Hochschule Wismar
University of Technology, Business and Design
Fakultät für Ingenieurwissenschaften

Automatisierte Hauptspeicher-Forensik auf Basis von Open-Source-Tools

–

Detektion von Verschlüsselungstrojanern mittels Maschinellen Lernens

Masterarbeit
im Studiengang IT-Sicherheit und Forensik

vorgelegt von M.Sc. Christian Haupt

Hamburg
am 20. Mai 2020

Erstgutachterin: Prof. Dr.-Ing. Antje Raab-Düsterhöft
Zweitgutachter: Dipl.-Ing. Hans-Peter Merkel

Aufgabenstellung

In den vergangenen Jahren gewann die Hauptspeicher-Forensik immer mehr an Bedeutung. Gleichzeitig ist die Vielzahl der Fälle sowie die ständig wachsende Datenmenge pro Fall für IT-Forensiker nur noch mit Software-Suiten zu bewältigen, die viele Arbeitsschritte automatisieren. Diese Arbeit gibt einen Überblick über die typischen Ziele, die heutzutage mit der Hauptspeicher-Forensik verfolgt werden. In diesem Zusammenhang ist der aktuelle Stand der Technik im Bereich Hauptspeicher-Forensik mit besonderem Schwerpunkt auf den zur Verfügung stehenden Automatisierungsmöglichkeiten und Werkzeugen (kommerzielle und Open-Source Tools) darzustellen. Basierend darauf sind weitere Möglichkeiten der Automatisierung vorzuschlagen und eine dieser Möglichkeiten ist konzeptuell auszuarbeiten.

Kurzreferat

Vor dem Hintergrund der Schäden durch Verschlüsselungstrojaner, welche Organisationen weltweit in den letzten Jahren zugefügt wurden, wird deutlich, dass neue Methoden zu deren Detektion benötigt werden. Dies gilt insbesondere für virtualisierte Systeme, welche in den letzten Jahren einen enormen Zuwachs an Nutzerzahlen erlebt haben. Diese Systeme haben besondere Sicherheitsanforderungen, bieten aber gleichzeitig auch besondere Möglichkeiten zur Bekämpfung von Trojanern. Die Verknüpfung dieser Möglichkeiten mit modernen Methoden aus dem Bereich Data Science bieten das Potential für eine enorme Erhöhung der Sicherheit heutiger IT-Systeme. Diese Arbeit knüpft an die bestehende Forschung auf diesem Gebiet an. Die Grundlage dazu bildet eine von Cohen und Nissim vorgestellte Methode zur Erkennung von Verschlüsselungstrojanern in Hauptspeicher-Abbildern virtualisierter Server mittels Maschinellen Lernens. Die Methode wird repliziert und auf Basis eines veränderten Hypervisors und Betriebssystem angewandt. Ihre Leistungsfähigkeit, mit einem neuen und erweiterten Satz von jeweils zehn Verschlüsselungstrojanern und gutartigen Programmen in dieser veränderten Umgebung umzugehen, wird erprobt. Es wird versucht die Detektionsleistung mit der Einführung zweier weiterer Algorithmen des Maschinellen Lernens zu verbessern. Die Ergebnisse zeigen, dass die Methode den hier gemachten Veränderungen gewachsen ist. Die Detektionsleistung ist bei bekannten und sogar bei unbekannten Trojanern sehr gut. Die Analysegeschwindigkeit der Hauptspeicher-Abbilder konnte darüber hinaus durch parallelisierte Bearbeitung wesentlich verbessert werden.

Abstract

Against the background of the damage caused by Ransomware to organizations worldwide in recent years, it is clear that new methods for their detection are needed. This is especially true for virtualized systems, which have experienced an enormous increase in user numbers in recent years. These systems have special security requirements, but at the same time offer special possibilities for fighting Ransomware. The combination of these possibilities with modern methods from the field of data science offer the potential for an enormous increase in the security of today's IT systems. This work ties in with existing research in this field. The basis for this work is a method presented by Cohen and Nissim for the detection of Ransomware in main memory images of virtualized servers using machine learning. The method is replicated and applied on the basis of a modified hypervisor and operating system. Its ability to handle a new and extended set of ten Ransomware samples and ten benign programs in this changed environment is evaluated. It is attempted to improve the detection performance by introducing two additional machine learning algorithms. The results show that the method can cope with the changes made here. The detection performance is very good for known and even unknown Ransomware samples. The analysis speed of the main memory images could also be significantly improved by parallel processing.

Danksagung

Ich möchte mich bei all denjenigen bedanken, die mich beim Erstellen dieser Masterarbeit unterstützt und motiviert haben.

Zuerst möchte ich mich bei meiner betreuenden Professorin Frau Raab-Düsterhöft und meinem betreuenden Zweitgutachter Herrn Dipl.-Ing. Merkel für ihre hilfreichen Anregungen und ihre konstruktive Kritik bedanken.

Daneben gilt mein Dank meinen Kollegen im Team Sherlock der SVA GmbH für ihre fachliche Unterstützung. Insbesondere danke ich meinem Teamleiter Günther Schöbel und meinen Kollegen aus der digitalen Forensik Christoph Brandt und Christian Haupt dafür, dass sie mir während der Anfertigung der Masterarbeit den Rücken freigehalten haben.

Der größte Dank gilt meiner Freundin Meriem Harathi und meiner Mutter Gerlinde Haupt für ihren emotionalen und motivierenden Rückhalt über die letzten Monate. Vielen Dank für eure stete Geduld und euer Verständnis!

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

Abkürzung	Bedeutung
API	Application Programming Interface
ARFF	Attribute-Relation File Format
ASUM-DM	Analytics Solutions Unified Method for Data Mining/ Predictive Analytics
AUC	Area Under Curve
Bagging	Bootstrap Aggregation
BSI	Bundesamt für Sicherheit in der Informationstechnik
C&C-Server	Command-&-Control-Server
COCO	Comon Objetcs in Context
CRISP-DM	Cross Industry Standard Process for Data Mining
CSV	Comma Seperated Value
DFIR	Digital Forensic and Incident Response
DLL	Dynamic-Link Libraries
ELF	Executable and Linkable Format
FN	Falsch Negativ
FP	Falsch Positiv
FPR	Falsch-Positiv-Rate
GB	Gigabyte
GHz	Gigahertz
IDS	Intrusion Detection Systeme
IoT	Internet-of-Things
IT	Informationstechnik
JSON	JavaScript Object Notation
K.Erg.	Keine Ergebnisse
KI	Künstliche Intelligenz
MFT	Master File Table
ML	Maschinelles Lernen
PNG	Portable Network Graphics
PVW	Positiver Vorhersagewert
RAM	Random Access Memorys
RAT	Remote-Access-Trojan

Abkürzung	**Bedeutung**
RN	Richtig Negativ
ROC	Receiver Operating Characteristic
RP	Richtig Positiv
RPR	Richtig-Positiv-Rate
SMO	Sequential Minimal Optimization
SVM	Support Vector Machines
UAC	User Account Control
VM	Virtuelle Maschinen

1 Einleitung

In den vergangenen Jahren gewann die Hauptspeicher-Forensik immer mehr an Bedeutung. Als ein Gebiet der digitalen Forensik beschäftigt sie sich mit der Auswertung des Random Access Memorys (RAM) von informationstechnischen (IT) Systemen. Der RAM enthält im Betrieb beispielsweise Namen und Metadaten zu laufenden und beendeten Prozessen, zu System-Kernel-Modulen und ausführbaren Dateien sowie Informationen zu Netzwerkverbindungen, Registry-Schlüsseln und vieles mehr (siehe z. B. [1] Kapitel Windows Memory Forensics). Die Gründe für das wachsende Interesse an der Hauptspeicher-Forensik sind vielfältig. Heutiger Schadcode kann beispielsweise fast ohne Fußabdruck auf dem Massenspeicher nur im RAM laufen (siehe z.B. [2]). Anti-forensische Maßnahmen, die bspw. das Sammeln von Abbildern des Massenspeichers eines Computersystems verhindern oder erschweren, lassen sich häufig nur über den Hauptspeicher nachweisen. Viele forensische Artefakte, wie Passwörter zu verschlüsselten Containern, sind nur im RAM zu finden (siehe z. B. [3]).

