelle definieren.

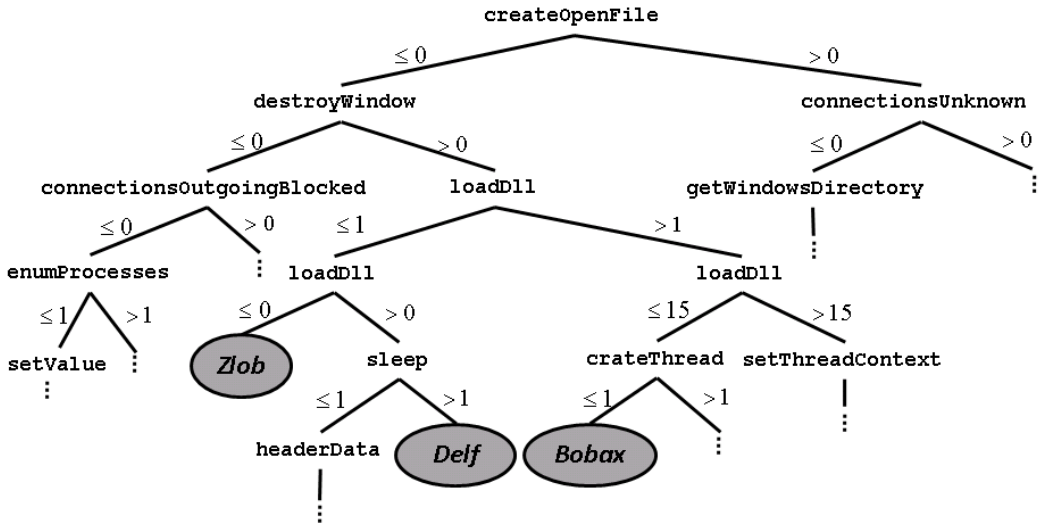



Abbildung 5.15.: Ein Ausschnitt des durch den erweiterten C4.5–Algorithmus erstellten Entscheidungsbaums

**Gewichtung durch ein Verfahren aus dem Bereich der Textanalyse** Da die Systemcall–Logdateien in dieser Arbeit als Textdateien dargestellt und verarbeitet werden, wird zunächst ein auf dieser Darstellung basierendes Gewichtungsverfahren angewendet.

Der in RapidMiner im Rahmen des Text-Plugins verfügbare *CorpusBasedWeighting-Operator* erstellt einen Gewichtsvektor aller Systemcalls *für jede Klasse*. Das Gewicht eines Systemcalls wird durch die Berechnung seiner durchschnittlichen Termfrequenz $TF$ (Definition 3.1) in allen Beispielen der vorliegenden Klasse und anschließender Normalisierung bestimmt.

Zur Erstellung *eines* Gewichtsvektors *für alle* 15 vorliegenden Schadprogramm–Familien wurde angenommen, dass Systemcalls deren Gewichtungen für verschiedene Familien sehr unterschiedlich sind, einen großen Beitrag zur Unterscheidung der Familien voneinander leisten. Ein für alle 15 Familien geltender, und somit von diversen Klassifikationsverfahren direkt verwendbarer Gewichtsvektor wurde erstellt, indem pro Systemcall

die Differenz zwischen dem maximalen und dem minimalen Gewicht über alle Familien $y_c \in \mathcal{Y}$ berechnet wurde.

Sei also $\vec{w}_c$ der vom CorpusBasedWeighting–Operator erzeugte Gewichtsvektor für die Familie $y_c \in \mathcal{Y}$. Dann wird der Gewichtsvektor $\vec{g}$ für die gesamten Beispielmenge über alle Familien berechnet durch

$$\vec{g} = \begin{pmatrix} g_1 \\ g_2 \\ \vdots \\ g_{|\mathcal{F}|} \end{pmatrix} \text{ mit } g_j = \max_{y_c \in \mathcal{Y}}(w_{jc}) - \min_{y_c \in \mathcal{Y}}(w_{jc}).$$

Dabei bezeichnet $w_{jc}$ die $j$–te Komponente des Gewichtsvektors für die Familie $y_c$, also das Gewicht des $j$–ten Systemcalls für die Familie $y_c$. Je größer die Differenz zwischen dem minimalen und dem maximalen CorpusBasedWeighting–Gewicht der einzelnen Familien ist, desto größer ist folglich auch das Gewicht $g_j$ des Gewichtsvektors für die gesamte Beispielmenge $\mathcal{X}$.

Die 23 Systemcalls mit einem Gewicht $g_j \geq 0,2$ sind in Tabelle 5.15 aufgelistet. Die Systemcalls, die bereits in der optimalen Merkmalsmenge des Tabellenschema der Entscheidungstabelle enthalten waren, sind fett gekennzeichnet und mit einem hochgestellten "T" versehen. Die Systemcalls, die bereits auf den oberen 5 Ebenen des Entscheidungsbaums benutzt wurden, sind fett gekennzeichnet und mit einem hochgestellten "B" versehen.

| Systemcall | Gewicht | Systemcall | Gewicht |
|---|---|---|---|
| **findWindow**$^{B}$ | 1,00 | **openFile**$^{T}$ | 0,59 |
| `getSystemTime` | 1,00 | `findFile` | 0,58 |
| `writeValue` | 1,00 | `vmProtect` | 0,45 |
| **createThread**$^{T,B}$ | 1,00 | **enumWindow**$^{T}$ | 0,39 |
| **sleep**$^{T,B}$ | 1,00 | **createProcess**$^{T}$ | 0,38 |
| **queryValue**$^{T}$ | 1,00 | **ping**$^{T}$ | 0,28 |
| **setValue**$^{B}$ | 1,00 | **createOpenFile**$^{T,B}$ | 0,27 |
| **loadDll**$^{T,B}$ | 0,99 | **createMutex**$^{T,B}$ | 0,23 |
| **openKey**$^{T}$ | 0,94 | `connection` | 0,20 |
| `openMutex` | 0,74 | **message**$^{T}$ | 0,20 |
| `enumModules` | 0,61 | **getFileAttributes**$^{T,B}$ | 0,20 |
| **enumProcesses**$^{T,B}$ | 0,61 | | |

Tabelle 5.15.: Die 23 Systemcalls mit einem $g_j$–Wert $\geq$ 0,2. Systemcalls des Tabellenschemas der Entscheidungstabelle sind mit $^{T}$ gekennzeichnet. Auf den oberen 5 Ebenen des Entscheidungsbaums benutzte Systemcalls sind mit $^{B}$ gekennzeichnet.

**SVM–Gewichtung mit RBF–Kern** Zur Bestimmung der Systemcalls, die zur Trennung der Klassen durch die SVM wichtig sind, wurde die SVM mit einem one–against–all Schema, $C = 120$ und einem RBF–Kern (Gleichung (4.15)) mit $\gamma = 0,008$ angewendet. Die Koeffizienten der 15 berechneten Normalenvektoren wurden als Gewichte interpretiert.

Für jeden Systemcall $s_j \in \mathcal{F}$ liegt also ein Wert $w_j$ für jede der $|\mathcal{Y}| = 15$ Schadprogramm–Familien vor. Die $w$–Werte wurden pro Familie normalisiert, so dass

$$\forall\, y_c = \{1, \ldots, |\mathcal{Y}|\}, j = \{1, \ldots, |\mathcal{F}|\} : w_{jc} \in [0, 1]$$

gilt. Die Systemcalls, denen bei einem der auf diese Weise berechneten Gewichtsvektoren ein Gewicht von 1 zugeordnet wird, sind `loadDll`, `openKey`, `sleep` und `writeValue`. Das Diagramm 5.16 zeigt die Gewichte dieser 4 Systemcalls (die normalisierten $w_{jc}$–Werte) für jede der 15 Schadprogramm–Familien.

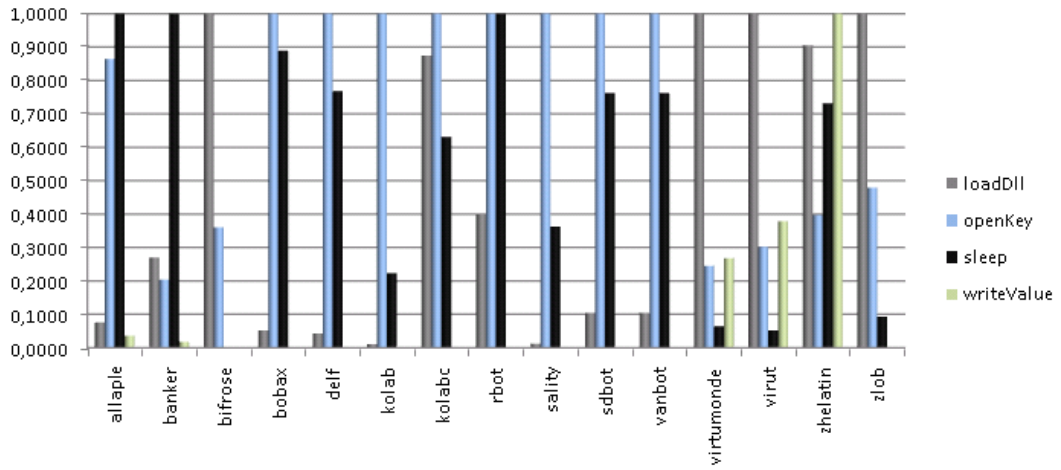



Diagramm 5.16.: Die 4 Systemcalls, für die bei einer one–against–all Trennung der SVM mit einem RBF–Kern für mindestens eine der Schadprogramm–Familien ein Gewicht von 1 berechnet wird. Dargestellt werden die Gewichte dieser Systemcalls für alle 15 Familien des Lerndatensatzes.

**SVM–Gewichtung bei linearer Trennung** Bei einer *linearer* Klassentrennung mit einem one–against–all Schema durch die SVM wurden ebenfalls die Koeffizienten der 15 berechneten Normalenvektoren normalisiert und als Gewichte der Systemcalls interpretiert. Dazu wurde der RapidMiner–Operator *SVMWeighting* verwendet. Um die Robustheit der Gewichte sicherzustellen, wurden nur die Systemcalls berücksichtigt, für die bei einer 10–fachen Kreuzvalidierung in jeder der 10 Runden ein Gewicht von 1 berechnet wird. Die Systemcalls, die diese Bedingung erfüllen und die Schadprogramm–Familien, für die das Gewicht von 1 berechnet wird, sind der Abbildung 5.17 und der dort dargestellten Matrix zu entnehmen. Systemcalls der anderen in diesem Abschnitt vorgestellten Merkmalsmengen sind wieder entsprechend gekennzeichnet.

**SVM–Gewichtung durch rekursives Entfernen von Systemcalls** Bei einem weiteren Gewichtungsverfahren anhand der Koeffizienten des SVM–Normalenvektors werden Merkmale rekursiv aus der Merkmalsmenge entfernt. Die SVM berechnet zunächst einen Normalenvektor $\vec{w}$ unter Berücksichtigung aller Systemcalls $\mathcal{F}$. Der Gewichtsvektor $\vec{c}$

`connectionsOutgoingBlocked`$^{T,B}$
`copyFile`
`createFile`$^{T}$
`createProcess`$^{T,g}$
`deleteValue`
`enumKeys`$^{B}$
`enumProcesses`$^{T,B,g}$
`getSystemTime`$^{g}$
`getWindowsDirectory`$^{T,B}$
`moveFile`
`openService`
`ping`$^{T,g}$

| | allaple | banker | bifrose | bobax | delf | kolab | kolabc | rbot | sality | sdbot | vanbot | virut | zhelatin | zlob |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| connectionsOutgoingBlocked | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| copyFile | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| createFile | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| createProcess | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |
| deleteValue | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |
| enumKeys | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| enumProcesses | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| getSystemTime | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| getWindowsDirectory | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| moveFile | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| openService | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| ping | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Hochgestellte Buchstaben bezeichnen die Merkmalsmengen, in denen dieser Systemcall bereits enthalten war. $^{T}$ steht für die Entscheidungstabelle, $^{B}$ für den Entscheidungsbaum und $^{g}$ für die Gewichte des Verfahrens der Textanalyse.

Matrix der Systemcalls mit einem Gewicht von 1 und der Klasse(n), für die dieser Gewichtswert berechnet wurde.

Abbildung 5.17.: Systemcalls, für die bei linearer Trennung one–against–all durch die SVM in allen 10 Runden der Kreuzvalidierung ein Wert von 1 für die Schadprogramm–Familie berechnet wird.

wird berechnet durch $c_j = (w_j)^2$. Der Systemcall mit dem kleinsten Gewicht, also $s_j$ mit

$$c_j = \arg\min_{1 \leq j \leq |\mathcal{F}|}(c_j),$$

wird aus der Menge der Systemcalls entfernt, so dass $\mathcal{F}_{temp} = \{\mathcal{F} \backslash s_j\}$ die neue Merkmalsmenge bildet. Das Verfahren wird rekursiv auf die Merkmalsmenge $\mathcal{F}_{temp}$ angewendet. Details dieses Verfahrens sind [GWBV02] zu entnehmen. Der in RapidMiner verfügbare Operator, der das Verfahren implementiert, nennt sich *W-SVMAttributeEval*.

Auch bei diesem Verfahren wurden zur Sicherstellung der Robustheit nur die Systemcalls berücksichtigt, für die bei einer 10–facher Kreuzvalidierung in mindestens einer der 10 Runden ein Gewicht von 1 berechnet wurde. Das Verfahren identifizierte die 3 Systemcalls `sleep`, `getComputerName` und `vmAllocate` als die zur Trennung der Schadprogramm–Familien wichtigsten Merkmale.