Für IT-Forensiker in Sicherheitsbehörden ist Letzteres ein wichtiger Grund die Hauptspeicher-Forensik in ihr Fähigkeitsportfolio aufzunehmen. Aber auch viele, zumeist größere Privatunternehmen haben in den letzten Jahren aufgrund der Bedrohungslage durch Hackerangriffe und Malware-Infektionen (siehe z. B. Lageberichte zur IT-Sicherheit des Bundesamtes für Sicherheit in der Informationstechnik (BSI), aktuell [4]) eigene Fähigkeiten im Bereich der digitalen Forensik und des *Incident Response* (engl. *Digital Forensic and Incident Response* (DFIR)) aufgebaut. Mit diesen soll schnell und effektiv auf IT-Sicherheitsvorfälle reagiert werden. Um sich im Angriffsfall ein Bild über den Zustand eines betroffenen Systems zu verschaffen, ist die Hauptspeicher-Forensik oft ein wichtiges Werkzeug.

Die Hauptspeicher-Forensik stellt im Vergleich mit anderen Gebieten der digitalen Forensik besonders hohe Ansprüche an die hier tätigen Experten. Obwohl die quelloffenen Hauptspeicher-Analyse-Werkzeuge *Rekall* [5] oder *Volatility* [6] es erlauben, unabhängig vom Betriebssystem z. B. mögliche Indikatoren für einen Virenbefall aus dem RAM zu extrahieren, müssen diese doch von Experten bedient werden. Die Auswertung der Indikatoren ist eine zeitaufwändige Tätigkeit, die nur gelingen kann, wenn der Durchführende über umfangreiche Erfahrung mit dem untersuchten System verfügt. Er benötigt vor allem Wissen über das Betriebssystem, die Architektur des Systems sowie darüber, wie sich Systeme mit einer ähnlichen Art des Einsatzes normalerweise verhalten. Erst dadurch können die individuellen Wechselwirkungen zwischen den automatisch gewonnenen Indikatoren manuell zu einer Einschätzung verdichtet werden, ob ein System befallen ist oder nicht. Gleiches gilt auch für Indikatoren, die auf ein Eindringen von Hackern in das System hindeuten.

Durch den immer weiter fortschreitenden Speicherbedarf digitaler Anwendungen befeuert, steigt die durchschnittliche Größe des Hauptspeichers heutiger Computersysteme zunehmend an. Dies verlängert forensische Untersuchungen solcher Systeme erheblich. Gleichzeitig wächst die Komplexität von Betriebssystemen und IT-System-Architekturen ebenso schnell wie die Heterogenität IT-forensischer Fälle und ihrer Datenquellen. Während vor ein paar Jahren meist nur der Massenspeicher eines Desktop-PCs im Mittelpunkt eines Falls stand, kamen über die Jahre mobile Geräte, Fahrzeuge oder das Internet-of-Things (IoT) hinzu. Für den einzelnen Forensiker wird es zunehmend schwieriger, in allen diesen Domänen auskunftsfähig zu sein.

Heutige Fälle sind oft nur noch mittels umfangreicher Software-Suiten zu bewältigen. Diese verarbeiten forensische Datenquellen automatisch und stellen die dabei gefundenen Artefakte danach übersichtlich dar. Im Bereich Hauptspeicher-Forensik sind sie jedoch mangelhaft. Viele große Hersteller von forensischen Software-Suiten betreiben in diesem Bereich den geringstmöglichen Aufwand und stützen sich häufig auf die Programme Rekall und Volatility ab, um ihren Nutzern zumindest grundlegende Funktionalitäten in diesem Bereich zur Verfügung zu stellen (siehe dazu bspw. [7] für Magnet Forensic von Axiom). Andere nutzen zwar eigene Routinen zur Auswertung von Hauptspeicher-Abbildern (siehe dazu [8] für Belkasoft und [9] für X-Ways), bieten dem Nutzer aber nur ein gleichwertiges oder sogar geringeres Funktionsspektrum im Vergleich zu den quelloffenen RAM-Analyse-Werkzeugen. Wieder andere Hersteller wie Nuix publizieren gar keine öffentlichen Informationen zur Hauptspeicher-Forensik mit ihren Tools, was ein Hinweis darauf sein könnte, welcher Stellenwert diesem Bereich zugewiesen wird.

Insgesamt mangelt es vor allem an der Möglichkeit, automatisch verschiedene Merkmale des RAMs zu Erkenntnissen über den Zustand eines Systems zu aggregieren und den IT-Forensiker zu entlasten.

Diese Herausforderung mit einer Kombination aus den modernen Methoden der Big-Data-Analyse und Maschinellem Lernen (ML) anzugehen, ist Inhalt zahlreicher wissenschaftlicher Publikationen der letzten Jahre. Big Data bezeichnet große, meist schwach strukturierte, Datenmengen, die laufenden Veränderungen unterliegen [10]. Das Erkennen von Strukturen, Regelmäßigkeiten oder verborgenen Zusammenhängen in Big Data wird als Data Mining bezeichnet [11]. Maschinelles Lernen ist eine Methode, um Computersysteme in die Lage zu versetzen, solche Muster zu erkennen und so Erkenntnisse automatisch zu generieren, die mit herkömmlichen Methoden der Informatik schwer oder gar nicht erbracht werden können [12].

Bisherige *Intrusion Detection Systeme* (IDS) und Anti-Virusprogramme arbeiten häufig heuristisch und versuchen Schadcode z. B. anhand von Hash-Signaturen und Ähnlichem zu erkennen. Die Effektivität dieser Scanner durch ein Erkennungsverfahren

auf Basis von Maschinellem Lernen besonders dahingehend zu verbessern, neue noch unbekannte Bedrohungen zu entdecken, ist eines der Ziele aktueller Forschungen in diesem Bereich. Langfristig können IT-Forensikern damit Werkzeuge an die Hand gegeben werden, welche die Triage von Computersystemen durch automatisierte Entscheidungsunterstützung beschleunigen und erleichtern.

Diese Arbeit knüpft an die bestehende Forschung an. Die Grundlage dazu bildet die von Cohen und Nissim [13] vorgestellte Methode zur Erkennung von Verschlüsselungstrojanern in Hauptspeicher-Abbildern virtualisierter Server. Verschlüsselungstrojaner stellen einen noch relativ jungen Trend im Bereich Schadcode dar, der jedoch aufgrund seiner Lukrativität massenhaften Einsatz gefunden hat und immer noch findet [4]. Dies gilt auch für die *Cloud*, welche aufgrund weitgehend virtualisierter Systeme besondere Sicherheitsanforderungen hat, aber auch besondere Möglichkeiten zur Bekämpfung von Trojanern bietet, wie in [13] dargestellt wird.

Zum einen soll im Rahmen dieser Replikations- und Erweiterungsarbeit Cohens Methode auf Basis virtualisierter Windows-10-Systeme nachvollzogen werden. Diese Windows-Version stellt die weltweit im privaten wie im beruflichen Kontext am häufigsten genutzte Betriebssystem-Distribution von Desktop-Systemen dar (siehe dazu [14], [15]). Sie ist damit ein lohnendes Ziel sämtlicher Arten von Schadcode. Durch mangelhafte Backup-Konzepte – gerade im privaten Bereich, aber auch bei kleinen und mittelständischen Firmen – sind die Auswirkungen eines Befalls durch Verschlüsselungstrojaner häufig katastrophal (siehe z. B. [16]).

Zum anderen soll die Methode mit einem umfangreicheren Satz neuer Verschlüsselungstrojaner überprüft werden. Schadcode unterliegt ständigem Wandel. Eine Methode zur Erkennung von Virenbefall und Hackerangriffen muss daher robust gegenüber diesem Wandel sein und auch zuverlässig arbeiten, wenn neue Angriffsvektoren genutzt werden.

Die folgenden vier Forschungsfragen stehen im Mittelpunkt dieser Arbeit:

- F1: Sind die von Cohen und Nissim [13] gemachten Beschreibungen ausreichend, um die von ihnen entwickelte Datenpipeline (von der Infizierung einer Virtuellen Maschine, über die Merkmalsextraktion bis hin zum Klassifizieren mittels Maschinellen Lernens) zu replizieren?
- F2: Ist der von Cohen und Nissim für *Windows-Server-2012-R2-Systeme* vorgeschlagene Merkmalssatz auf das Betriebssystem *Windows 10* übertragbar?
- F3: Wie verändert sich die Leitungsfähigkeit der Klassifizierer bei Infizierung des *Windows-10-Systems* mit einem neuen und umfangreicheren Satz an Verschlüsselungstrojanern?
- F4: Können Cohen und Nissims Ergebnisse durch den Einsatz der Klassifizierer *LibLINEAR* und *LibSVM* verbessert werden?

Cohen und Nissim stellen selbst Forschungsfragen auf. Diese unter den oben beschriebenen Anpassungen von Material und Methode erneut zu beantworten, ist die Voraussetzung für die Beantwortung der in dieser Arbeit aufgestellten Forschungsfragen F1 – F4. Es werden jedoch nur die den Forschungsfragen Eins bis Vier aus [13] behandelt (siehe Abschnitt 3: CuN-F1 – CuN-F4). Das Thema Fernzugriffstrojaner (engl. *Remote-Access-Trojan* (RAT)) wird nicht betrachtet, da die Anforderungen an die Infrastruktur einer Labor-Umgebung den Rahmen dieser Arbeit bei weitem sprengen.

2 Allgemeine Grundlagen

Dieses Kapitel gibt einen Überblick über die wichtigsten thematischen Grundlagen der Arbeit. Es soll dem Leser dabei helfen, das in Kapitel 3 Material und Methode dargestellte Vorgehen zu verstehen. Dazu wird zuerst auf *Verschlüsselungstrojaner*, ihre Wirkungsweise und Angriffsvektoren sowie Mechanismen zu ihrer Erkennung eingegangen. Im Folgenden wird das Thema *Hauptspeicher-Forensik* erörtert und besonders werden die Funktionen des Frameworks Volatility, welches in dieser Arbeit genutzt wird, vorgestellt. Abschließend wird das Thema *Maschinelles Lernen* erörtert. Hier steht ein Überblick über die Klassifizierer-Familien, welche im späteren Verlauf genutzt werden, im Vordergrund. Ebenso werden die zur Leistungsmessung der Klassifizierer genutzten Verfahren erklärt.

2.1 Verschlüsselungstrojaner

Verschlüsselungstrojaner (engl. Ransomware) sind eine Art von Schadcode, der den Nutzer eines befallenen IT-Systems daran hindert, auf seine Daten oder die Funktionen des Systems zuzugreifen. Häufig versucht der Verschlüsselungstrojaner nicht das System zu zerstören, sondern in den meisten Fällen bestimmte Dateitypen zu verschlüsseln. Oft sind dies Bild- und Dokument-Dateitypen (siehe Abb. 2-1: Verschlüsselung). Der Designer des Schadcodes versucht damit gezielt, wichtige Inhalte des Nutzers zu treffen, seien es Urlaubsbilder im privaten oder Firmendokumente im beruflichen Kontext [17]. In anderen Fällen wird nur der Master File Table (MFT), also das Inhaltsverzeichnis der auf dem Massenspeicher liegenden Dateien, verschlüsselt. Dies stellt einen weiteren Weg dar, den Nutzer am Zugriff auf seine Dateien und sein System als Ganzes zu hindern [18]. Zusätzlich versuchen viele Verschlüsselungstrojaner Backups des Nutzers zu zerstören (siehe Abb. 2-1: Infektion) oder sich auf andere Rechner im lokalen Netzwerk zu verbreiten [17].

Nach erfolgreicher Verschlüsselung wird dem Nutzer auf dem Bildschirm ein Erpresserschreiben präsentiert (siehe Abb. 2-1: Erpressung). Er wird darin aufgefordert, Geld an den Urheber des Schreibens zu übermitteln. Ihm wird versprochen, nach Eingang des Geldes wieder Zugriff auf seine Dateien zu erhalten. Dieses Angebot ist häufig nur für bestimmte Zeiträume gültig, um das Opfer unter Handlungsdruck zu setzen. Teilweise werden nach Verstreichen bestimmter Zeitintervalle und, wenn der Nutzer bestimmte vom Erpresser unerwünschte Aktionen durchführt, zufällige Dateien vom System gelöscht. Die Zahlung soll meist in einer Kryptowährung wie z. B. Bitcoin oder auf einem anderen elektronischen Weg erfolgen, der die Anonymität des Empfängers sicherstellt [17]. Wichtig zu betonen ist, dass die Zahlung des Lösegeldes den Opfern in vielen Fällen nicht den Zugriff auf ihre Dateien ermöglicht. So sind Tro-

janer-Familien im Umlauf, welche die Dateien des Nutzers nicht verschlüsseln, sondern z. B. mit Zufallswerten überschreiben. Bei anderen wurde vom Designer die Entschlüsselung nicht richtig (siehe z.B. GandCrab in [19]) oder gar nicht implementiert. In beiden Fällen ist eine Entschlüsselung der Dateien auch nach Erhalt des vermeintlichen Passworts vom Erpresser nicht möglich [4].

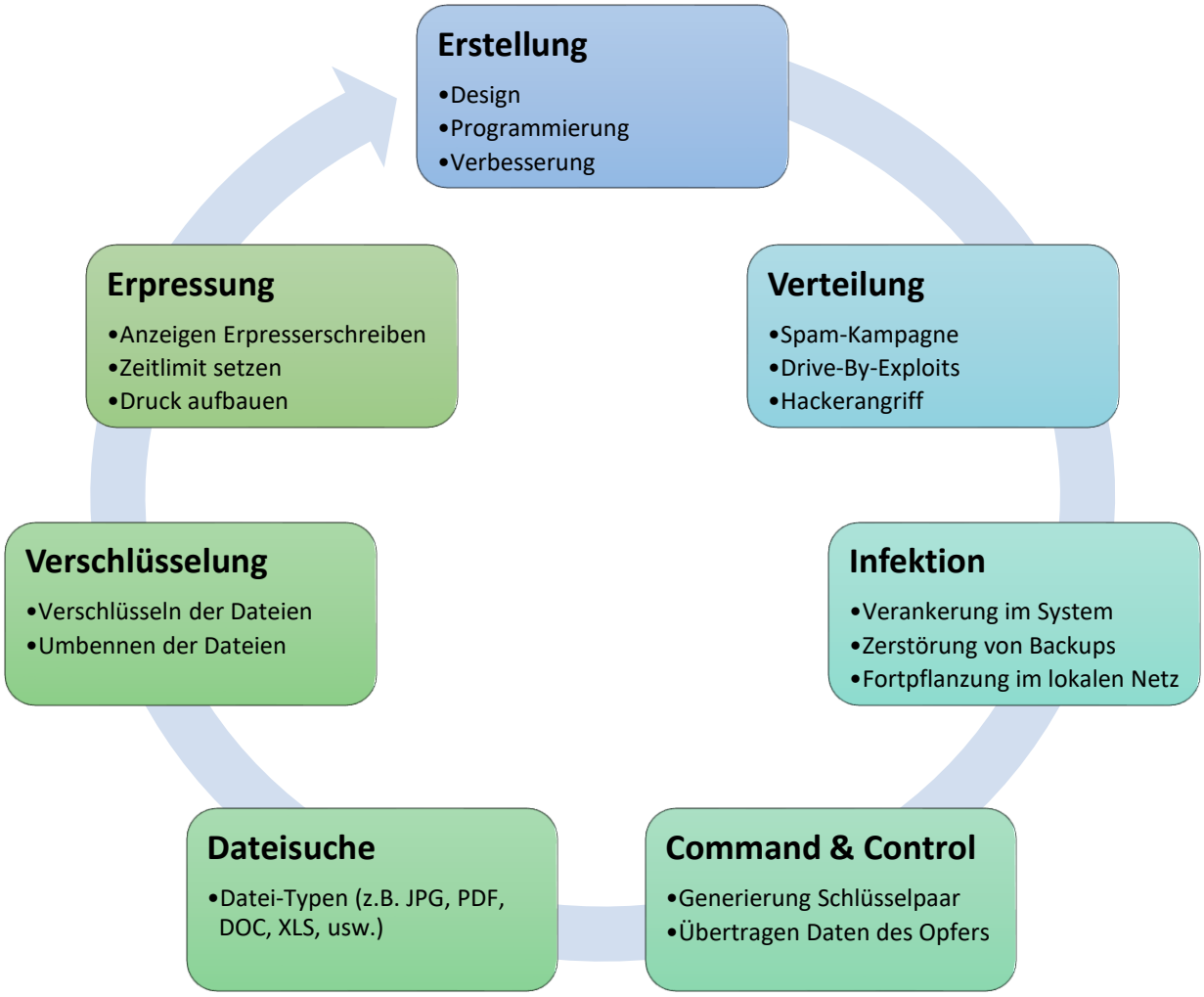


Abb. 2-1: Lebenszyklus Verschlüsselungstrojaner (nach [20])

Der Infektionsvektor für das Eindringen des Trojaners ins System besteht häufig aus Spam-E-Mails (siehe Abb. 2-1: Verteilung), aber auch gezielte Phishing-Kampagnen gegen bestimmte Organisationen als Ganzes oder gegen einen ausgewählten Teil ihrer Mitarbeiter (sog. Spear-Phishing) sind üblich. In solchen E-Mails verbergen sich präparierte Dokumente, die über Schwachstellen in den Anzeigeprogrammen oder über Makro-Schnittstellen den Trojaner im System verankern [17]. Eine Infektion ist jedoch auch über Drive-By-Exploits, Innentäter oder nach einem vorangegangenen Eindringen von Hackern in das System möglich [4].