### Zusammenfassung der Ergebnisse

Die Gewichtungen der Systemcalls sind stellenweise sehr verschieden, da die Gewichtungsverfahren unterschiedliche Bewertungskriterien verwenden. Einige Systemcalls, die dennoch in mehreren Mengen vorhanden sind, werden im Folgenden aufgelistet. Dabei

gibt die Zahl in den Klammern die Anzahl der in diesem Abschnitt erzeugten Merkmalsmengen an, in denen der Systemcall enthalten ist:
`sleep` (5)
`loadDll` (4)
`enumProcesses` (4)
`createThread` (3)
`createOpenFile` (3)
`createMutex` (3)
`getFileAttributes` (3)
`openKey` (3)
`writeValue` (2)

Die 18 Systemcalls, denen bei der Gewichtung mit Hilfe des *CorpusBasedWeighting*–Operators und der anschließenden Berechnung des Gewichtsvektors $\vec{g}$ ein Gewicht von $g_j = 0$ zugewiesen wurde, werden auch bei den übrigen Gewichtungsverfahren mit sehr kleinen Gewichten gegen 0 bewertet.

Bei der Anwendung des Entscheidungsbaum–Lerners aus dem vorangegangenen Abschnitt auf die 23 Systemcalls aus der Tabelle 5.15 nahm die Accuracy des Verfahrens zwar um 2,04% ab, die Laufzeit verkürzte sich jedoch um beachtliche 63%.

In [YZF06] wurde die SVM auf eine durch die Rough Set Theorie reduzierte und gewichtete Merkmalsmenge angewendet. Die Rough Set Theorie wurde in Abschnitt 3.4.2 beschrieben. Die Laufzeit des Lernverfahrens und die Genauigkeit des Modells konnten in [YZF06] verbessert werden. Im Gegensatz dazu verschlechterten sich die Ergebnisse des SVM–Modells aus Abschnitt 5.5.2 bei der Anwendung eines beliebigen, in diesem Abschnitt berechneten, Gewichtsvektors. Zum Beispiel nahm die Accuracy der SVM bei der Anwendung des Verfahrens auf die Merkmalsmenge der 23 Systemcalls aus der Tabelle 5.15 gegenüber dem auf allen 120 Systemcalls gelernten Modell um etwa 3,2% ab. Die Laufzeit stieg um etwa 70%.

Die verschiedenen Gewichtungen dieses Abschnitts werden in dem folgenden Abschnitt 5.5.5 teilweise zur Auswahl der Systemcalls genutzt, deren Parameterwerte berücksichtigt werden. Es wird angenommen, dass die Einbeziehung der Parameterwerte von Systemcalls, die hier in mehreren Mengen mit einem hohen Gewicht belegt wurden, eine bessere Trennung der Schadprogramm–Familien liefert.

### 5.5.5. Berücksichtigung von Parametern und Werten

Bei den bisher vorgestellten Ansätzen wurden Merkmalsmengen aus einzelnen Systemcalls, Systemcall–Mengen oder Systemcall–Sequenzen zur Klassifikation von Schadprogramm–Familien verwendet. Die Parameternamen und die Parameterwerte der Systemcalls wurden dabei nicht berücksichtigt. Die Modelle sind folglich nicht dazu in der Lage Angriffe voneinander zu unterscheiden, die dieselben Systemcalls nutzen. Wie in Abschnitt 3.6 beschrieben, lässt sich jedoch zum Beispiel eine Mimicry Attacke ausschließlich anhand der Parameterwerte erkennen. Derartige Malware kann demnach auch nur der richtigen Schadprogramm–Familie zugeordnet werden, wenn die Systemcall–Argumente mit in die Analyse einbezogen werden. Daher wird in diesem Abschnitt die Bedeutung

von Parameterwerten für die Klassifikation der vorliegenden Schadprogramm–Familien evaluiert.

Ein Großteil der Arbeiten, die Systemcall–Argumente zur Erkennung von Angriffen berücksichtigen, verwenden zur Analyse nur eine kleine *Teilmenge der Systemcalls* mit *allen* Parameterwerten. Es werden Profile normaler Parameterwerte erzeugt, mit denen die Parameterwerte eines Systemcalls verglichen und aufgrund dessen klassifiziert werden. Anhand dieser Klassifikation wird anschließend eine Aussage über die Klasse des *Systemcalls* getroffen. Diese Klassifikation liefert dann wiederum die Basis zur Klassifikation ganzer Programmausführungen oder Teile dieser. Die in Abschnitt 3.6 vorgestellten Ansätze [Zan06] und [MVKV06] wenden ein solches Verfahren an.

Ein anderer Ansatz wurde in [RHW+08] verfolgt. Dort besteht die Merkmalsmenge zur Klassifikation von Schadprogramm–Familien aus Systemcalls mit unterschiedlichen Detaillierungsgraden der Parameter. Im Folgenden wird ein ähnlicher Ansatz gewählt. Dabei werden *alle* Systemcalls berücksichtigt, jedoch nur ein *Teil der Parameterwerte*. Es wird evaluiert, ob das Ergebnis der SVM durch eine solche Merkmalsmenge verbessert werden kann. Die in Abschnitt 5.5.3 bewerteten Verfahren werden hier nicht angewendet, da sich die Merkmalsmenge durch die Berücksichtigung von Parameterwerten erheblich vergrößert. Wie bereits mehrfach erwähnt, können diese Verfahren derart große Merkmalsmengen schlecht handhaben.

### Merkmalsmengen

Die Einbeziehung aller im Datensatz vorhandenen Parameter und Werte ist nicht möglich, da die dabei entstehende Merkmalsmenge zu groß wird. Ein Systemcall mit einem seiner Parameterwerte repräsentiert ein Merkmal, so dass kleinste Abweichungen der Parameterwerte zu einem weiteren Merkmal führen. Durch die Vielzahl möglicher Parameterwerte entsteht folglich eine enorm große Merkmalsmenge. Zum Beispiel erzeugt der Systemcall `open_file` mit seinem Parameter `srcfile` und den Parameterwerten `c:\123a` und `c:\123b` durch ein einziges differierendes Zeichen 2 Merkmale der Form

`openFile(srcfile=c:\123a)` und `openFile(srcfile=c:\123b)`.

Die zur Verfügung stehende Rechenleistung reichte zum Einlesen des Datensatzes bei der Berücksichtigung aller Parameterwerte nicht aus. Daher kann die Größe der dabei entstehenden Merkmalsmenge nicht genau spezifiziert werden.

Im Folgenden werden nur *einige* Parameterwerte ausgewählt. Zur Erzeugung einer Merkmalsmenge, in der Parameterwerte der Systemcalls berücksichtigt werden, müssen die Parameterwerte in den Textdateien enthalten sein, in die die XML–Berichte konvertiert werden. Folglich wird der in Abschnitt 5.2 beschriebene Parser zur selektiven Auswahl von Parameterwerten angewendet.

Wie in Abschnitt 5.2 ebenfalls bereits erläutert, kann jeder Systemcall eine fest vorgegebene Menge an Parametern spezifizieren. Die zu einem Systemcall gehörenden Parameternamen sind immer identisch. Daher können aus diesen Angaben keine zusätzlichen Informationen gewonnen werden. Da die Parameternamen hier zur Veranschaulichung jedoch hilfreich sind, werden sie trotzdem berücksichtigt.

Die Merkmalsmenge zur Erstellung der Vektoren enthält folglich neben allen Systemcalls *ohne* Parameterwerte für einige Systemcalls ein Merkmal für jeden auftretenden

Parameterwert. Es wird wieder die Thread–Darstellung der Daten gewählt, für die dann $TF$–Merkmalsvektoren erstellt werden.

Als Anhaltspunkt zur Auswahl der Systemcalls, für die ein oder mehrere Parameterwerte berücksichtigt werden, dienen die Ergebnisse des vorangegangenen Abschnitts 5.5.4. Dabei wird angenommen, dass die Parameterwerte der Systemcalls, die dort in mehreren Mengen mit einem hohen Gewicht belegt wurden, zu einer besseren Trennung der Klassen beitragen. Die bewerteten Merkmalsmengen werden zunächst beschrieben.

**Einzelne Parameter** Zu Beginn wurden die Parameterwerte eines oder mehrerer Parameter der Systemcalls berücksichtigt, für die in dem vorangegangenen Abschnitt bei der one–against–all Trennung der SVM mit RBF–Kern für mindestens eine der Familien ein Gewicht von 1 berechnet wurde (siehe Abbildung 5.16). Die Ergebnisse der SVM für jeden einzelnen Systemcall mit den ausgewählten Parameter sind in den ersten Zeilen der Tabellen 5.19 und 5.20 dargestellt. Ein "`a`" anstelle der Angabe eines Parameters kennzeichnet die Berücksichtigung *aller* Parameter des Systemcalls.

Die Systemcalls `openFile` und `changeServiceConfig` wurden intuitiv ausgewählt.

***RBF_SVM*** Die Menge $RBF\_SVM$ enthält die 4 Systemcalls aus dem Diagramm 5.16 mit jeweils *einem* Parameter, so dass in diesem Fall auf allen 120 Systemcalls des Datensatzes und den Werten der in der Tabelle 5.16 angegebenen Parameter gelernt wird.

| Systemcall | Parameter |
|---|---|
| `loadDll` | `filename` |
| `openKey` | `key` |
| `sleep` | `milliseconds` |
| `writeValue` | `section` |

Tabelle 5.16.: Parameter, deren Werte in der Menge $RBF\_SVM$ berücksichtigten werden.

***lin_SVM*** Die Menge $lin\_SVM$ enthält die in Abbildung 5.17 aufgeführten Systemcalls mit einem Wert von 1 bei einer linearer Trennung one–against–all der SVM. Auch hier wurde pro Systemcall *ein* zu berücksichtigender Parameter ausgewählt, so dass die Merkmalsmenge neben allen übrigen Systemcalls die Werte der in der Tabelle

Der Systemcall `connectionsOutgoingBlocked` ist ebenfalls in der Menge $lin\_SVM$ enthalten, besitzt jedoch keine Parameter. Für den Parameter `host` des Systemcalls `ping` sind keine doppelten Werte im Datensatz vorhanden. Da somit nicht davon auszugehen ist, dass dieser Parameter zur Verbesserung der Klassifikationsgenauigkeit beitragen kann, wurden diese Werte nicht berücksichtigt.

***rek*** Die Menge $rek$ enthält die 3 Systemcalls mit einem Wert von 1 bei rekursiver Entfernung der Merkmale anhand der Gewichtungen des SVM–Normalenvektors wie in Abschnitt 5.5.4 beschrieben. Dabei werden die Werte der Parameter berücksichtigt, die in der Tabelle 5.18 aufgeführt sind.

| Systemcall | Parameter |
|---|---|
| copyFile | srcfile |
| createFile | srcfile |
| createProcess | filename |
| deleteValue | key |
| enumKeys | key |
| enumProcesses | quantity |
| getSystemTime | quantity |
| getWindowsDirectory | quantity |
| moveFile | srcfile |
| openService | servicename |

Tabelle 5.17.: Parameter, deren Werte in der Menge $lin\_SVM$ berücksichtigten werden.

| Systemcall | Parameter |
|---|---|
| sleep | milliseconds |
| getComputerName | quantity |
| vmAllocate | protect |

Tabelle 5.18.: Parameter, deren Werte in der Menge $rek$ berücksichtigten werden.

### Ergebnisse

Die Tabellen 5.19 und 5.20 stellen die Ergebnisse der SVM in der Lernphase bei der Anwendung auf die um einige Parameterwerte erweiterten Merkmalsmengen dar. Da in diesem Abschnitt der Lerndatensatz* verwendet wird, in dem leere Dateien gelöscht wurden, wird zur Herstellung der Vergleichbarkeit zunächst das SVM–Ergebnis auf einzelnen Systemcalls *ohne* Berücksichtigung von Parameterwerten auf diesem leicht veränderten Datensatz angegeben.

Die SVM–Parameter entsprechen den Werten aus Abschnitt 5.5.1 (RBF–Kern, $\gamma = 0,008$, $C = 120$). Die Accuracy wird auch hier als Durchschnitt bei 10–facher Kreuzvalidierung berechnet.

**Auswertung der Ergebnisse der Lernphase** Obwohl nur ein kleiner Anteil der Parameterwerte berücksichtigt wurde und es eine Vielzahl weiterer Möglichkeiten und Kombinationen von Parameterwerten gibt, sind verglichen mit der Anwendung des Lernverfahrens auf Systemcalls *ohne* Parameterwerte bereits einige Tendenzen zu erkennen.

Eine grundsätzliche Beobachtung ist, dass sich die Accuracy unabhängig von den einbezogenen Parameterwerten verbessert. Die größte Steigerung wird mit bis zu 1,86% bei der Berücksichtigung aller Parameterwerte des Systemcalls `loadDll` erreicht. Die Accuracy–Varianz des Verfahrens sinkt dabei. Das bedeutet, dass das erstellte Modell stabiler wird. Ebenso sinkt die Anzahl der Stützvektoren. Es liegen folglich weniger Beispiele an den Klassengrenzen. Allerdings steigt die Laufzeit des Lernverfahrens.

Bei der Berücksichtigung der Werte des Parameters `servicename` des Systemcalls `changeServiceConfig` steigt die Accuracy zwar nur geringfügig, allerdings verkürzt sich die Laufzeit des Verfahrens hier um 25,32%.

| Merkmale | Ø Accuracy (%) | TP | FP | rel. Laufzeit-(%) Änderung (%) | $\|\mathcal{F}\|$ | rel. Änderung SV (%) |
|---|---|---|---|---|---|---|
| nur Systemcalls | 87,34 ± 0,36 | 112.563 | 16.055 | - | 120 | - |
| alle Systemcalls & `loadDll(filename)` | 88,59 ± 0,35 | 113.946 | 14.672 | -14,83 | 1.908 | -4,30 |
| alle Systemcalls & `loadDll (a)` | **88,64 ± 0,20** | 114.013 | 14.605 | +11,54 | 4.883 | **-4,73** |
| alle Systemcalls & `openKey(key)` | 87,81 ± 0,36 | 112.945 | 15.673 | -13,34 | 7.255 | -0,55 |
| alle Systemcalls & `sleep(milliseconds)` | 87,52 ± 0,11 | 112.563 | 16.055 | -23,88 | 757 | +5,86 |
| alle Systemcalls & `writeValue(section)` | 87,60 ± 0,18 | 112.672 | 15.946 | -24,87 | 134 | +0,00 |
| alle Systemcalls & $RBF\_SVM$ | 88,30 ± 0,17 | 113.567 | 15.051 | -2,70 | 9.424 | +0,52 |
| alle Systemcalls & `openFile(srcfile)` | 87,56 ± 0,24 | 112.616 | 16.002 | -19,68 | 5.444 | +0,08 |
| alle Systemcalls & `changeServiceConfig (servicename)` | 87,63 ± 0,29 | 112.702 | 15.916 | **-25,32** | 5.444 | +0,07 |
| alle Systemcalls & $lin\_SVM$ | 87,54 ± 0,37 | 112.291 | 16.327 | -14,93 | 5.751 | +0,65 |
| alle Systemcalls & $rek$ | 87,51 ± 0,27 | 112.559 | 16.059 | -24,86 | 791 | +3,95 |

Tabelle 5.19.: Accuracy der SVM auf den $TF$–Vektoren der Threads des Lerndatensatzes* bei einer Merkmalsmenge aus den Systemcalls und einigen Parameterwerten. Berechnung der Accuracy als Durchschnitt der Werte aller Klassen bei 10–facher Kreuzvalidierung. Angabe der Anzahl an Merkmalen $|\mathcal{F}|$, der relativen Laufzeitänderung bezüglich der Laufzeit des Lernverfahrens auf $TF$–Vektoren *ohne* Berücksichtigung von Parameterwerten und dem relativen Zuwachs bzw. der Abnahme an Stützvektoren (SV). Die beste Accuracy, die größte Laufzeitreduzierung und die höchste Abnahme der Stützvektoren sind fett markiert.

Obwohl die Anzahl der Merkmale $|\mathcal{F}|$ und somit die Dimension der Vektoren bei der Berücksichtigung der Parameterwerte der Menge $RBF\_SVM$ immens steigt und mit 9.424 unterschiedlichen Merkmalen 9.304 zusätzliche Merkmale gegenüber der Merkmalsmenge ohne Parameterwerte enthält, kann die Laufzeit um 2,7% gesenkt werden. Die Genauigkeit des Verfahrens steigt um etwa 1%. Die Varianz der Accuracy sinkt.

Bei der Auswertung der Recall–Werte aus Tabelle 5.20 lassen sich noch etwas deutlichere Tendenzen erkennen. Die Menge $RBF\_SVM$ weist für alle Schadprogramm–Familien, ausgenommen der Familie *Bobax*, höhere Werte auf als das Verfahren bei der Anwendung auf Systemcalls ohne Parameterwerte. Die Kombination der berücksichtigten Parameterwerte scheint demnach dazu beizutragen, dass die SVM die Schadprogramme besser voneinander trennen kann.

Da der Recall der Familie *Bobax* bei der Berücksichtigung der Werte *aller* Parameter des Systemcalls `loadDll` steigt, scheinen für diese Familie die Werte eines Parameters charakteristisch zu sein, die in der Menge $RBF\_SVM$ nicht berücksichtigt wurden. Neben dem Parameter `filename` besitzt dieser Systemcall noch die Parameter `address`, `end_address`, `quantity`, `successfull`, `size` und `filename_hash`.

| Merkmale | Class–Recall pro Schadprogramm–Familie (%) | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ALLAPLE | BANKER | BIFROSE | BOBAX | DELF | KOLAB | KOLABC | RBOT | SALITY | SDBOT | VANBOT | VIRTUMONDE | VIRUT | ZHELATIN | ZLOB |
| nur Systemcalls | 99,35 | 15,50 | 8,46 | 48,22 | 24,56 | 9,69 | 9,88 | 15,81 | 26,80 | 6,12 | 6,46 | 16,84 | 83,16 | 19,16 | 70,30 |
| alle Systemcalls & `loadDll(filename)` | 99,70 | 23,66 | 5,38 | 52,23 | 27,89 | 8,72 | 10,84 | 17,54 | **34,65** | 6,65 | 7,60 | **52,53** | 86,03 | 23,70 | 70,00 |
| alle Systemcalls & `loadDll (a)` | 99,70 | 18,88 | 4,62 | **53,70** | 29,71 | 9,69 | 10,12 | 17,32 | 31,31 | 6,91 | 7,98 | **52,53** | 86,00 | **27,90** | 70,45 |
| alle Systemcalls & `openKey(key)` | 99,50 | 21,45 | 4,62 | 46,21 | 26,63 | 10,41 | 10,12 | 17,19 | 27,55 | 4,52 | 9,13 | 12,79 | 85,88 | 21,18 | 71,06 |
| alle Systemcalls & `sleep(milliseconds)` | 99,63 | 15,62 | 6,92 | 40,83 | 24,69 | 9,44 | 10,36 | **17,98** | 29,71 | 5,05 | 6,46 | 16,50 | 85,42 | 19,58 | 71,06 |
| alle Systemcalls & `writeValue(section)` | 99,35 | 16,78 | 8,46 | 51,00 | 24,50 | 10,17 | 8,43 | 16,72 | 28,35 | 6,38 | 5,32 | 16,84 | 83,48 | 19,66 | 70,45 |
| alle Systemcalls & *RBF_SVM* | **99,87** | **25,52** | **9,23** | 43,65 | **30,72** | 10,90 | **12,53** | 17,95 | **34,65** | **7,45** | 6,46 | 48,15 | **86,11** | 24,45 | 70,45 |
| alle Systemcalls & `openFile(srcfile)` | 99,37 | 17,95 | 6,15 | 44,80 | 25,57 | **12,11** | 12,29 | 17,51 | 27,74 | 4,52 | **45,63** | 19,19 | 84,76 | 19,24 | **72,73** |
| alle Systemcalls & `changeServiceConfig (servicename)` | 99,35 | 16,20 | 6,92 | 50,85 | 23,74 | 10,41 | 10,36 | 15,77 | 28,77 | 5,59 | 6,46 | 16,84 | 83,91 | 19,75 | 70,45 |
| alle Systemcalls & *lin_SVM* | 99,35 | 15,85 | 6,15 | 46,14 | 23,37 | 9,93 | 10,60 | 17,22 | 30,75 | 7,18 | 7,22 | 16,84 | 84,85 | 19,41 | 70,91 |
| alle Systemcalls & *rek* | 99,61 | 14,57 | 7,69 | 41,15 | 23,87 | 10,65 | 11,57 | 17,82 | 30,47 | 4,52 | 6,08 | 16,84 | 85,37 | 19,75 | 70,61 |

Tabelle 5.20.: Class Recall der SVM auf den $TF$–Vektoren der Threads des Lerndatensatzes* bei einer Merkmalsmenge aus den Systemcalls und einigen Parameterwerten im Vergleich zu den Ergebnissen ohne Berücksichtigung von Parameterwerten. Der jeweils beste Wert pro Schadprogramm–Familie ist fett markiert.

Der Class Recall der Schadprogramm–Familie *Vanbot* ist bei der Einbeziehung der Werte des Parameters `srcfile` des Systemcalls `openFile` fast 7 Mal höher als bei den anderen Merkmalsmengen. Der Wert dieses Parameters scheint folglich wertvolle Informationen zur Erkennung dieser Familie zu beinhalten.

| Modell | Accuracy (%) | TP | FP | Rel. Laufzeit-Änderung (%) |
|---|---|---|---|---|
| nur Systemcalls | 82,37 | 3219 | 689 | - |
| alle Systemcalls & `loadDll (filename)` | 86,36 | 3375 | 533 | +71,89 |
| alle Systemcalls & $RBF\_SVM$ | 86,08 | 3364 | 544 | +245,78 |
| alle Systemcalls & `changeServiceConfig (servicename)` | 82,73 | 3233 | 675 | -3,41 |

Tabelle 5.21.: Accuracy, TP und FP einer Auswahl gelernter SVM–Modelle auf dem Testdatensatz* unter Berücksichtigung von Parameterwerten in der Merkmalsmenge.

**Auswertung der Ergebnisse der Testphase** In Tabelle 5.21 sind die Ergebnisse einer Auswahl der unter Berücksichtigung von Parameterwerten gelernten Modelle bei der Anwendung auf die Threads der Testmenge* dargestellt. Entsprechend der Modellgewinnung auf dem Lerndatensatz*, in dem leere Dateien gelöscht wurden, wird auch der entsprechende Testdatensatz* benutzt, in dem keine leeren Dateien vorhanden sind.

Die Testphase bestätigt einige Ergebnisse der Lernphase. Auch hier verbessert sich die Accuracy bei der Berücksichtigung von Parameterwerten. Das Modell, das auf den Parameterwerten der Menge $RBF\_SVM$ gelernt wurde, ist deutlich besser als das Modell, bei dem keine Parameterwerte berücksichtigt wurden.

Die Laufzeit zur Klassifikation der Beispiele verringert sich bei der Berücksichtigung der Werte des Parameters `servicename` des Systemcalls `changeServiceConfig`, allerdings verlängert sich die Laufzeit bei den übrigen Modellen entgegen der Laufzeiten in der Lernphase.

Der Class Recall bei der Anwendung der Modelle auf den Testdatensatz* ist in Tabelle 5.22 dargestellt. Die Schadprogramm–Familie *Kolabc* wird besser erkannt, wenn *keine* Parameterwerte berücksichtigt werden. Für die Familien *Bifrose*, *Sdbot* und *Vanbot* war der Recall bereits in der Lernphase niedrig. Diese Familien werden in der Testphase nicht erkannt. Die Berücksichtigung der Parameterwerte aus der Menge $RBF\_SVM$ verbessert den Class Recall zum Teil stark. Bei den Familien *Delf* und *Zhelatin* kann die Trefferquote verglichen mit dem Modell bei dem keine Parameterwerte berücksichtigt wurden, durch diese erweiterte Merkmalsmenge ungefähr verdoppelt werden. Es bestätigt sich demnach die Vermutung der Lernphase, dass die Kombination der in der Menge $RBF\_SVM$ berücksichtigten Parameterwerte zur besseren Trennbarkeit der Beispiele durch die SVM beiträgt.

| Modell | Class–Recall pro Schadprogramm–Familie (%) | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ALLAPLE | BANKER | BIFROSE | BOBAX | DELF | KOLAB | KOLABC | RBOT | SALITY | SDBOT | VANBOT | VIRTUMONDE | VIRUT | ZHELATIN | ZLOB |
| nur Systemcalls | 99,39 | 5,88 | 0,00 | 21,82 | 25,00 | 9,43 | **8,86** | 42,86 | 17,86 | 0,00 | 0,00 | 13,33 | 90,16 | 41,43 | 13,33 |
| alle Systemcalls & `loadDll(filename)` | 99,73 | 9,80 | 0,00 | **31,52** | 32,14 | **16,98** | 5,06 | **50,00** | **36,61** | 0,00 | 0,00 | **26,67** | 91,97 | **85,24** | 13,33 |
| alle Systemcalls & *RBF_SVM* | **99,88** | **13,73** | 0,00 | 23,64 | **46,43** | 9,43 | 5,06 | 32,14 | 35,71 | 0,00 | 0,00 | 20,00 | **92,37** | **85,24** | **20,00** |
| alle Systemcalls & `changeServiceConfig (servicename)` | 99,39 | 5,88 | 0,00 | 30,30 | 25,00 | 9,43 | 5,06 | 46,43 | 19,64 | 0,00 | 0,00 | 13,33 | 90,16 | 41,43 | 13,33 |

Tabelle 5.22.: Class Recall einer Auswahl gelernter SVM–Modelle auf dem Testdatensatz* unter der Berücksichtigung von Parameterwerten in der Merkmalsmenge. Der jeweils beste Wert pro Schadprogramm–Familie ist fett markiert.

**Zusammenfassung der Ergebnisse** Zusammenfassend kann aufgrund der in diesem Abschnitt präsentierten Ergebnisse festgehalten werden, dass die Berücksichtigung von Parameterwerten dazu beiträgt, dass die SVM die Schadprogramm–Familien besser voneinander trennen kann. Es gibt Parameterwerte anhand derer bestimmte Familien besser erkannt werden können. Dementsprechend verbessert sich der Class Recall zum Teil erheblich. Trotz der wesentlich größeren Merkmalsmengen kann die Laufzeit in der Lernphase in den meisten Fällen gesenkt werden, verlängert sich jedoch unter Umständen bei der Klassifikation neuer Beispiele.

### 5.5.6. Verbesserung der Familienvorhersage

In den vorangegangenen Abschnitten wurden verschiedene Lernverfahren auf der Grundlage unterschiedlicher Vektordarstellungen und Merkmalsmengen evaluiert. Dabei wurden einzelne Threads der Systemcall–Logdateien klassifiziert. Die Threads eines Traces werden dabei häufig verschiedenen Schadprogramm–Familien zugeordnet. Durch ähnliche Verhaltensweisen der verschiedenen Schadprogramme kann eine solche Klassifikation stellenweise sinnvoll sein.

Da die Threads in dem verwendeten Datensatz jedoch alle mit dem Label des Traces versehen sind, aus dem sie extrahiert wurden, wird in diesem Abschnitt bewertet, ob das Klassifikationsergebnis verbessert werden kann, wenn für alle Threads eines Traces dieselbe Klasse vorhergesagt wird oder ob die Anzahl der Fehlklassifikationen dann steigt.