Frühe Verschlüsselungstrojaner setzen noch symmetrische Verschlüsselungsverfahren ein. Dies ermöglicht die Entschlüsselung aller betroffenen Systeme bei Bekanntwerden des Passworts. Die Entwicklung verläuft anschließend über symmetrische Verschlüsselung mit systemspezifischen Passwörtern, hin zu asymmetrischen Verschlüsselungsverfahren. Teilweise erfolgt die Schlüsselgenerierung in Abstimmung mit einem *Command-&-Control-Server* (*C&C-Server*) im Internet, um zu verhindern, dass privater und öffentlicher Schlüssel auf dem System des Opfers verbleiben (siehe Abb. 2-1: Command & Control). So werden die Chancen für eine Wiederherstellung der Dateien geschmälert [17].

Wie aus obigen Erläuterungen herauszunehmen ist, sind heutige Verschlüsselungstrojaner nicht mehr nur der Schadcode-Kategorie der Trojaner zuzuordnen. Als Trojaner (Metapher für: Durch listige Täuschung Angreifer/feindliche Eroberer in die Stadt einschleusen (griechische Mythologie)) werden schädliche Programme verstanden, die sich als nützliche Applikation tarnen. Sobald sie ausgeführt werden, beginnen sie unerwünschte Funktionen auszuführen. Da Verschlüsselungstrojaner heutzutage auch auf anderen Wegen auf die Systeme der Opfer gelangen, ist diese Bezeichnung nicht mehr zutreffend. Das englische Wort *ransomware* (zusammengesetzt aus *ransom*, engl. für Lösegeld, und *ware*, als Abkürzung für Software) ist hier zutreffender [17]. Der Einfachheit halber wird der im Deutschen am weitesten verbreitete Begriff des *Verschlüsselungstrojaners* in dieser Arbeit verwendet.

2.2 Schadensausmaß

Der in den letzten Jahren global entstandene Schaden durch Verschlüsselungstrojaner ist enorm. *Cybersecurity Ventures* geht für das Jahr 2021 von einem erfolgreichen Angriff alle 11 Sekunden auf Unternehmen weltweit aus. Die kumulierten Kosten werden auf 20 Mrd. US-Dollar geschätzt [20]. Bereits einzelne erfolgreiche Angriffe, wie der auf den norwegischen Aluminiumkonzern *Norsk Hydro* im Jahr 2019 durch den Trojaner *Lockergoga*, schlagen mit geschätzten 35 bis 43 Mio. US-Dollar zu Buche. Diese Schadenssumme wurde erreicht, obwohl das Unternehmen laut eigenen Angaben kein Lösegeld zahlte und einen Großteil der befallenen Systeme aus Backups wiederherstellen konnte. Die Produktion konnte während dieser Zeit auf manuell umgestellt werden, um eine Unterbrechung zu vermeiden [21]. Für Unternehmen oder Privatpersonen ohne ausreichende Backup-Strategie kann von noch schwerwiegenderen Folgen bei einem umfangreichen Befall ausgegangen werden.

Auch das BSI prognostiziert, dass der Schaden durch diese Art von Schadcode, besonders aufgrund der immer gezielteren Angriffe auf große Unternehmen mit speziell zugeschnittenen Trojanern, noch steigen wird. Die immer professionellere Entwicklung mit modular aufgebautem Quellcode, die relativ geringen Ansprüche an das

Knowhow der Entwickler im Vergleich zu anderem Schadcode [22] sowie die hohe Agilität bei der Anpassung der Angriffsvektoren sind ebenfalls Gründe hierfür [4].

2.3 Klassische Erkennungsmechanismen

Antivirus-Software erkennt Schadcode mittels einer Vielzahl unterschiedlicher Methoden. Am weitesten verbreitet ist die **signaturbasierte Erkennung**. Programme werden dabei von den Herstellern der Antivirus-Software manuell oder automatisch analysiert. Sobald sicher festgestellt ist, dass es sich um Schadcode handelt, wird ein eindeutiger Identifizierer (z. B. in Form eines Hash-Wertes oder durch die Extraktion von einzigartigen Codefragmenten [23]) über das Programm gebildet und in die Signaturdatenbank des Antivirus-Programms aufgenommen. Neue Programme können nun gegen diese Datenbank geprüft werden. Die signaturbasierte Erkennung hat den Vorteil einer geringen *Falsch-Positiv-Rate*. Diese gibt das Verhältnis der fälschlicherweise als Schadcode klassifizierten Programme zur Gesamtanzahl der gutartigen Programme in einer Stichprobe an. Gleichzeitig ist der Aufwand der Prüfung eines Programms auf diese Art gering [24].

Schadcode-Designer haben jedoch z. B. mit *polymorphen* oder *metamorphen* Programmen auf diese Erkennungsmechanismen reagiert. Durch diese Methoden wird der Quellcode eines Programms verändert, während seine Funktionsweise dieselbe bleibt [25]. Die signaturbasierte Erkennung bietet daher nur Schutz gegen bekannte und unveränderte Bedrohungen. Wie in Abschnitt 2.1 beschrieben, ist sie deswegen bei den schnelleren Entwicklungszyklen und der immer größeren Professionalität der Schadcode-Designer gerade im Bereich der Verschlüsselungstrojaner nicht mehr ausreichend.

Antivirus-Hersteller versuchen mit Methoden der heuristischen und verhaltensbasierten Erkennung auch solchen Schadcode zu entdecken, der seinen Quellcode verändert bzw. noch nicht in den Signaturdatenbanken aufgenommen wurde. Die **heuristische Erkennung** sucht nach allgemeinen Merkmalen in Schadcode-Familien (z. B. in Form von nicht aufeinander folgenden Quellcode-Fragmenten mittels Platzhaltern variabler Länge, ein Beispiel hierfür sind *yara*-Regeln, siehe dazu [1]),) und kann so auch Abwandlungen eines Programmes erkennen [25]. Die **verhaltensbasierte Erkennung** überwacht die Ausführung von Programmen und greift ein, sobald bestimmte Funktionsmuster erkannt wurden, die auf potenziell unerwünschte bzw. gefährliche Aktivitäten hindeuten. Eine Erweiterung dieser Methode, welche jedoch aufgrund ihres Ressourcenverbrauchs und der für die Analyse eines Programms langen Laufzeit noch keine weite Verbreitung erfahren hat, ist die **Sandbox-Erkennung**. Potenzieller Schadcode wird dabei in einer speziell gesicherten und einfach von außen zu überwachenden Umgebung, der sogenannten *Sandbox*, ausgeführt. Werden

dabei vorher definierte Funktionsmuster beobachtet, wird das Programm als schädlich eingestuft, ohne dass das System des Nutzers durch den Schadcode verändert werden kann [25].

Schadcode-Designer haben vermehrt Wege gefunden, auch diese aufwändigeren Erkennungsmechanismen zu umgehen. Im Folgenden werden beispielhaft drei Wege vorgestellt:

- **Antiheuristische Methoden**: Z. B. durch die Benutzung von besonderen *Packern*, welche den Schadcode auf unterschiedliche Art und Weise komprimieren, um auffällige Merkmale zu verstecken [25].
- **Verhaltensanpassung**: Z. B. durch das Zuschneiden des Schadcodes auf weit verbreitete Antivirus-Programme, durch das Benutzen nur bestimmter Systemcalls (Aufrufe der Schnittstellen des Betriebssystems) oder eines besonderen Programmablaufs, um die Entdeckungswahrscheinlichkeit zu minimieren [25].
- **Antiemulation**: Z. B. durch das Erkennen von Sandboxen und das anschließende Stilllegen des Schadcodes, um in diesen Umgebungen kein auffälliges Verhalten zu zeigen [26].