Zudem kann auf der Grundlage einer solchen Klassifikation eine Aussage über die Familienzugehörigkeit eines ganzen Traces getroffen werden. Zur Bestimmung der "Trace–Familie" können die berechneten Familienzugehörigkeiten der einzelnen Threads genutzt werden. Dabei sind verschiedene Vorgehensweisen denkbar.

Die wohl einfachste Methode ist eine Mehrheitsentscheidung, bei der alle Threads eines Traces und folglich auch das Trace selbst der Familie zugeordnet werden, die für die Mehrheit der Threads vorhergesagt wurde. Im Folgenden wird ein ähnliches Verfahren angewendet. Allerdings werden dabei die *Konfidenzen* einzelner Threads für die verschiedenen Familien berücksichtigt. Eine hohe Klassenkonfidenz (oder Familienkonfidenz) steht für eine hohe Zugehörigkeitswahrscheinlichkeit des Threads zu dieser Familie.

Die Familienkonfidenzen aller zu einem Trace gehörenden Threads $\vec{x}_i \in X_T$ mit $X_T$ der Menge der Threads des Traces $T$ werden für jede einzelne Familie aufsummiert. Als neues Label für alle Threads eines Traces wird die Schadprogramm–Familien gewählt, die dabei den höchsten Wert erreicht. Formal bedeutet dies, dass das neue Label $\hat{y}'$ eines Threads bestimmt wird durch

$$\hat{y}' = \max_{y_c \in \mathcal{Y}} \sum_{\vec{x}_i \in X_T} conf_c(\vec{x}_i). \tag{5.2}$$

Dabei bezeichnet $\mathcal{Y}$ die Menge der Schadprogramm–Familien mit $y_c \in \mathcal{Y}$ einer Familie. In dem vorliegenden Datensatz gibt es 15 Familien, so dass $|\mathcal{Y}| = 15$ gilt. $conf_c(\vec{x}_i)$ gibt die Konfidenz des Threads $\vec{x}_i$ für die Familie $y_c$ an. Falsche Familienvorhersagen einzelner Threads, die durch kleine Abweichungen der Konfidenz–Werte der tatsächlichen Familie und der vorhergesagten Familie hervorgerufen werden, fallen demnach bei der Klassifikation weniger stark ins Gewicht als bei einer einfachen Merhheitsentscheidung.

Zur Integration dieser Zuordnung in die Lernumgebung RapidMiner wurde ein Operator implementiert, der die in Gleichung (5.2) definierte Funktion realisiert. Dieser Operator kann in einem Nachbearbeitungsschritt auf den Datensatz der durch verschiedene Modelle klassifizierten Threads angewendet werden.

Tabelle 5.23 zeigt die Ergebnisse dieser Nachbearbeitung für eine Auswahl von Modellen, die im Rahmen vorangegangener Experimente erstellt und im Verlauf dieses Kapitels beschrieben wurden. Zum Vergleich der Ergebnisse werden die Werte *vor* der Anwendung des Operators ebenfalls angegeben.

| Modell | Accuracy (%) | TP | FP | Verbesserung Accuracy (%) |
|---|---|---|---|---|
| SVM (nur Systemcalls) | 82,02<br>**91,02** | 3.215<br>**3.568** | 705<br>**352** | 9,00 |
| SVM (6er–Sequenzen) | 88,60<br>**93,16** | 3.473<br>**3.652** | 447<br>**268** | 4,56 |
| Nearest Neighbor | 90,74<br>**93,75** | 3.557<br>**3.675** | 363<br>**236** | 3,01 |
| Entscheidungstabelle | 89,36<br>**92,42** | 3.503<br>**3.623** | 417<br>**297** | 3,06 |
| Entscheidungsbaum | 91,33<br>**93,29** | 3.581<br>**3.657** | 339<br>**263** | 1,96 |
| Graph (Bayes)— TF | 84,52<br>**89,97** | 3.313<br>**3.527** | 607<br>**393** | 5,45 |
| SVM (alle Systemcalls)&<br>`loadDll(filename)` * | 86,36<br>**91,02** | 3.375<br>**3.557** | 533<br>**351** | 4,66 |
| SVM (alle Systemcalls)&<br>$RBF\_SVM$* | 86,08<br>**90,74** | 3.364<br>**3.546** | 544<br>**362** | 4,66 |
| SVM (alle Systemcalls)&<br>`changeServiceConfig`<br>`(servicename)`* | 82,73<br>**91,07** | 3.219<br>**3.559** | 689<br>**349** | 8,34 |

Tabelle 5.23.: Ergebnisse vor dem Nachbearbeitungsschritt (die jeweils erste Zeile) und nach dem Nachbearbeitungsschritt (die jeweils zweite Zeile) bei der Anwendung einer Auswahl von Modellen auf den Testdatensatz: Accuracy, TP und FP bei der Vorhersage derselben Schadprogramm–Familie für alle Threads eines Traces aufgrund der Einzelkonfidenzen der Threads für jede Familie sind fett markiert. Mit "*" gekennzeichnete Modelle wurden auf dem Datensatz erstellt, in dem leere Dateien gelöscht wurden und werden hier auch entsprechend auf den Testdatensatz* angewendet.

**Ergebnisse** Durch den Nachbearbeitungsschritt kann die Accuracy aller Modelle erhöht werden. Besonders auffällig ist die Verbesserung bei dem SVM–Modell, das auf Systemcallvektoren gelernt wurde. Die Differenz beträgt 9%, so dass dieses Modell nun ebenfalls eine Accuracy von über 90% erreicht. Auch die ohnehin hohen Genauigkeiten des $k$NN–Verfahrens und des Entscheidungsbaum–Lerners konnten noch weiter verbessert werden.

| Modell | Class–Recall pro Schadprogramm–Familie (%) | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | ALLAPLE | BANKER | BIFROSE | BOBAX | DELF | KOLAB | KOLABC | RBOT | SALITY | SDBOT | VANBOT | VIRTUMONDE | VIRUT | ZHELATIN | ZLOB |
| SVM | 99,42 | 5,88 | 0,00 | 20,24 | 21,43 | 9,43 | 8,86 | 46,43 | 17,86 | 0,00 | 0,00 | 13,33 | 87,97 | 41,43 | 13,33 |
| (nur Systemcalls) | **100,00** | **92,16** | **0,00** | **38,10** | **46,43** | **18,87** | **24,05** | **64,29** | **80,36** | **0,00** | **0,00** | **40,00** | **99,41** | **90,00** | **20,00** |
| SVM | 99,73 | 19,61 | 0,00 | 30,95 | 35,71 | 13,21 | 68,35 | 71,43 | 50,00 | 0,00 | 0,00 | 60,00 | 92,90 | 85,24 | 46,67 |
| (6er-Sequenzen) | **100,00** | **96,08** | **0,00** | **38,10** | **39,29** | **37,74** | **96,20** | **89,29** | **80,36** | **0,00** | **0,00** | **80,00** | **99,41** | **90,00** | **46,67** |
| Nearest–Neighbor | 99,73 | 78,43 | 25,00 | 35,71 | 46,43 | 35,85 | 77,22 | 60,71 | 45,54 | 0,00 | 0,00 | 86,67 | 96,45 | 87,62 | 60,00 |
| | **100,00** | **96,08** | **58,33** | **38,10** | **60,71** | **37,74** | **96,20** | **89,29** | **80,36** | **0,00** | **0,00** | **100,00** | **100,00** | **88,57** | **93,33** |
| Entscheidungstabelle | 99,73 | 58,82 | 16,67 | 29,76 | 25,00 | 26,42 | 69,62 | 60,71 | 36,61 | 0,00 | 0,00 | 86,67 | 96,84 | 86,19 | 26,67 |
| | **100,00** | **94,12** | **0,00** | **38,10** | **50,00** | **37,74** | **82,28** | **78,57** | **66,07** | **0,00** | **0,00** | **100,00** | **100,00** | **88,57** | **20,00** |
| Entscheidungsbaum | 99,73 | 78,43 | 25,00 | 34,52 | 53,57 | 30,19 | 78,48 | 71,43 | 55,36 | 0,00 | 0,00 | 86,67 | 98,62 | 87,62 | 60,00 |
| | **100,00** | **100,00** | **25,00** | **38,10** | **60,71** | **37,74** | **96,20** | **89,29** | **66,07** | **0,00** | **0,00** | **100,00** | **100,00** | **88,57** | **93,33** |
| Graph (Bayes) | 98,69 | 11,76 | 58,33 | 23,21 | 10,71 | 35,85 | 7,59 | 14,29 | 35,71 | 3,85 | 18,18 | 60,00 | 83,83 | 84,29 | 26,67 |
| TF | **99,54** | **0,00** | **50,00** | **36,31** | **39,29** | **60,38** | **10,13** | **53,57** | **80,36** | **0,00** | **27,27** | **80,00** | **99,41** | **88,57** | **40,00** |
| alle Systemcalls & | 99,73 | 9,80 | 0,00 | 31,52 | 32,14 | 16,98 | 5,06 | 50,00 | 36,61 | 0,00 | 0,00 | 26,67 | 91,97 | 85,24 | 13,33 |
| `loadDll(filename)` | **100,00** | **92,16** | **0,00** | **38,79** | **50,00** | **18,87** | **24,05** | **64,29** | **80,36** | **0,00** | **0,00** | **40,00** | **99,40** | **88,57** | **20,00** |
| alle Systemcalls & | 99,88 | 13,73 | 0,00 | 23,64 | 46,43 | 9,43 | 5,06 | 32,14 | 35,71 | 0,00 | 0,00 | 20,00 | 92,37 | 85,24 | 20,00 |
| *RBF_SVM* | **100,00** | **92,16** | **0,00** | **38,79** | **50,00** | **18,87** | **10,13** | **64,29** | **80,36** | **0,00** | **0,00** | **40,00** | **99,40** | **88,57** | **20,00** |
| alle Systemcalls & | 99,39 | 5,88 | 0,00 | 30,30 | 25,00 | 9,43 | 5,06 | 46,43 | 19,64 | 0,00 | 0,00 | 13,33 | 90,16 | 41,43 | 13,33 |
| `changeServiceConfig` | **100,00** | **92,16** | **0,00** | **38,79** | **46,43** | **18,87** | **24,05** | **64,29** | **80,36** | **0,00** | **0,00** | **40,00** | **99,40** | **90,00** | **20,00** |
| `(servicename)` | | | | | | | | | | | | | | | |

Tabelle 5.24.: Class Recall einer Auswahl der gelernten Modelle vor der Nachbearbeitung (die jeweils erste Zeile) und nach der Nachbearbeitung (die jeweils zweite Zeile) der Familienvorhersagen durch die Zuordnung aller Threads eines Traces zu der Schadprogramm–Familie mit dem höchsten Konfidenzwert über alle Threads. Mit "*" gekennzeichnete Modelle wurden auf dem Datensatz erstellt, in dem leere Dateien gelöscht wurden und wurden hier auch entsprechend auf den Testdatensatz* angewendet.

Eine Performanzsteigerung ist auch bei der Betrachtung des Class Recalls in Tabelle 5.24 deutlich zu erkennen. Zum Beispiel werden die Familien *Banker* und *Sality* bei den SVM–Modellen nun deutlich besser erkannt. Bis auf einige wenige Ausnahmen verbessert sich der Recall bei *allen* Modellen über alle Schadprogramm–Familien hinweg zum Teil erheblich. Die Ausnahmen bilden einige Werte der Entscheidungstabelle und des Graph–Modells.

Zusammenfassend kann festgehalten werden, dass die Anwendung dieser einfachen Funktion bereits deutliche Verbesserungen der Accuracy und des Recalls bei der Klassifikation neuer Daten des Testdatensatzes liefert.

### Vergleich mit der Klassifikation von Traces

Um die auf einzelnen Threads erzielten Ergebnisse mit Modellen vergleichen zu können, die ganze Systemcall–Berichte klassifizieren, ist es notwendig zu evaluieren, wie vielen Traces die klassifizierten Threads entsprechen und wie viele Traces demnach richtig und falsch klassifiziert werden. Diese Bewertung wurde beispielhaft für das Modell des Entscheidungsbaum–Lerners bei der Klassifikation der Testdaten durchgeführt.

Nach der Aktualisierung der Familienvorhersagen des Entscheidungsbaum–Modells durch die beschriebene Nachbearbeitungsprozedur werden noch 297 Threads falsch klassifiziert (siehe Tabelle 5.23). Diese Threads gehören zu 27 Traces. Bei der Anwendung des Algorithmus zur Erstellung eines Entscheidungsbaums auf $TF$–Systemcallvektoren ganzer *Traces* wird in der Lernphase lediglich eine Accuracy von 60,83% ± 1,94% erreicht. Bei der Klassifikation der 75 Traces des Testdatensatzes werden lediglich 8 Traces der richtigen Schadprogramm–Familie zugeordnet. Folglich werden 67 Traces falsch klassifiziert.

Wie Tabelle 5.25 zu entnehmen ist, ist auch die Accuracy der übrigen Verfahren bei der Durchführung der Lernphase auf Systemcallvektoren ganzer Traces schlecht. Die vorliegenden Schadprogramm–Familien können demnach auf der Grundlage ganzer Traces nicht gut klassifiziert werden. Die Klassifikation ganzer Traces durch die Klassifikation ihrer Threads scheint folglich bessere Ergebnisse zu liefern.

| Lernverfahren | Komponentenberechnung | | | |
|---|---|---|---|---|
| | TFIDF | Term Frequency | Term Occurence | Binary Occurence |
| SVM (nur Systemcalls) | 59,13 ± 2,02 | 49,12 ± 5,56 | 60,83 ± 1,94 | **77,39 ± 3,56** |
| Nearest–Neighbor | 75,01 ± 2,64 | 74,11 ± 1,57 | 73,60 ± 2,71 | **77,39 ± 2,38** |
| Entscheidungstabelle | 77,22 ± 2,21 | **77,73 ± 2,90** | 66,37 ± 3,43 | 77,50 ± 2,93 |
| Entscheidungsbaum | 77,22 ± 2,21 | 77,73 ± 2,90 | **80,21 ± 1,77** | 77,50 ± 2,93 |
| Graph (Bayes) | **65,01 ± 3,66** | 65,55 ± 3,62 | 62,97 ± 3,68 | 55,51 ± 2,62 |

Tabelle 5.25.: Accuracy verschiedener Lernverfahren auf den Traces des Lerndatensatzes bei unterschiedlicher Komponentenberechnung der Systemcallvektoren, ermittelt als Durchschnitt der Werte aller Klassen bei 10–facher Kreuzvalidierung. Das beste Ergebnis jedes Lernverfahrens ist fett markiert. (Die Ergebnisse für Sequenzvektoren, Mengenvektoren und Parameterwerte werden nicht angegeben.)

# 6. Zusammenfassung

Die automatische Klassifikation von Schadprogrammen ist eine komplexe Aufgabe, die aufgrund der steigenden Vielfalt auftretender Angriffe und der ständigen Veränderung der Programmcodes bekannter Malware immer mehr an Bedeutung gewinnt. Klassische signaturbasierte Verfahren, wie sie von den meisten Antivirussoftware–Herstellern verwendet werden, können nur die Schadprogramme erkennen, für die bereits eine Signatur vorliegt. Die Signaturen müssen häufig noch manuell erstellt werden, so dass es kaum möglich ist, mit der schnellen Entwicklung bösartiger Programme Schritt zu halten.

Das Ziel dieser Diplomarbeit war die Analyse von Betriebssystem–Logdateien zur automatischen Klassifikation von Malware. Dazu wurden Data Mining Techniken eingesetzt. Es wurde versucht, Varianten einer Schadprogramm–Familie auf der Basis von Systemcall–Logdateien der richtigen Familie zuzuordnen.

Dieses Kapitel fasst in Abschnitt 6.1 die Ergebnisse der in Kapitel 5 durchgeführten Experimente zur Klassifikation von Schadprogramm–Familien zusammen. Zudem werden einige Probleme der vorgestellten Ansätze genannt. In Abschnitt 6.2 werden Lösungsvorschläge skizziert, weiterführende Analysen der vorliegenden Systemcall–Logdateien und Erweiterungen der Modelle vorgeschlagen und Verbesserungspotenziale der vorgestellten Ansätze aufgezeigt. Abschließend wird in Abschnitt 6.3 ein Fazit gezogen.

## 6.1. Zusammenfassung

In Kapitel 5 wurden verschiedene Ansätze zur Klassifikation von Schadprogramm–Familien auf der Grundlage von Systemcall–Logdateien vorgestellt. Die einzelnen Threads einer Schadprogramm–Ausführung wurden extrahiert und in eine Textdarstellung konvertiert. Anschließend wurden die Textdateien mit Hilfe des Vector Space Modells aus dem Bereich der Textanalyse in eine Vektordarstellung transformiert. Diese Vektordarstellung von Systemcall–Logdateien liefert eine gute Datengrundlage zur Anwendung maschineller Lernverfahren. Werden bei der Vektordarstellung nur Systemcalls berücksichtigt, ist die Menge der Merkmale begrenzt, so dass Lernverfahren genutzt werden können, die große Merkmalsmengen nicht handhaben können.

### Komponentenberechnung der Vektoren und die Ergebnisse verschiedener Lernverfahren

Die in einer Logdatei aufgezeichneten Systemcalls beschreiben die Merkmalsmenge. Die extrahierten Merkmale wiederum definieren die Dimensionen der Vektoren. Die einzelnen Vektorkomponenten können auf unterschiedliche Weise berechnet werden. Es stellte sich heraus, dass die Termfrequenz ($TF$) am besten geeignet ist. Die $TF$–Werte eines Threads geben die Häufigkeiten der Merkmale in diesem Beispiel an.

Die Stützvektormethode lieferte bei der Klassifikation von $TF$–Systemcallvektoren einzelner Threads mit 87,20% ± 0,27% bereits ein gutes Ergebnis, das sich durch das Lernen eines Modells auf Sequenz– und Mengenvektoren weiter verbesserte. Es wurden lediglich Sequenzen und Mengen untersucht, die durch eine Fenstergröße von maximal 6 Elementen erstellt wurden, da die Laufzeit des Verfahrens bei diesen Merkmalsmengen stark anstieg und eine weitere Vergrößerung der Fenster somit nicht praktikabel war.

Mengenvektoren scheinen sich besser zur Klassifikation der Schadprogramm–Familien zu eignen. Diese Darstellung liefert in den meisten Fällen verglichen mit den Sequenzen ähnliche oder bessere Ergebnisse bezüglich der Accuracy und des Class Recalls und die SVM benötigt weniger Rechenzeit zur Erstellung eines Modells. Leider geben die Experimente keinen eindeutigen Hinweis auf eine zu bevorzugende Sequenzlänge oder eine Mengengröße. Gute Ergebnisse im Hinblick auf die Accuracy und die Laufzeit des Verfahrens in der Lernphase und in der Testphase liefert die SVM bei der Klassifikation von $TF$–Vektoren, erstellt auf der Grundlage von 3er–Mengen.

Neben der SVM wurden ein $k$NN–Verfahren, ein Verfahren zur Erstellung einer Entscheidungstabelle (Regeln), ein Graph–Lerner (Bayes–Klassifikation) und ein Entscheidungsbaum–Lerner auf die $TF$–Systemcallvektoren der Logdateien angewendet. Aufgrund der Beliebtheit der SVM in vielen Bereichen des Data Minings und im Bereich der Intrusion Detection überraschten die guten Performanzen dieser Verfahren. Sie liegen bei bis zu 93% Genauigkeit und damit unter Berücksichtigung der Accuracy–Varianzen um bis zu 5,69% über dem Ergebnis der SVM bei der Verwendung derselben Datengrundlage aus Systemcallvektoren. Auch der Class Recall lag für einige Schadprogramm–Familien erheblich über dem der SVM. Die Genauigkeit des SVM–Modells näherte sich erst bei einer Datengrundlage aus Sequenz– und Mengenvektoren den Ergebnissen der übrigen vorgestellten Verfahren, jedoch unter erheblicher Verlängerung der Laufzeit.

### Bedeutung einzelner Systemcalls und der Parameterwerte

Durch die Betrachtung der Systemcalls, die das optimale Schema der Entscheidungstabelle definieren und der Systemcalls, die auf den ersten Ebenen des erstellten Entscheidungsbaums abgefragt und somit als wichtig erachtet wurden, konnte ein Einblick in die erstellten Modelle gewonnen werden. Die Anwendung verschiedener Gewichtungsverfahren ermöglichte die Extraktion weiterer Charakteristiken. Trotz unterschiedlicher Bewertungskriterien der verschiedenen Gewichtungsverfahren konnten einige Systemcalls identifiziert werden, die von verschiedenen Verfahren mit hohen Gewichten belegt wurden. Zum Beispiel wurden die Systemcalls `loadDll`, `openKey`, `sleep` und `writeValue` als wichtig zur Trennung der Schadprogramm–Familien angesehen.

Bei der Anwendung des Entscheidungsbaum–Lerners auf eine gewichtete, reduzierte Merkmalsmenge konnte die Laufzeit in der Lernphase unter geringen Accuracy–Einbußen erheblich verkürzt werden. Entgegen der in [YZF06] präsentierten Ergebnisse nahm die Accuracy der SVM bei einer Reduzierung der Merkmalsmenge aus Systemcalls ab und die Laufzeit des Algorithmus stieg.

Anders verhielt sich die SVM bei der Berücksichtigung eines Teils der Parameterwerte in der Merkmalsmenge. Einige Schadprogramm–Familien wurden anhand bestimmter Parameterwerte wesentlich besser erkannt. Zum Beispiel scheinen die Werte des Parameters `filename` des Systemcalls `loadDll` zur Erkennung des Trojaners *Virtumonde* wichtig zu

sein. Da der Systemcall `loadDll` Bibliotheken lädt, ist es zur Erkennung dieser Malware offensichtlich gut zu wissen, *welche* Bibliotheken geladen werden.

Die Laufzeit der SVM konnte durch die Einbeziehung bestimmter Parameterwerte gesenkt werden. Parameterwerte tragen demnach zu einer besseren Trennbarkeit aller Schadprogramm–Familien durch die SVM bei. Da sich die Menge der Merkmale bei der Einbeziehung von Parameterwerten stark erhöht, konnte die Güte der SVM lediglich auf einigen Merkmalsmengen bewertet werden, in denen *Teilmengen* der Parameterwerte berücksichtigt wurden. Andere Lernverfahren konnten aufgrund der großen Merkmalsmengen nicht getestet werden. Die Ergebnisse der SVM belegen jedoch den Informationsgehalt der Parameterwerte.

### Eine Familie für alle Threads eines Traces

Als Datengrundlage zur Klassifikation von Schadprogramm–Familien wurden die Threads der aufgezeichneten Prozesse extrahiert und klassifiziert. Dieses Verfahren ermöglicht die Familien–Zuordnung eines Angriffs bereits vor seiner Beendigung, so dass bereits frühzeitig passende Gegenmaßnahmen ergriffen werden können. Für die Threads eines Traces werden teilweise unterschiedliche Familien vorhergesagt. Um die Performanz der Klassifikationsverfahren zu verbessern, wurden die klassifizierten Threads eines Traces auf der Basis ihrer Familienkonfidenzen *einer* Familie zugeordnet. Durch diesen Nachbearbeitungsschritt wurde die Accuracy aller Modelle erhöht. Besonders auffällig war die Verbesserung des SVM–Modells, das auf Systemcallvektoren gelernt und auf die Testmenge angewendet wurde. Die Differenz betrug 9%, so dass dieses Modell nun ebenfalls eine Genauigkeit von über 90% erreichte. Bis auf einige wenige Ausnahmen verbesserte sich der Class Recall aller Modelle über alle Schadprogramm–Familien hinweg und dies zum Teil erheblich.

Das vorgestellte Verfahren schafft zudem die Basis für die Klassifikation ganzer Traces. Ausgehend von den Threads eines Traces, die *einer* Familie zugeordnet wurden, kann die Klassenzugehörigkeit der Traces als die Klasse angegeben werden, die für die Threads vorhergesagt wurde. Die Güte des Modells kann dann auch bezüglich der Traces berechnet werden. Dieser Schritt wurde beispielhaft für das Modell des Entscheidungsbaum–Lerners bei der Anwendung auf die Testdaten durchgeführt und ergab eine wesentlich höhere Genauigkeit als ein Entscheidungsbaum–Modell, das direkt auf den Traces erstellt wird.

## 6.1.1. Probleme der vorgestellten Ansätze

Die Annahme bei den in dieser Arbeit vorgestellten Ansätzen besteht darin, dass sich die Vektorkomponenten verschiedener Schadprogramm–Familien voneinander unterscheiden. Ist diese Voraussetzung nicht gegeben, können die Schadprogramme nicht klassifiziert werden.

Die präsentierten Ansätze liefern Modelle für die Zuordnung aufgezeichneter Systemcall–Threads zu einer Schadprogramm–Familie. Dabei wird die Güte der Modelle unter der Annahme berechnet, dass alle Threads eines Traces zu einer der Schadprogramm–Familien gehören. Malware kann jedoch durchaus Ausführungsstränge enthalten, die normales Verhalten darstellen. Solche Teilprozesse können nicht identifiziert werden. Es kann

also keine Differenzierung zwischen normalem und schadhaftem oder bekanntem und unbekanntem Verhalten vorgenommen werden. Die Übergabe normaler oder unbekannter Programmausführungen an eines der gelernten Modelle stellt somit ein Problem dar, da auch diese Beispiele fälschlicherweise einer der Schadprogramm–Familien zugeordnet werden.

Ein weiteres Problem ist die Tatsache, dass zunächst genügend aufgezeichnete Systemcall–Traces der Programm–Ausführungen einer Schadprogramm–Familie vorliegen müssen, um auf diesen Daten ein Modell lernen zu können. Damit ein gutes Modell erstellt werden kann, sollten die aufgezeichneten Programm–Ausführungen zudem ausschließlich charakteristische Threads der Familie enthalten. Zur Erweiterung eines bestehenden Modells um neue Schadprogramm–Familien muss das Modell neu trainiert werden.

## 6.2. Ausblick

Die Analyse von Betriebssystem–Logdateien in der Intrusion Detection zur Klassifikation von Schadprogramm–Familien ist ein umfangreiches Forschungsgebiet mit einer Vielzahl möglicher Analysemethoden und Vorgehensweisen. Aufgrund des beschränkten Zeitrahmens der Diplomarbeit wurden nur einige Teil–Aspekte des Problems detailliert betrachtet. Dieser Abschnitt beschreibt einige Evaluierungen und Erweiterungen der vorgestellten Ansätze, die ergänzende Erkenntnisse und Verbesserungen liefern könnten.

### 6.2.1. Weitere Evaluierungen

Es besteht die Möglichkeit einer Vielzahl weiterer Bewertungen, die auf den vorliegenden Daten durchgeführt werden können. In diesem Abschnitt sollen nur einige Ideen vorgestellt werden.

#### Kontextabhängige Fenstergrößen

In Abschnitt 5.5.2 wurde die Lernbarkeit der SVM auf Systemcall–Mengen und Systemcall–Sequenzen bewertet. Zur Erstellung der Merkmalsmengen wurden Fenster verschiedener Größen über die Daten geschoben. Dabei lieferten die Experimente bezüglich der verwendeten Fenstergrößen unterschiedliche Ergebnisse für die Schadprogramm–Familien und gaben keinen Hinweis auf eine für alle Familien zu bevorzugende Fenstergröße. Eine auf der Entropie basierende Auswahl der optimalen Fenstergröße, wie in [ESL01] vorgeschlagen, hätte somit wahrscheinlich keine signifikanten Verbesserungen der Modelle zur Folge.