2.4 Erkennung mittels Maschinellen Lernens

Die hier dargestellten Antworten der Schadcode-Designer auf die Fortschritte der Antivirus-Hersteller erschweren die Erkennung von bereits bekannten schädlichen Programmen. Das Hauptproblem für die Antivirus-Hersteller liegt aber vor allem darin, neuartigen Schadcode in Programmen aufzudecken (sogenannte *Zero-Days*, also Schadcode, der vorher völlig unbekannt war). In den letzten Jahren wurden vermehrt Bemühungen unternommen, dieser Herausforderung durch den Einsatz moderner Methoden wie dem *Data Mining* und dem *Maschinellen Lernen* zu begegnen. Diese Methoden eignen sich grundsätzlich dazu, auch unbekannten Schadcode korrekt als solchen zu klassifizieren. Das zugrundeliegende Vorgehen besteht aus einem zweistufigen Prozess, bei dem zuerst *Merkmale* (engl. *Features*) aus dem zu untersuchenden Programm extrahiert werden. Diese beschreiben das Programm auf unterschiedliche Art und Weise. Es kann sich um ausgeführte Systemcalls, bestimmte Strukturen des Quellcodes oder auch Beschreibungen des Verhaltens handeln [24].

Anschließend werden Klassifikationsmethoden aus dem Bereich des Maschinellen Lernens eingesetzt, um zuerst ein *Modell* der Daten zu erstellen und dieses danach zur *Klassifikation* einzusetzen. Für die Modellerstellung müssen die Merkmale aus Stichproben von *gutartigen* (engl. *benign*) und bösartigen bzw. *infizierten* Programmen (engl. *infected*) extrahiert werden und dem eingesetzten Klassifikationsalgorithmus vorgelegt werden. Der Klassifikationsalgorithmus erstellt auf Basis dieser Daten

ein Modell. Dieses Modell wird auch *Klassifizierer* genannt. Sobald ein Klassifizierer auf diese Art trainiert wurde, können ihm Merkmale vorgelegt werden, welche aus neuen Programmen extrahiert wurden. Das Ergebnis der Klassifikation ist häufig ein boolescher Wert bzw. ein Prozentwert, mit dem ausgedrückt wird, zu welcher Klasse (gutartig bzw. infiziert) ein Programm gehört [24]. Notwendige Randbedingungen und eine detaillierte Beschreibung des Maschinellen Lernens sind im Abschnitt 2.6 dieser Arbeit zu finden.

Obwohl das Training der Klassifizierer und besonders die dafür notwendige Datensammlung sehr aufwendig sind, haben die beachtlichen Fortschritte bei der Schadcode-Erkennung mittels *Data Minings* bereits dazu geführt, dass Maschinelles Lernen in vielen kommerziellen Antivirus-Produkten Einzug gehalten hat (siehe bspw. hier: [27] und [28]).

2.5 Hauptspeicher-Forensik

Die digitale Forensik ist die streng methodisch vorgenommene Datenanalyse auf Datenträgern und in Computernetzen zur Aufklärung von Vorfällen unter Einbeziehung der Möglichkeiten der strategischen Vorbereitung insbesondere aus der Sicht des Anlagenbetreibers eines IT-Systems [29]. Die Hauptspeicher-Forensik beschäftigt sich als ein Gebiet der digitalen Forensik mit der Auswertung des *Random Access Memorys* von informationstechnischen Systemen. Der RAM enthält im Betrieb beispielsweise Namen und Metadaten zu laufenden und beendeten Prozessen, zu System-Kernel-Modulen und ausführbaren Dateien sowie Informationen zu Netzwerkverbindungen, Registry-Schlüsseln und vielem mehr (siehe z. B. [1] Kapitel Windows Memory Forensics).

Alle Funktionen eines Betriebssystems oder eines darauf laufenden Programms spiegeln sich im RAM wider. Obwohl die Speicherung von Daten im RAM grundsätzlich volatiler Natur ist und z. B. von einer ständigen Stromversorgung abhängt, können Veränderungen darin auch lange nach dem Ausführen einer Funktion erhalten bleiben. Somit gibt der Hauptspeicher eines IT-Systems Einblicke in den aktuellen Zustand aller laufenden Programme und ermöglicht im gewissen Umfang auch einen Blick in die Vergangenheit [1].

Konkret speichert der RAM Code und auch Daten, die der Prozessor für seine Berechnungen benötigt oder welche Zwischenergebnisse einer solchen Berechnung sind. Dazu werden allen im sogenannten *user mode* laufenden Programmen vom Betriebssystem private Hauptspeicherbereiche (sog. virtuelle Adressräume) zugewiesen. Der *user mode* umfasst alle Programme, welche nicht zum Kern des Betriebssystems gehören [30]. Programme in modernen Betriebssystemen wissen deshalb nicht, an welchen physikalischen Hauptspeicherstellen sich ihr Code oder ihre Daten befinden.

Zum einen hat dies den Vorteil, dass Programme nicht versehentlich Hauptspeicherbereiche anderer Programme und des Betriebssystems lesen oder überschreiben können. Zum anderen können vom Betriebssystem (konkret dem *Memory Manager*) die Adressräume geteilt und über die gesamte Größe des zur Verfügung stehenden physikalischen Hauptspeichers verteilt werden [31]. Momentan nicht benötigte Teile des Adressraums werden dabei auch auf den Massenspeicher ausgelagert, wenn im RAM nicht mehr genügend Platz zur Verfügung steht (unter Windows 10 bspw. in die Datei *pagefile.sys*). Bei Bedarf werden sie wieder zurück in den RAM geladen. Dies geht mit einer im Vergleich zu den Hauptspeicherzugriffszeiten wesentlich größeren Verzögerung einher, ermöglicht es jedoch, Code und Daten von mehr Programmen gleichzeitig auszuführen als ohne Auslagerung. Für die Programme laufen diese Aktivitäten völlig transparent ab. Die Hauptspeicherbereiche der Programme im *user mode*, in denen aus digital-forensischer Sicht interessante Artefakte zu finden sind, heißen *stack* und *heap*. Sie enthalten im Fall eines Schadcode-Befalls des IT-Systems bspw. Informationen darüber,

- welcher Teil eines Schadcodes ausgeführt wurde
- mit welchen Teilen des Systems der Schadcode interagiert hat
- welche Daten (Tastaturanschläge, Dateiinhalte) abgeflossen sind
- welche Schlüssel bei einer Verschlüsselungsoperation genutzt wurden
- u.v.m [1]

Darüber hinaus sind im Hauptspeicher auch vom Betriebssystem benötigte Informationen abgelegt. Die Funktionen des Betriebssystems laufen im sogenannten *kernel mode*. Sie werden in einem gemeinsamen vom *user mode* getrennten Adressraum verwaltet. Hier sind weitere interessante forensische Artefakte zu finden. Dies sind z. B. geladenen *Module* (engl. *module*), welche u.a. geladenen Gerätetreiber enthalten.

2.5.1 Der forensische Prozess

Das Vorgehen bei der forensischen Untersuchung von Hauptspeichern ist analog dem Leitfaden „IT-Forensik“ des BSI (siehe [29] und Abb. 2-2). Im Folgenden ist dieses Vorgehen zusammengefasst und es wird geschildert, welche Besonderheiten bei der Hauptspeicher-Forensik zu beachten sind.

- **Strategische Vorbereitung**: In diesem Untersuchungsabschnitt werden Maßnahmen getroffen, die vor dem Eintreten eines Vorfalls liegen und dessen spätere Aufklärung betreffen. Dies ist beispielsweise das Einschalten von Logging-Diensten. In dieser Arbeit wurden die eingesetzten *Virtuellen Maschinen* (VM) vorbereitet, um eine erfolgreiche Untersuchung des RAMs sicherzustellen (siehe Abschnitt 3.1 und Anhang A).