In [ESL01] wurde allerdings auch eine kontextabhängige Wahl der Fenstergröße vorgestellt. Dabei werden vorangegangene Subsequenzen einer Systemcallsequenz zur Vorhersage eines Systemcalls ermittelt. Dieses Verfahren könnte auch bei der Klassifikation von Schadprogramm–Familien zu einer Verbesserung der Ergebnisse führen.

#### Gruppierung von Parameterwerten

In Abschnitt 5.5.4 wurde versucht, die Bedeutung einzelner Systemcalls zur Lernbarkeit von Schadprogramm–Familien anhand einiger ausgewählter Kriterien zu ermitteln. Die

Ergebnisse wurden in dem Abschnitt 5.5.5 zur Auswahl von Systemcalls genutzt, deren Parameterwerte in der Merkmalsmenge berücksichtigt werden sollten. Eine Auswahl anhand anderer Kriterien ist natürlich ebenfalls möglich.

Ein Systemcall bildet mit jedem Wert der einzubeziehenden Parameter ein Merkmal. Da bei dieser Form der Darstellung bereits kleinste Abweichungen eines Parameterwertes zu einem weiteren Merkmal führen, werden die Merkmalsmengen sehr schnell immens groß. Daher wurden nur Teilmengen der Parameterwerte berücksichtigt. Um das enorme Wachstum der Merkmalsmenge zu reduzieren ohne einen großen Informationsverlust zu erleiden, könnte eine passende Abstraktion der Parameterwerte erfolgen. Dabei kann ähnlich wie bei den durch das Tool *LibAnomaly* implementierten Modellen vorgegangen werden. Diese Modelle wurden in [Zan06] und [MVKV06] (siehe Abschnitt 3.6) bereits verwendet. Ein Parameterwert könnte demnach aufgrund bestimmter Eigenschaften wie der Pfadtiefe, Dateiendungen, der Buchstabenverteilung oder möglicherweise aussagekräftiger Verzeichniszugriffe charakterisiert und abstrahiert werden. Dabei kann angenommen werden, dass bestimmte Abweichungen in den Parameterwerten für die Klassifikation eines Schadprogramms irrelevant sind.

Die Parameterwerte würden folglich aufgrund definierter Ähnlichkeitsmaße in Gruppen eingeteilt. Kleine Abweichungen der Werte führen somit nicht mehr zu neuen Merkmalen. Ein Merkmal würde eine Gruppe und nicht eine spezielle Parameterausprägung repräsentieren. Die Menge der Merkmale könnte stark reduziert werden, so dass die Berücksichtigung aller Parameterwerte oder wenigstens einer größeren Anzahl an Parameterwerten möglich wäre.

### Berücksichtigung der Baumstruktur

In Abschnitt 5.1 wurde die Datengrundlage dieser Arbeit beschrieben. Die von einem Schadprogramm während seiner Ausführung aufgerufenen und aufgezeichneten Systemcalls liegen in der Form von XML–Logdateien vor. Die implizite Baumstruktur dieser Dateien ist in der Abbildung B.2 des Anhangs B dargestellt. Diese hierarchische Struktur wurde im Rahmen der vorgestellten Ansätze nur insofern berücksichtigt, als dass die Threads als eine Ebene dieser Baumstruktur extrahiert wurden. Um die gesamte Baumstruktur einer Logdatei zu berücksichtigen, könnte zum Beispiel die Stützvektormethode mit dem in Abschnitt 4.4.2 vorgestellten Kern für Baumstrukturen angewendet werden.

### Bewertung der Unterscheidbarkeit von Schadprogramm–Familien

Die Erkennungsraten der gegebenen Schadprogramm–Familien sind divergent. Während einige Familien von den meisten Lernverfahren gut klassifiziert werden, wird bei anderen eine große Anzahl der gegebenen Beispiele einer falschen Familie zugeschrieben. Ob diese falschen Zuordnungen aufgrund ähnlicher Verhaltensweisen der Malware vielleicht sogar Sinn machen, könnte durch einen Vergleich dieser Verhaltensweisen bewertet werden.

Zum Beispiel ordnen die erstellten Modelle einen großen Anteil der Threads der Familie *Sdbot* der Familie *Rbot* zu. Diese Falschklassifizierung ist nach dem Vergleich der Virenbeschreibungen, die der Antivirussoftware–Hersteller *Avira Antivir* im Internet zur Verfügung stellt[21], nachvollziehbar. Die Beschreibungen sind vollkommen identisch. Die

[21] http://www.avira.com/de/threats/index.html

Schadprogramm–Familien nehmen Änderungen an der Registry vor, machen sich die Verwundbarkeit von Software zu Nutze und ermöglichen den unbefugten Zugriff auf den Computer.

Neben *Avira Antivir* gibt es eine Vielzahl anderer Antivirussoftware–Hersteller, deren Malware–Beschreibungen für derartige Analysen verwendet werden könnten.

### 6.2.2. Modell–Erweiterungen

In dieser Arbeit wurden Modelle zur Klassifikation von Threads verschiedener Schadprogramm–Familien erstellt. Die Threads wurden aus den Logdateien der Programmausführungen unterschiedlicher Malware extrahiert. Im Folgenden werden zwei Ansätze vorgestellt, die interessante Erweiterungen der Modelle darstellen könnten.

#### Klassifikation von Traces

In Abschnitt 5.5.6 wurde bereits ein Ansatz vorgestellt, der unter anderem die Basis für die Zuordnung eines kompletten Traces aufgrund der klassifizierten Threads liefert. Dazu werden *alle Threads* eines Traces *einer* Schadprogramm–Familie zugeschrieben. Um die gelernten Modelle bezüglich der Klassifikation von *Traces* bewerten zu können, müssen die Gütekriterien neu berechnet werden. Ist das geschehen, können die Ergebnisse mit den Modellen verglichen werden, die direkt auf ganzen Traces gelernt wurden, für die also keine Extraktion der Threads erfolgte. Dieser Schritt wurde in Abschnitt 5.5.6 beispielhaft für eines der erstellten Modelle durchgeführt und bedarf einer Automatisierung.

Weiteres Verbesserungspotenzial bezüglich der Modelle zur Klassifikation von Traces und von Threads könnte in der Anpassung der in Abschnitt 5.5.6 in der Gleichung (5.2) vorgestellten Funktion liegen.

Eine weitere Möglichkeit zur Klassifikation der Traces durch Teilsequenzen der aufgezeichneten Systemcall–Folgen könnte die Extraktion der Systemcalls einzelner *Prozesse* sein. Ein Trace kann mehrere Prozesse enthalten, die wiederum mehrere *Threads* ausführen können. Die Prozess–Darstellung der Logdateien ist demnach gröber als die Darstellung durch einzelne Threads. Die Prozesse könnten auf ähnliche Weise wie die Threads extrahiert und den ursprünglichen Logdateien später wieder zugeordnet werden. So kann auch hier eine Klassifikation der Traces erfolgen und die Performanzen der Modelle können mit denen anderer Verfahren verglichen werden.

#### Identifizierung unbekannten Verhaltens

In Abschnitt 6.1.1 wurde bereits darauf hingewiesen, dass die in dieser Arbeit erstellten Modelle unbekanntes Verhalten nicht als solches identifizieren können. Bei der Klassifikation von Schadprogrammen kann "unbekanntes" Verhalten in Programm–Ausführungen nicht trainierter Schadprogramm–Familien bestehen oder in Sequenzen normaler Systemcallaufrufe. Damit solche Beispiele auch als unbekannt identifiziert werden, könnte ähnlich wie in [RHW+08] (siehe Abschnitt 3.7.2) ein Schwellwert genutzt werden. Eine denkbare Vorgehensweise wäre, die Threads, deren Familienkonfidenz für alle Schadprogramm–Familien unter diesem definierten Schwellwert liegt, als unbekannt zu klassifizieren. Die

Klassifikation ganzer Traces könnte dann zum Beispiel wieder durch die oben beschriebene Prozedur der Wahl einer Familie für alle Threads erfolgen oder durch ein einfaches Zählen maliziöser Threads und erneuter Anwendung eines Schwellwerts.

### 6.2.3. Alternative Datensätze

Die Experimente zur Klassifikation von Schadprogramm–Familien wurden lediglich auf der Grundlage der in Abschnitt 5.1 beschriebenen Systemcall–Berichte des LS6 durchgeführt. Für eine eingehendere Bewertung der Klassifikations– und Generalisierungsfähigkeit der verwendeten Lernverfahren sollten diese Verfahren auf ähnlichen Datensätzen getestet werden.

**Einheitliche Label** Die Labelung der in dieser Arbeit verwendeten Systemcall–Berichte des LS6 erfolgte durch den Upload der Binärdateien der Schadprogramme zu *Virustotal*. Wie in Abschnitt 5.1 beschrieben wurde, stehen somit die Label unterschiedlicher Anti–Virus–Engines zur Verfügung, die häufig verschiedene Namen für dasselbe Schadprogramm verwenden. Bei dem verwendeten Datensatz wurde der am häufigsten vergebene Schadprogramm–Name als Label für die mit dem Programm erzeugten Logdateien vergeben. Es wurde bereits darauf hingewiesen, dass die vorliegenden Label folglich nicht zwingend eindeutig sind. Das bedeutet wiederum, dass bei Falschklassifikationen der Logdateien nicht sicher ist, ob es sich tatsächliche um einen Fehler handelt oder ob lediglich das Label nicht richtig gewählt wurde. Zur zweifelsfreieren Bewertung der in Abschnitt 5.5 vorgestellten Verfahren könnten diese auf einen Datensatz angewendet werden, in dem die Beispiele mit den Labeln *eines* Antivirussoftware–Herstellers versehen werden. So können Falschklassifikationen aufgrund unterschiedlicher Namensgebungen ausgeschlossen werden. Es wird vermutet, dass sich die Ergebnisse der durchgeführten Experimente auf der Grundlage der so gelabelten Daten verbessern.

**DARPA–Datensatz** Um eine detailliertere Vergleichbarkeit der Ergebnisse dieser Diplomarbeit mit den Ergebnissen anderer Arbeiten im Bereich der Klassifikation von Schadprogramm–Familien zu ermöglichen, ist die Anwendung der vorgestellten Verfahren und Darstellungsformen auf Datensätze sinnvoll, die in anderen Arbeiten verwendet wurden. In Abschnitt 5.1 wurde die Nutzung der in [RHW+08] verwendeten Daten bereits ausgeschlossen, da diese Daten nur in der von Rieck et. al. gewählten und hier in Abschnitt 3.7.2 beschriebenen, konvertierten Form vorliegen.

In [KAT07] wurde, wie auch in vielen anderen Arbeiten im Bereich der *Angriffserkennung*, der DARPA–Datensatz des MIT LL verwendet. Eine Bewertung der in dieser Diplomarbeit vorgestellten Verfahren und Datendarstellungen auf diesem Datensatz würde die Möglichkeit eines direkten Vergleichs mit den Ergebnissen aus [KAT07] bieten.

## 6.3. Fazit

Die immer schneller werdende Verbreitung von Viren, Würmern und anderer Malware und die Vielzahl verschiedener Schadprogramme sowie deren kontinuierliche Veränderung erfordern eine ständige Aktualisierung von Anti–Malware–Produkten und die Bewahrung

eines Überblicks über die aktuelle Bedrohung, um die Sicherheit von Computersystemen zu gewährleisten. Schadprogramme, die zum Beispiel aufgrund von Code–Änderungen zunächst als andersartig und unbekannt betrachtet werden, können aufgrund ähnlicher Verhaltensweisen vielfach bekannten Gruppen, so genannten "Schadprogramm–Familien", zugeordnet werden. Dadurch ist zum einen die Erkennung einer Malware möglich, ohne dass eine Signatur vorliegen muss. Zum anderen ist die Familien–Zuordnung wichtig, da es zur Ergreifung passender Gegenmaßnahmen häufig sehr hilfreich oder sogar notwendig ist zu wissen, *welche* Malware vorliegt und nicht nur, dass ein Angriff stattgefunden hat.

Anders als bei dem Großteil der Ansätze im Bereich der Intrusion Detection war das Ziel dieser Diplomarbeit daher nicht die *Erkennung* von Angriffen, sondern die Analyse von Systemcall–Logdateien, von denen bereits bekannt ist, dass sie bösartiges Verhalten beschreiben. Der Prozess der *Schadprogramm–Klassifikation* sollte automatisiert werden. Die Logdateien wurden dazu in eine passende Darstellung konvertiert. Es wurden unterschiedliche Merkmalsmengen genutzt, auf die maschinelle Lernverfahren angewendet wurden. Bereits auf der Basis der einfachsten getesteten Merkmalsmenge einzelner Systemcalls ohne die Berücksichtigung von Parameternamen und –werten konnten gute Klassifikations–Modelle erstellt werden. Die gewählten Darstellungsformen und die maschinellen Lernverfahren scheinen sich folglich gut zur Klassifikation der Schadprogramm–Familien zu eignen.

In dem vorliegenden Kapitel wurden bereits einige Ideen zu Erweiterungen und Verbesserungen der in dieser Diplomarbeit beschriebenen Ansätze vorgestellt. Das Gebiet der Klassifikation von Schadprogramm–Familien mit der Hilfe von Data Mining Techniken bietet noch eine Vielzahl weiterer Ansatzpunkte zur Automatisierung und zur Verbesserung des Schutzes von Computersystemen.

# A. Implementierungsdetails der Parser

## Beispiel eines *CWSandbox*–Berichtes

```
<?xml version="1.0"?>
<!-- This analysis was created by CWSandbox (c)...-->
<analysis cwsversion="2.0.42" time=... />
<calltree>
<process_call index="1" pid="1644" filename=.../>
...
</calltree>

<processes>
<process index="1" pid="1644" filename=...>
<thread tid="716">
<all_section>
...
<load_dll filename="WS2_32.dll" successful="1" address="&#x24;71A10000"
size="94208" filename_hash="hash_error"/>
<open_key key="HKEY_LOCAL_MACHINE\Software\Microsoft\Advanced INF Setup"/>
...
</all_section>

</thread>
</process>
</processes>

<running_processes>
<running_process pid="0" filename="&#x28;SystemIdle&#x29;"/>
<running_process pid="4"... />
</running_processes>
</analysis>
```

Abbildung A.1.: Auszug eines mit *CWSandbox* erzeugten XML–Berichtes. Aus Darstellungsgründen wurden zusätzliche Zeilenumbrüche eingefügt und Header–Informationen sowie Textteile ausgelassen. Auslassungen sind durch "..." gekennzeichnet.

## Parser

Die implementierten Parser zur Konvertierung einer XML–Logdatei, wie sie beispielhaft in Abbildung A.1 dargestellt ist, in eine Textdarstellung nutzen das *SAX API* [22] und berücksichtigen nur den Teil der XML–Logdatei, der zwischen den Tags `<processes>` und `</processes>` steht. Die übrigen Informationen dienen lediglich der Angabe der *CWSandbox*–Version, der Startzeit des Prozesses in *CWSandbox* etc. und können daher ignoriert werden. Der Tag "`<processes>`" kennzeichnet den Beginn eines Prozesses, der Tag `<thread>` den Beginn eines Threads. Die Systemcallaufrufe eines Schadprogramms innerhalb seiner Threads sind in der XML–Datei dokumentiert.

Systemcalls, die in der XML–Datei Unterstriche enthalten werden in der Textdatei ohne diese Unterstriche gespeichert. Der Buchstabe *nach* dem Unterstrich wird zur besseren Lesbarkeit groß geschrieben. So wird aus `open_file` zum Beispiel `openFile`.

### Berücksichtigung aller oder keiner Parameter

Bei dem Aufruf des Parsers kann angegeben werden, ob Parameternamen und/oder Parameterwerte aus dem XML–Format übernommen werden sollen oder ob die erzeugte Textdatei lediglich die Systemcalls beinhalten soll.

Bei der Angabe der Option `-p` werden alle Parameter*namen* übernommen. Bei der Angabe der Option `-v` werden alle Parameter*werte* übernommen. Die Parameternamen oder –werte werden durch Kommata separiert in einer Klammer hinter dem Systemcall angegeben. Natürlich können auch beide Optionen zugleich verwendet werden, so dass alle Parameternamen *und* –werte in das Textformat konvertiert werden. In diesem Fall werden die Angaben in der Form "`Parameter=Wert`" ebenfalls durch Kommata separiert in einer Klammer hinter dem Systemcall angegeben.

### Selektive Auswahl von Parametern

Die grundlegende Implementierung dieses Parsers bleibt unverändert. Der Parser bietet jedoch anstelle der Optionen `-p` und `-v` die Möglichkeit zur Angabe einer *Properties–Datei*. Die Datei muss Systemcall–Parameter–Paare enthalten, die vom Parser als Schlüssel–Wert–Paare interpretiert werden. Wird beispielsweise eine Properties–Datei mit dem Eintrag

"`open_file srcfile`"

übergeben, wird für den Systemcall `open_file` der Parameter `srcfile` mit seinen Werten in die Textdarstellung übernommen. Für alle anderen Systemcalls, für die in dieser Datei kein Parameter spezifiziert wird, werden Parameternamen und –werte *nicht* konvertiert. Die Abbildung 5.6 in Abschnitt 5.2 zeigt einen Auszug der so erstellten Textdatei, wenn eine Properties–Datei mit dem oben beispielhaft angegebenen Inhalt übergeben wird.

Wird anstelle des Parameters (in dem obigen Beispiel "`srcfile`") ein "`a`" angegeben, also zum Beispiel "`open_file a`", werden *alle* Parameternamen und –werte des entsprechenden Systemcalls in die Textdarstellung übernommen.

[22] SAX definiert eine einfache Programmierschnittstelle für XML. Das XML–Dokument wird sequentiell, ereignisgesteuert durchlaufen und so gelesen. Für weitere Details siehe http://www.saxproject.org/

# B. Grafiken

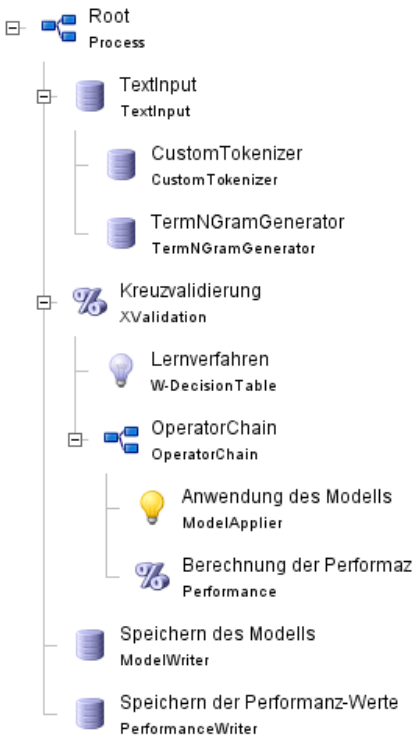



Abbildung B.1.: Beispielhafter Aufbau eines Experiments in RapidMiner. Nach dem Einlesen der Daten (TextInput) erfolgt die Merkmalsextraktion (CustomTokenizer) und gegebenenfalls die Erstellung von Sequenzen oder Mengen (TermNGramGenerator). Aufgrund dieser Merkmale erfolgt die Berechnung der Vektoren. Die Kreuzvalidierung zerlegt den eingelesenen Datensatz in eine Trainings– und eine Testmenge. Die Anzahl der Wiederholungen dieses Vorgangs kann vom Nutzer festgelegt werden. Für jede Trainingsmenge wird durch das ausgewählte Lernverfahren ein Modell erstellt und dessen Performanz wird durch die Testmenge bewertet. Das Gesamtmodell wird auf allen Daten des Lerndatensatzes erstellt und gespeichert. Die Accuracy des Modells wird als Durchschnitt der Einzel–Performanzen der Modelle berechnet, die im Zuge der Kreuzvalidierung erstellt wurden und kann ebenfalls gespeichert werden.

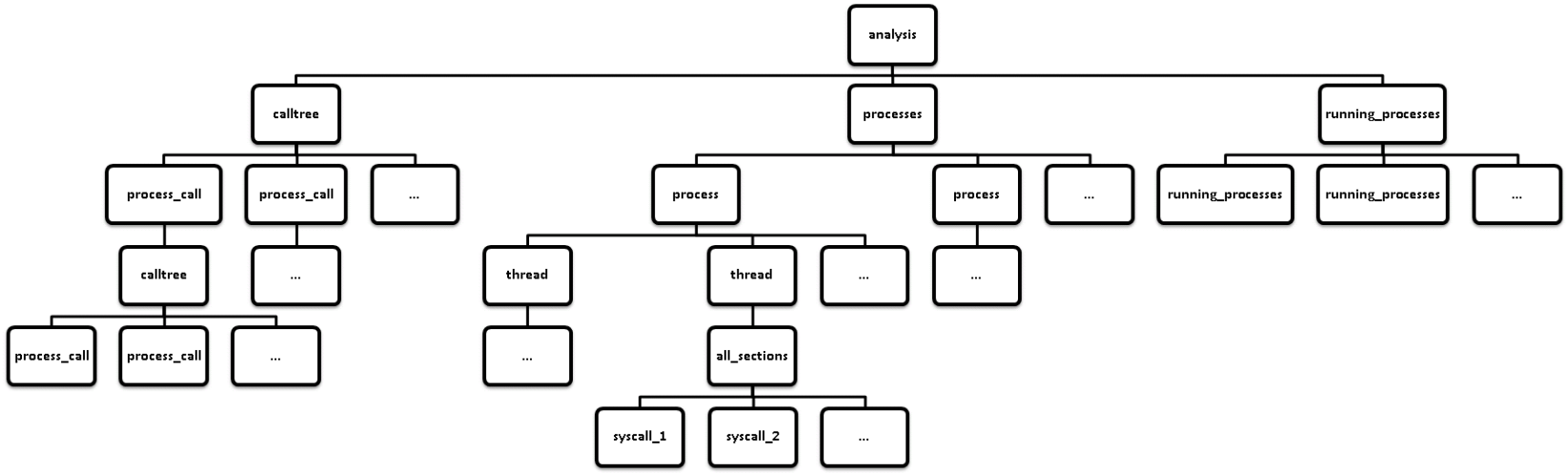



Abbildung B.2.: Die baumartigen Struktur der XML–Dateien des LS6. "..." bedeutet, dass der Elter 1 bis $n$ der auf dieser Ebene angegebenen Kinder haben kann.

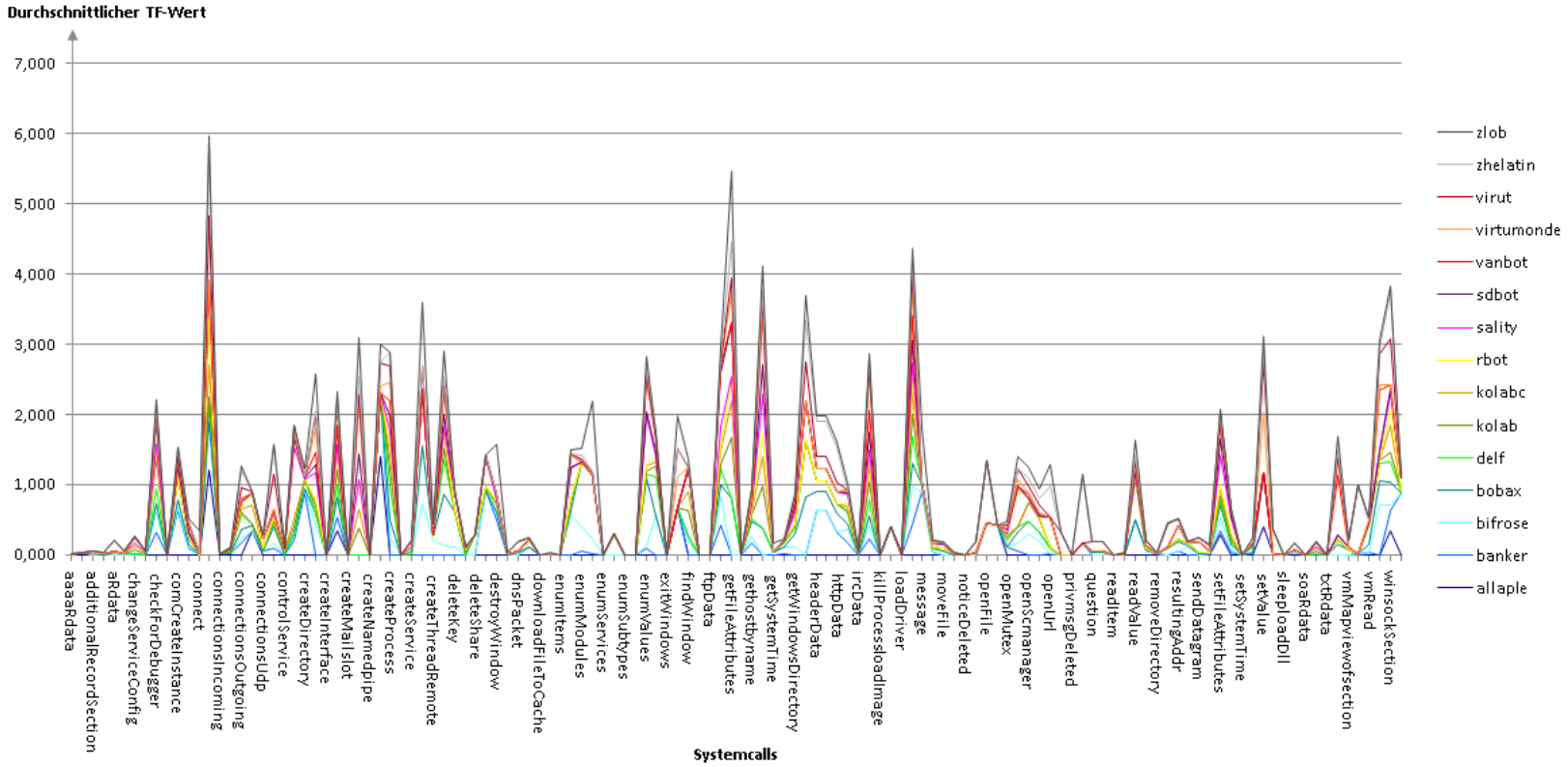



Diagramm B.3.: Durchschnittliche $TF$–Werte jedes Systemcalls pro Schadprogramm–Familie im Lerndatensatz — vergrößerte Darstellung. Aus Darstellungsgründen werden auf der x–Achse nicht alle Beschriftungen angezeigt.