- **Operationale Vorbereitung**: In diesem Untersuchungsabschnitt werden Maßnahmen getroffen, die nach dem Eintreten eines Vorfalls, aber vor dem Beginn der Datensammlung liegen. Dies ist z. B. die Enumeration der zu untersuchenden Datenquellen. Im Rahmen dieser Arbeit wurden umfangreiche Skripte mit der Programmiersprache Python entwickelt, die ein Auswerten des RAMs ermöglichen (siehe Anhang 3.2).
- **Datensammlung**: In diesem Untersuchungsabschnitt werden Abbilder aller relevanten Datenträger erstellt, um diese analysieren zu können, ohne Veränderungen an den originalen Daten befürchten zu müssen. Dies geschieht beim Massenspeicher üblicherweise unter Einsatz eines *Write-Blockers* (z. B. Tableau Forensic USB 3.0 Bridge [32]) und eines *Imagers* (z. B. FTK Imager [33]). Ersterer blockiert alle Schreibzugriffe auf den Datenträger und letzterer macht eine Bit-genaue Kopie der Inhalte auf einem Sicherungsmedium. Dieser Kopiervorgang wird mit einer Prüfziffer in Form eines Hashwertes (z. B. mit MD5) abgesichert. Anhand dieses Wertes können auch spätere Veränderungen an der Sicherungskopie nachgewiesen werden. Die Erstellung eines Hauptspeicher-Abbildes gestaltet sich schwieriger, da sein Inhalt nur während der Laufzeit des Systems vorgehalten wird und durch jede Aktion des Benutzers verändert wird. Die Datensammlung ist jedoch auch in einer solchen Situation möglich. Einen umfangreichen Überblick über Methoden und Möglichkeiten gibt z. B. [1] Kapitel Memory Acquisition. Die Datensammlung in dieser Arbeit geschieht automatisch auf Basis selbst entwickelter Skripte (siehe ebenfalls Abschnitt 3.2).
- **Untersuchung**: In diesem Untersuchungsabschnitt werden aus den vorher erstellten Abbildern forensisch relevante Daten extrahiert. Das können z. B. Bilddateien sein, aber auch E-Mails oder Chatverläufe. Die in dieser Arbeit vorgestellte Methode beruht auf der Extraktion von Merkmalen mit dem Hauptspeicher-Analyse-Framework Volatility (siehe Abschnitte 2.5.2 und 3.3.1). Dazu zählen u.a. die Anzahl geladene Module, laufende Timer und ausgeführte Prozesse.
- **Datenanalyse:** In diesem Untersuchungsabschnitt werden die extrahierten forensischen Daten einer Detailanalyse unterzogen. Ziel ist es, meist eine Verbindung zwischen Ereignissen (Erstellen einer Timeline) oder Daten und Personen (Attribution) herzustellen. In der Hauptspeicher-Forensik geht es darüberhinaus häufig um das Erkennen von Schadcode-Befall oder das Auslesen von Passwörtern. Im Rahmen dieser Arbeit werden die mit Volatility extrahierten Merkmale weiter verdichtet, um ihre Anzahl zu reduzieren und spezifische Fragestellungen zu beantworten. Diese erleichtern es den

eingesetzten Klassifizierern, auf Basis von Maschinellem Lernen, infizierte von gutartigen RAM-Abbildern zu unterscheiden (siehe Abschnitt 3.4)

- **Dokumentation**: Parallel zu allen Untersuchungsabschnitten wird eine prozessbegleitende Dokumentation geführt, welche am Ende in einem Abschlussbericht zusammengefasst wird. Die Nachvollziehbarkeit des forensischen Vorgehens hat dabei oberste Priorität. Es muss möglich sein, dieselben Untersuchungsergebnisse zu erzielen, wenn der geschilderte Ablauf von einem anderen Forensiker wiederholt wird. Für die in dieser Arbeit vorgestellte Methode wird dieses Ziel mit dem Abschnitte 3.5 verfolgt. Zusätzlich liegt dieser Arbeit eine umfangreiche Datensammlung auf CD bei, welche als Grundlage für weitere Replikationen dienen soll.

Abb. 2-2: Abschnitte des forensischen Prozesses (nach [30])

2.5.2 Das Framework Volatility

Volatility ist eines von mehreren Frameworks zur Analyse von Hauptspeichern. Es erlaubt den schnellen Zugriff auf bestimmte RAM-Inhalte, deren Auffinden ohne eine automatisierte Unterstützung langwierig bis unmöglich ist. Sowohl bei der Untersuchung von laufenden Systemen als auch von RAM-Abbildern wird die Arbeit von Forensikern durch das Framework erheblich vereinfacht.

Von Freiwilligen in *Python* implementiert und unter *GNU General Public License 2* stehend, kann Volatility 2.6.1 von der eigenen *Github*-Seite heruntergeladen und frei benutzt werden [34]. Der Quellcode steht ebenfalls offen zur Verfügung und kann beliebig angepasst und erweitert werden [1].

Obwohl Volatility das laut Aussage auf der eigenen Github-Seite am weitesten verbreitete und am meisten genutzte Hauptspeicher-Analyse-Framework ist, gibt es noch andere, wie z. B. *Rekall* von Google [5]. Ein Vergleich zwischen den Frameworks ergibt jedoch schnell, dass Volatility eine wesentlich größere und aktiviere Community hat als Rekall (siehe hierzu die Github *Insights* und *Issues*). Aus diesen Gründen wurde die Entscheidung für den Einsatz von Volatility in dieser Arbeit getroffen. Ob die Hauptspeicher-Analysen dieser Arbeit auch mit Rekall durchgeführt werden können, muss das Ziel zukünftiger Untersuchungen sein.

Im Oktober 2019 wurde die Version 3.0 des Frameworks veröffentlicht [35]. Aufgrund des Beta-Stadiums der Software sowie der noch unvollständigen Dokumentation hat sich der Autor dieser Arbeit dazu entschlossen, auf die stabile Version 2.6.1 des Frameworks zurückzugreifen. Auf diese Version beziehen sich daher alle im Folgenden getroffenen Aussagen.

Volatility besteht aus einer Reihe von Subsystemen, die zusammen die umfangreiche Funktionalität des Frameworks zur Verfügung stellen. Auf unterster Ebene arbeitet Volatility mit sogenannten *VTypes*. Ähnlich wie Dateien auf dem Massenspeicher strukturieren sogenannte Datenstrukturen im Hauptspeicher dessen Inhalte. Datenstrukturen setzen sich aus grundlegenden Datentypen (in der Programmiersprache *C* bspw. *char*, *int*, *long*, *float* usw.) zusammen [1]. Aufgrund seiner guten Eignung für die Betriebssystemprogrammierung, z. B. wegen seiner Möglichkeiten zur direkten Verwaltung von Speicherzuweisungen, ist C häufig anzutreffen, wenn der speicherresidente Zustand moderner Betriebssysteme analysiert wird. Die Volatility-Community hat mit den *VTypes* eine Syntax geschaffen, um *C*-Datenstrukturen in der Frameworkeigenen Sprache *Python* nachzubilden. Diese Abstraktionsebene ermöglicht es gezielt, nach Datenstrukturen im RAM zu suchen und spezielle Funktionen anzubieten. Zum Beispiel können so Windows-eigene Zeitstempel, die in einem eigenen Format (*_LARGE_INTEGER*, zusammengesetzt aus zwei 32-bit Integer-Werten) gespeichert werden, automatisch und global im Framework in ihre entsprechende für den Menschen lesbare Repräsentation übersetzt werden [1].

Die meist automatisch aus z. B. Microsofts Debugging-Symbolen (PDB-Dateien) generierten *VTypes* können durch sogenannte *Overlays* an die Besonderheiten einer Betriebssystemversion angepasst werden. Dies ermöglicht es u.a., Fehler des automatischen Erzeugungsprozesses auszugleichen und wiederum spezielle Funktionen anzubieten.

Die Position der *VTypes* im RAM, ihre jeweilige Bedeutung an der Stelle sowie bspw. die Übersetzung von virtuellen zu physikalischen Adressen führt Volatility mit sogenannten *Adressräumen* (engl. *address space*) durch. Diese ermöglichen, den Hauptspeicher eines Systems in definierte Teile zu zerlegen und einzeln zu bearbeiten (vgl. Kapitel 2.5, eingangs: *user mode* und *kernel mode*). Wenn Volatility versucht eine Adresse zu lesen, wird diese zuerst von einer virtuellen in eine physikalische Adresse, abhängig von der Architektur des Systems, übersetzt (vgl. Abb. 2-3). Das Ergebnis ist ein Offset im physikalischen Speicher, von dem aus die gesuchten Daten gelesen werden können, wenn es sich um einen rohen Speicherauszug handelt. Bei *Crashdump*-Dateien, die bspw. nach einem Absturz des Systems erstellt werden, müssen noch die enthaltenen *Header* beachtet werden. Sie werden in einem Crashdump-Adressraum definiert. Dieser übersetzt die physikalische Adresse in eine Adresse innerhalb der Crashdump-Datei, welche den Speicherinhalt zum Zeitpunkt des Absturzes enthält. Mit diesen verschachtelten Adressräumen kann Volatility mit einer großen Bandbreite von Architekturen sowie Speicherabbild-Dateien umgehen [1].

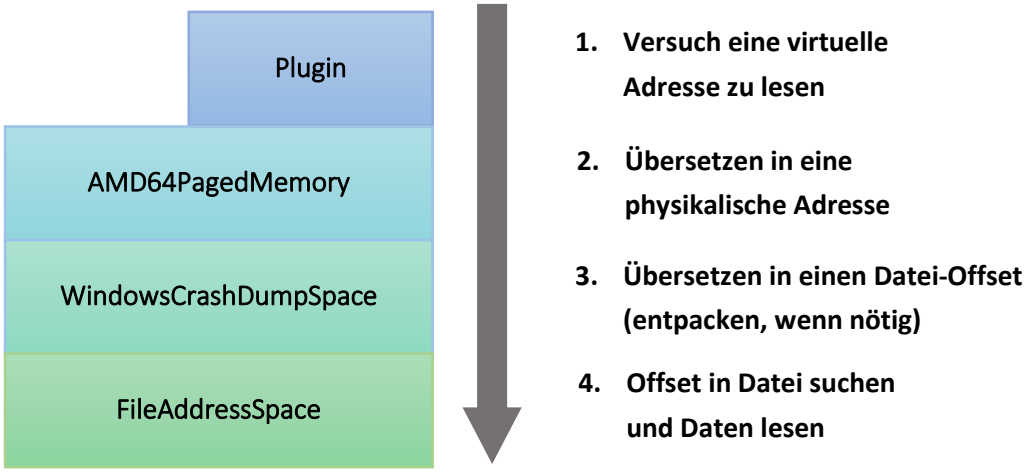


Abb. 2-3: Die Volatility-Adressräume im Detail (nach [1])

Ein im RAM gefundener *VType* wird als sogenanntes *Volatility-Objekt* instanziiert. Zu jedem Objekt können zusätzlich Klassen angelegt werden, welche die bereits vorhandenen Funktionalitäten der *VTypes* zusätzlich erweitern. So sind beispielweise automatische Checks auf einem Prozess-Objekt (in Windows z. B. *_EPROCESS*) möglich, die das Ziel haben, herauszufinden, ob der Prozess bösartig ist [1].

Volatility fasst *VTypes*, *Overlays* und *Objekte* zu sogenannten *Profilen* zusammen, die meist nach dem Betriebssystem, dem aktuellen Service Pack und der Hardware Architektur (*x86, x64, ARM*) benannt sind. Für ein System bzw. ein RAM-Abbild können alle notwendigen Einstellungen mittels *Profilen* gemeinsam geladen werden.

Die Volatility-Funktionen, welche der Endanwender in der digitalen Forensik nutzt, werden durch *Plugins* bereitgestellt. Von der Community meistens auf Basis neuer

Erkenntnisse in der IT-Forensik geschrieben, finden und analysieren sie bestimmte Komponenten des Betriebssystems, Benutzeranwendungen oder sogar bösartige Codebeispiele [1].

2.6 Maschinelles Lernen

Algorithmen des Maschinellen Lernens haben sich bei der Lösung komplexer Probleme als sehr vielversprechend erwiesen. Viele Anwendungen, die bereits heute eine breite Nutzung erfahren, wie z. B. Suchen im World Wide Web (WWW) oder das Übersetzen oder Transkribieren von Sprache, basieren auf diesen Methoden. Maschinelles Lernen als ein Bereich der Informatik beschäftigt sich mit Algorithmen, welche automatisierte Lösungen für komplexe Probleme bieten, die mit den klassischen Programmiermethoden nur schwer oder gar nicht lösbar sind [36]. Solche Probleme entstehen häufig dann, wenn aus einem bestehenden Datensatz heraus Schlüsse über neue noch unbekannte Daten gezogen werden sollen. Ein Beispiel ist das Erkennen von handgeschriebenem Text in einem Bild. Basierend auf einem begrenzten Datensatz von vorhanden Bildern, muss der Programmierer Regelsysteme entwerfen, die ebenfalls auf noch unbekannten Bildern funktionieren. In den 1980er Jahren wurde diese Herangehensweise mit sogenannten *Expertensystemen* (engl. *expert systems*) erprobt. Diese scheiterten vor allem an der Komplexität der entstehenden Systeme und der Dauer deren Implementierung, da für jedes Problem ein komplett neues System entworfen werden musste [36].

Mittels der Methoden des Maschinellen Lernens können solche Probleme generisch gelöst werden. Es ist nicht notwendig, jeweils spezifische Systeme zu implementieren. Vielmehr wird ein Algorithmus mit den vorhandenen Daten *angelernt* und so auf ein Problem spezialisiert. Meistens gilt dabei, je größer der Datensatz für das Anlernen, desto genauer wird der Algorithmus. Diesen Vorgang bezeichnet man als das *Erstellen* (manchmal auch *Trainieren* oder *Anlernen*) *eines Modells* (engl. *model*) [36]. Im nächsten Schritt wird das Modell auf noch unbekannte Daten angewandt, um diese z. B. zu klassifizieren, zu clustern, Assoziationen zwischen Daten herzustellen oder Vorhersagen zu treffen, wie diese sich im Zeitverlauf verändern [37].

Das Themenfeld *Künstliche Intelligenz* (KI), welches häufig mit ML gleichgesetzt wird, beschäftigt sich mit der Frage, wie man Maschinen intelligente Verhaltensweisen beibringt (z. B. mittels Expertensystemen). ML ist eine von vielen Methoden, um dies zu tun. Es unterscheidet sich aber dahingehend von KI, dass es zum Ziel hat, Maschinen zu erstellen, welche lernen können Aufgaben auszuführen [36].

Im Folgenden sollen die für diese Arbeit wichtigsten Begriffe aus dem Bereich des Maschinellen Lernens vorgestellt werden. Hierbei werden die englischen Begriffe den

in dieser Arbeit verwendeten deutschen gegenübergestellt, da das Gros der Fachliteratur auf Englisch gehalten ist und dem Leser so ein leichterer Vergleich zu dieser ermöglicht werden soll.

2.6.1 Überwachtes und unüberwachtes Maschinelles Lernen

Das in dieser Arbeit mit ML adressierte Problem gehört in den Bereich der Klassifikation. Es geht also darum, automatisierte Vorhersagen zu treffen, ob ein Datensatz zu einer bestimmten Kategorie gehört. Um Modelle zur Lösung dieser Art von Problemen zu erstellen, wird häufig sogenanntes *überwachtes Lernen* eingesetzt. Hierbei wird dem Algorithmus zum Anlernen ein Datensatz (der *Trainingsdatensatz*) präsentiert, welcher mit *Bezeichnern* versehene (engl. *labelled*) *Instanzen* enthält. Eine Instanz ist ein individuelles und eigenständiges Beispiel für das Konzept, welches der Algorithmus lernen soll. Der Bezeichner ist bei Klassifikationsproblemen die Antwort auf die Frage, zu welcher *Klasse* (engl. *class*) oder auch *Kategorie* die Instanz gehört [36]. Beispielsweise könnte eine Instanz in der Schrifterkennung das gescannte Bild eines mit der Hand geschriebenen Buchstabens sein. Der Bezeichner ist dann der Buchstabe, welcher auf dem Bild zu sehen ist. Dieser muss für den Trainingsdatensatz noch vom Menschen vorgegeben werden. Grundsätzlich ist auch die Zugehörigkeit zu mehreren Klassen möglich, die hierfür notwendigen Betrachtungen sind jedoch in dieser Arbeit nicht relevant. Wenn vor dem Lernen keine Bezeichner vergeben werden, spricht man von nicht überwachtem (engl. *unsupervised*) Lernen. Unüberwachtes Lernen wird üblicherweise bei den Clustering-Verfahren des ML angewandt [38].

Der Algorithmus kennt beim überwachten Lernen somit den Datensatz, welchen er klassifizieren soll und auch die Klasse, zu der er gehört. Seine ganze Aufgabe besteht nun darin zu lernen, wie die A*ttribute* (engl. *attributes*) der Instanz beschaffen sein müssen, um auf eine bestimmte Klasse schließen zu können [37]. Attribute sind Werte, welche unterschiedliche Aspekte der Instanz beschreiben. Beispiele für Attribute in der Schrifterkennung könnten sein, ob das Bild des Buchstabens viele Rundungen oder einen geschlossenen bzw. offenen Kreis enthält. Attribute nehmen häufig numerische oder boolesche, aber auch nominale Werte an, wenn die Zugehörigkeit zu einer Kategorie ausgedrückt werden soll [37]. Man spricht von einem Attributvektor, wenn mehrere Attribute in einem mathematischen Konstrukt zusammengefasst werden.

Um ein möglichst genaues Modell (bei diesem Lerntyp häufig auch *Klassifizierer*, engl. *classifier,* genannt) zu erhalten, sollten die Attribute so gewählt werden, dass sie den wichtigsten *Merkmalen* (engl. *features*) des Objektes entsprechen, welches klassifiziert werden soll. Im Maschinellen Lernen ist die Attributauswahl ein eigenes umfangreiches Feld, welches in dieser Arbeit nicht weiter betrachtet werden soll.