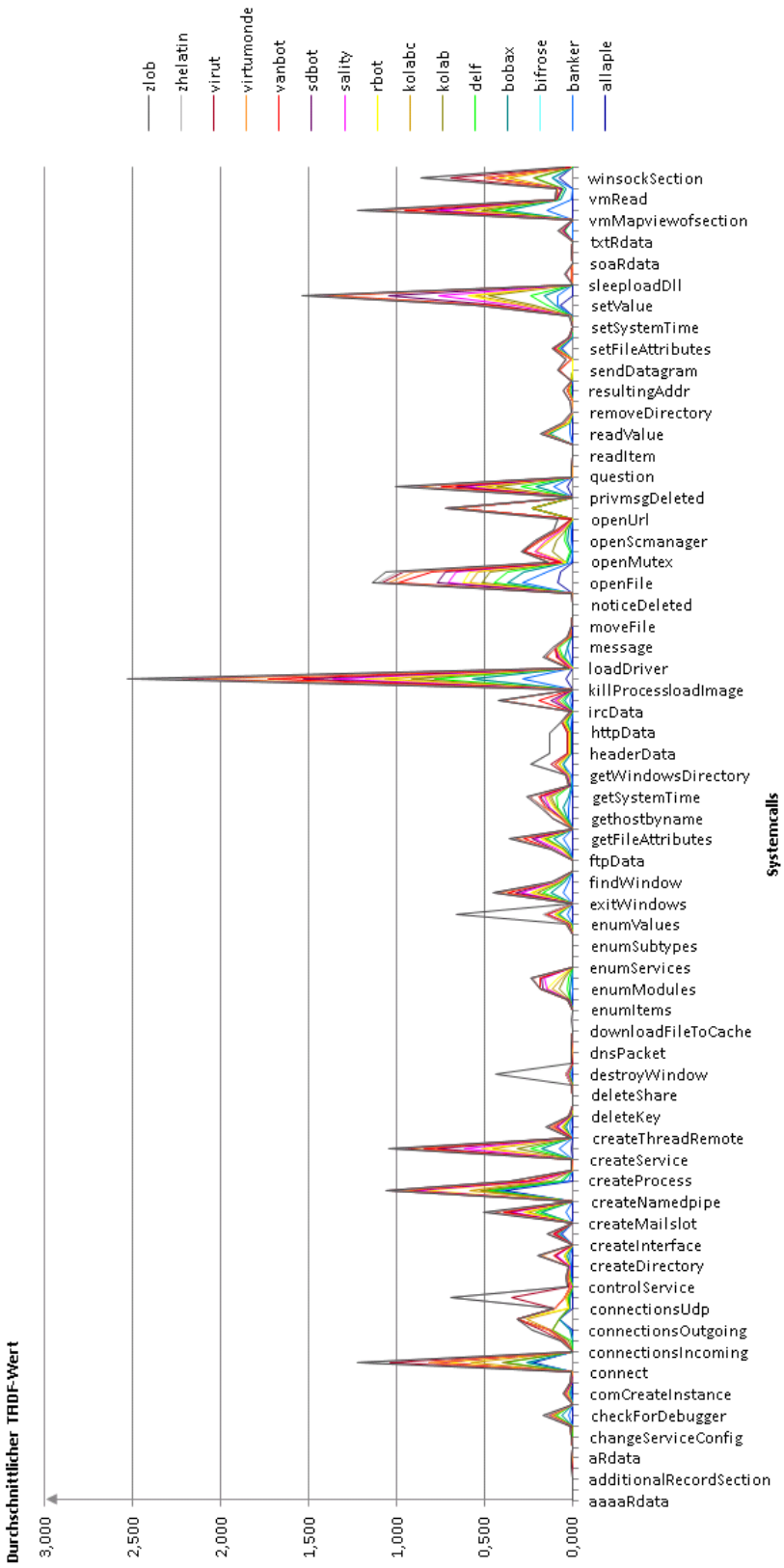



Diagramm B.4.: Durchschnittliche $TFIDF$-Werte der Systemcalls pro Schadprogramm–Familie im Lerndatensatz — vergrößerte Darstellung. Aus Darstellungsgründen werden auf der x–Achse nicht alle Beschriftungen angezeigt.

# Literaturverzeichnis


[AK91] Aha, David W. und Dennis Kibler: *Instance-Based Learning Algorithms.* Machine Learning, 6:37–66, 1991.

[Bis02] Bishop, Matt: *Computer Security: Art and Science.* Addison-Wesley Professional, 2002.

[Bou04] Bouckaert, Remco R.: *Bayesian Network Classifiers in Weka.* http://www.cs.waikato.ac.nz/~ml/publications/2004/uow-cs-wp-2004-14.pdf, 2004.

[BSI02] BSI, Bundesamt für Sicherheit in der Informationstechnik: *BSI-Leitfaden zur Einführung von Intrusion-Detection-Systemen.* http://www.bsi.de/literat/studien/ids02/index.htm, 10 2002. 11.08.2009.

[BSI08] BSI: *Internet–Lagebild Erstes Quartal 2008.* Technischer Bericht, Bundesamt für Sicherheit in der Informationstechnik, 2008.

[Bur98] Burges, Christopher J. C.: *A Tutorial on Support Vector Machines for Pattern Recognition.* Data Mining and Knowledge Discovery, 1998.

[CD01] Collins, Michael und Nigel Duffy: *Convolution Kernels for Natural Language.* In: *Advances in Neural Information Processing Systems 14*, Seiten 625–632. MIT Press, 2001.

[Coh95] Cohen, William W.: *Fast Effective Rule Induction.* In: *Proceedings of the 12th International Conference on Machine Learning*, Seiten 115–123. Morgan Kaufmann, 1995.

[CS01] Crammer, Koby und Yoram Singer: *On the Algorithmic Implementation of Multiclass Kernel-based Vector Machines.* Journal of Machine Learning Research, 2:265–292, 2001.

[CV95] Cortes, Corinna und Vladimir Vapnik: *Support–Vector Networks.* Machine Learning, 20(3):273–297, 1995.

[ESL01] Eskin, Eleazar, Salvatore J. Stolfo und Wenke Lee: *Modeling System Calls for Intrusion Detection with Dynamic Window Sizes.* DARPA Information Survivability Conference and Exposition (DISCEX), Seiten 165–175, 2001.

[FHSL96] Forrest, Stephanie, Steven A. Hofmeyr, Anil Somayaji und Thomas A. Longstaff: *A sense of self for unix processes.* In: *Proceedings of*

*the 1996 IEEE Symposium on Security and Privacy*, Seiten 120–128. IEEE Computer Society Press, 1996.

[FPsS96] Fayyad, Usama, Gregory Piatetsky-shapiro und Padhraic Smyth: *From Data Mining to Knowledge Discovery in Databases.* AI Magazine, 17:37–54, 1996.

[GGM08] Golovanov, Sergey, Alexander Gostev und Denis Maslennikov: *Kaspersky Security Bulletin 2008. Entwicklung der IT-Bedrohungen im ersten Halbjahr 2008.* http://www.viruslist.com/de/analysis?pubid=200883626#2, 9 2008. 03.08.2009.

[GWBV02] Guyon, Isabelle, Jason Weston, Stephen Barnhill und Vladimir Vapnik: *Gene Selection for Cancer Classification using Support Vector Machines.* Machine Learning, 46(1-3):389–422, 2002.

[HCL09] Hsu, Chih–Wei, Chi–Chung Chang und Chih–Jen Lin: *A Practical Guide to Support Vector Classification.* http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf, 5 2009.

[Hei08] Heise: *Umfrage: Fast 4 Millionen deutsche Opfer von Computer–Kriminalität.* http://www.heise.de/security/Umfrage-Fast-4-Millionen-deutsche-Opfer-von-Computer-Kriminalitaet--/news/meldung/110494, 7 2008. 21.01.2009.

[HLSM90] Heady, Richard, George Lugar, Mark Servilla und Arthur Maccabe: *The Architecture of a Network Level Intrusion Detection System.* Technischer Bericht, University of New Mexico, Albuquerque, NM, 8 1990.

[IDS97] IDSG, Intrusion Detection Subgroup: *Report on the NS/EP Implications of Intrusion Detection Technology Research and Development.* Technischer Bericht, National Security Telecommunications Advisory Committee, Dezember 1997.

[KAT07] Khan, Latifur, Mamoun Awad und Bhavani Thuraisingham: *A new intrusion detection system using support vector machines and hierarchical clustering.* The International Journal on Very Large Data Bases (VLDB Journal), 16(4):507–521, 2007.

[KAW94] Kephart, Jeffrey O., William C. Arnold und Thomas J. Watson: *Automatic Extraction of Computer Virus Signatures.* In: *Proceedings of the 4th Virus Bulletin International Conference*, Seiten 178–184, 1994.

[KFH05] Kang, Dae–Ki, Doug Fuller und Vasant Honavar: *Learning Classifiers for Misuse Detection Using a Bag of System Calls Representation.* In: *Information Assurance Workshop, 2005. IAW '05. Proceedings from the Sixth Annual IEEE SMC*, Seiten 118–125, 2005.

[KMVV03] Kruegel, Christopher, Darren Mutz, Fredrik Valeur und Giovanni Vigna: *On the Detection of Anomalous System Call Arguments.* In: *Computer Security ESORICS 2003*, LNCS, Seiten 326–343. Springer, 2003.

[Koh95] Kohavi, Ron: *The Power of Decision Tables.* In: *Proceedings of the European Conference on Machine Learning*, Seiten 174–189, London, UK, 1995. Springer Verlag.

[LS98] Lee, Wenke und Salvatore J. Stolfo: *Data mining approaches for intrusion detection.* In: *Proceedings of the 7th USENIX Security Symposium*, 1998.

[LV02] Liao, Yihua und V. Rao Vemuri: *Use of K–Nearest Neighbor classifier for intrusion detection.* Computer Science, 21(5):439 – 448, 2002.

[Mie06] Mierswa, Ingo: *Making Indefinite Kernel Learning Practical.* Technischer Bericht, Collaborative Research Center 475, University of Dortmund, 2006.

[MN98] McCallum, Andrew und Kamal Nigam: *A Comparison of Event Models for Naive Bayes Text Classification.* In: *AAAI Workshop on "Learning for Text Categorization"*, 1998.

[MVKV06] Mutz, Darren, Fredrik Valeur, Christopher Kruegel und Giovanni Vigna: *Anomalous system call detection.* ACM Transactions on Information and System Security, 9:61–93, 2006.

[Paw91] Pawlak, Zdzislaw: *Rough Sets - Theoretical Aspects of Reasoning about Data.* Kluwer Academic Publisher, Dordrecht, The Netherlands, 1991.

[Paw98] Pawlak, Zdzislaw: *Rough Set Theory and its Applications to Data Analysis.* Cybernetics and Systems, 29(7):661–688, 1998.

[Pla98] Platt, John C.: *Machines using Sequential Minimal Optimization.* In: Schoelkopf, B., C. Burges und A. Smola (Herausgeber): *Advances in Kernel Methods – Support Vector Learning*, Seiten 185–208. MIT Press, Cambridge, MA, USA, 1998.

[PTL00] Polkowski, Lech, Shusaku Tsumoto und Tsau Y. Lin: *Rough Set Methods and Applications. New Developments in Knowledge Discovery in Information Systems.* Physica-Verlag, Heidelberg, 2000.

[Qui86] Quinlan, J. Ross: *Induction of Decision Trees.* Machine Learning, 1(1):81–106, 03 1986.

[Qui93] Quinlan, J. Ross: *C4.5: Programs for Machine Learning.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[RHW+08] Rieck, Konrad, Thorsten Holz, Carsten Willems, Patrick Düssel und Pavel Laskov: *Learning and Classification of Malware Behaviour.* In: *Detection of Intrusions and Malware, and Vulnerability Assessment*, Seiten 108–125. Springer Berlin/Heidelberg, 2008.

[Slo92] Slowinski, Roman: *Intelligent Decision Support: Handbook of Applications and Advances of the Rough Sets Theory.* Kluwer Academic Publishers, Dordrecht, The Netherlands, 1992.

[SSWB98] Schölkopf, Bernhard, Alex J. Smola, Robert Williamson und Peter Bartlett: *New Support Vector Algorithms*, 1998.

[SWY75] Salton, G., A. Wong und C.S. Yang: *A Vector Space Model for Automated Indexing.* Communications of the ACM, 18(11):613 – 620, 1975.

[TFJ+08] Turner, Dean, Marc Fossi, Eric Johnson, Trevor Mack, Joseph Blackbird, Stephen Entwisle, Mo King Low, David McKinney und Candid Wueest: *Symantec Global Internet Security Threat Report – Trends for July-December 07.* Technischer Bericht, Symantec, April 2008.

[TJH+05] Tsochantaridis, Ioannis, Thorsten Joachims, Thomas Hofmann, Yasemin Altun und Yoram Singer: *Large margin methods for structured and interdependent output variables.* Journal of Machine Learning Research, 6:1453–1484, 2005.

[Vap95] Vapnik, Vladimir N.: *The nature of statistical learning theory.* Springer-Verlag New York, Inc., New York, NY, USA, 1995.

[Web99] Webb, Geoffrey I.: *Decision Tree Grafting From the All Tests But One Partition.* In: *IJCAI*, Band 18, Seiten 702–707. Morgan Kaufmann, 1999.

[YZF06] Yao, JingTao, Songlun Zhao und Lisa Fan: *An Enhanced Support Vector Machine Model for Intrusion Detection.* In: *Rough Sets and Knowledge Technology*, LNCS, Seiten 538–543. Springer, 2006.

[Zan06] Zanero, Stefano: *Unsupervised Learning Algorithms for Intrusion Detection.* Doktorarbeit, Politecnico di Milano – Dipartimento di Elettronica e Informazione, 2006.

[ZZJZ07] Zhou, Guodong, Min Zhang, Donghong Ji und Qiaoming Zhu: *Tree Kernel-Based Relation Extraction with Context-Sensitive Structured Parse Tree Information.* In: *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL)*, Seiten 728–736, 2007.

# Erklärung

Hiermit erkläre ich, Andrea Matuszewski, die vorliegende Diplomarbeit mit dem Titel

**Analyse von Betriebssystem-Logdateien**

selbständig verfasst und keine anderen als die hier angegebenen Hilfsmittel verwendet, sowie Zitate kenntlich gemacht zu haben.

Dortmund, 17. September 2009