2.6.2 Der Data-Mining-Prozess

Ein Begriff, der ebenso eng mit Maschinellem Lernen verknüpft ist wie der Begriff KI, ist Data Mining. Data Mining verfolgt das Ziel, in großen, meist schwach strukturierten, Datenmengen Strukturen, Regelmäßigkeiten oder verborgene Zusammenhänge zu erkennen [11]. Wiederum stellt das Maschinelle Lernen nur eine Methode dar, um dieses Ziel zu erreichen. Da auch diese Arbeit in den Bereich Data Mining, fällt wurde der *Standardprozess für das Data Mining* (engl. *Cross Industry Standard Process for Data Mining* (*CRISP-DM*)) als Grundlage für das Vorgehen gewählt. Der auf diesem Gebiet neuere Standard *Analytics Solutions Unified Method for Data Mining/Predictive Analytics* (*ASUM-DM*) ist noch stärker auf den Einsatz von ML in der Wirtschaft ausgerichtete und wird daher aus Sicht des Autors nicht der akademischen Ausrichtung dieser Arbeit gerecht.

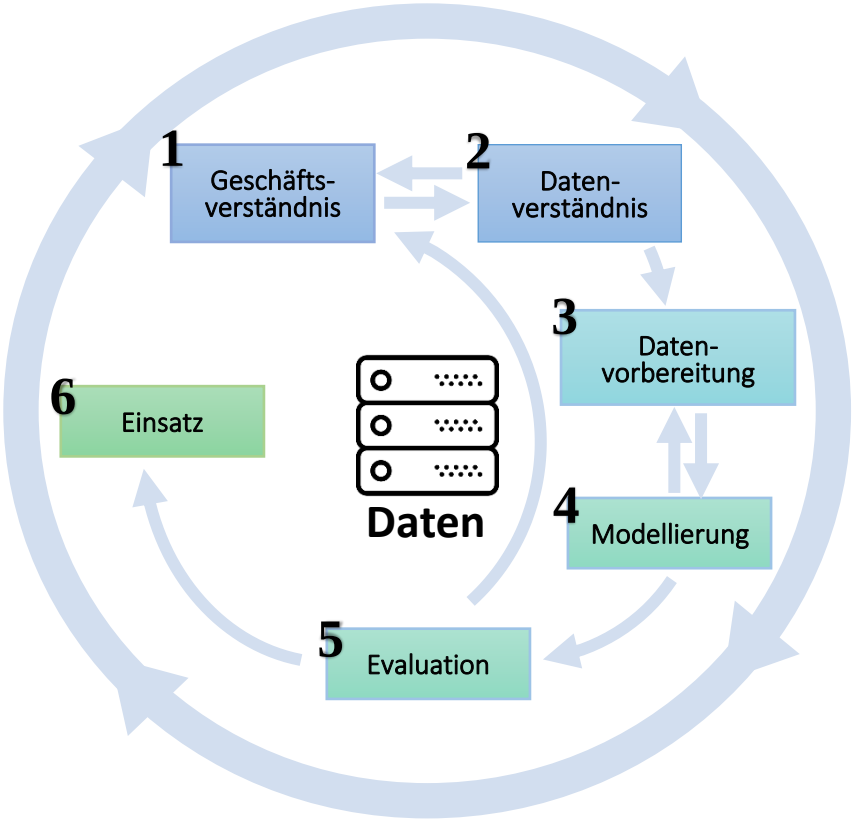


Abb. 2-4 Cross Industry Standard Process for Data Mining (nach [39])

Ziel von CRISP-DM ist es, einen umfassenden Prozess für die Durchführung von Data-Mining-Projekten zu bieten. Dazu wird der Lebenszyklus des Projektes in sechs Phasen zerlegt. In Abb. 2-4 sind diese Phasen in den farbigen Kästchen dargestellt. Die Pfeile zwischen den Phasen deuten die üblichen Übergänge zwischen ihnen an. Der äußere Kreis nimmt Bezug auf die zyklische Natur von Data-Mining-Projekten und illustriert, wie aus den über das Projekt gewonnen Erfahrungen ein neues, genaueres Geschäftsverständnis entsteht [39].

Im Folgenden wird kurz umrissen, was das Ziel der einzelnen Phasen ist und in welchen Abschnitten dieser Arbeit deren wesentliche Aufgaben behandelt und ihre Ergebnisse dargestellt werden.

1. **Geschäftsverständnis**
   In der initialen Phase des Projektes geht es darum, die Anforderungen, welche sich meistens aus einem Geschäftsbetrieb entwickeln, zu verstehen und dieses Verständnis in die Definition eines Data-Mining-Problems umzusetzen. Außerdem muss die aktuelle Situation zu Beginn des Projektes ausreichend bekannt sein, um bspw. den Ressourcen-Bedarf abschätzen zu können [39].
   Sowohl die Geschäftsziele (Erkennen von Verschlüsselungstrojanern) als auch die Data-Mining-Ziele (Forschungsfragen) sind in der Einleitung (siehe Abschnitt 1) dieser Arbeit beschrieben. Die aktuelle Situation wird in Abschnitt 3, speziell in Unterabschnitt 3.1 beschrieben.
2. **Datenverständnis**
   In der zweiten Phase werden die Ausgangsdaten für das Data Mining gesammelt, untersucht und ihre Qualität gesichert [39].
   Abschnitt 3.2 Datenerhebung beschreibt die hierfür notwendigen Tätigkeiten und ihre Ergebnisse.
3. **Datenvorbereitung**
   Diese Phase umfasst alle Aktivitäten, welche die Erzeugung des finalen Datensatzes aus den anfänglich in Phase 2 gesammelten Roh-Daten betrifft. Unter anderem sind dies: Datenselektion, Datenaufbereitung und –integration sowie die Formatierung der Daten. Durchschnittlich werden 50% – 70% der Gesamtprojektdauer für diese Aktivitäten verwandt [39].
   Diese Schritte sind in Abschnitt 3.3 Merkmalextraktion umfassend dargestellt.
4. **Modellierung**
   Das Auswählen der richtigen Lern-Technik, des entsprechenden Algorithmus, das Design sowie die Durchführung von Tests (z. B. mittels Testdatensätzen) für das Modell und dessen Erstellung sind die Bestandteile dieser Phase [39].
   In Abschnitt 3.4 bis Abschnitt 4 wird erläutert, wie die Umsetzung dazu in dieser Arbeit konkret aussieht.
5. **Evaluation**
   Bevor das Modell praktisch eingesetzt werden kann, muss es umfassend auf Erreichung der in Phase 1 aufgestellten Geschäftsanforderungen geprüft werden. Hierbei handelt es sich nicht um Tests, ob das Modell die üblichen Performance-Maßstäbe im ML erreicht, sondern darum, ob es dem vorgesehenen Einsatzweck genügt [39].
   Die Abschnitte 5 und 6 stellen die Antwort auf diese Frage dar.

6. **Einsatz**
   In dieser Phase wird das Modell in den produktiven Einsatz überführt. Dies wird häufig vom *Konsumenten* des Modells durchgeführt. Der *Data Analyst*, welcher in den vorherigen Phasen federführend war, übernimmt jetzt nur eine unterstützende Rolle. Die Arbeitsschritte umfassen das Planen der Überführung, das Überwachen und Warten des im Einsatz befindlichen Modells sowie eine Retrospektive auf das Projekt [39].
   Obwohl der produktive Einsatz der für diese Arbeit erzeugten Modelle nicht zu deren Fokus gehört, umfasst der Abschnitt 6 einige Ergebnisse, die zu dieser Phase gehören.

Abschließend soll angemerkt werden, dass der Data-Mining-Prozess iterativer Natur ist. Es ist nicht ungewöhnlich, zu einer früheren Phase des Prozesses zurückzukehren. Fehlerhafte oder schlechte Ergebnisse können ein Grund dafür sein. Der in dieser Arbeit dargestellte Data-Mining-Prozess und dessen Ergebnisse mögen dem Leser wasserfall-artig anmuten. Die praktische Umsetzung war jedoch von vielen Schleifen geprägt, die der Autor drehen musste. Diese werden nur soweit dargestellt, wie sie zum Verständnis der Ergebnisse bzw. des beschrittenen Weges notwendig sind.

2.6.3 Arten von Klassifikationsalgorithmen

Viele Probleme des Data Minings lassen sich so transformieren, dass sie mit den Klassifikationsalgorithmen des ML gelöst werden können. Dabei wird z. B. die Frage nach dem Kreditrisiko neuer Kunden einer Bank auf die binäre Klassifikationsfrage reduziert, ob diese *gute* oder *schlechte* Kunden sein werden [39]. Beantworten lässt sich diese Frage mit einem oder mehreren der mannigfaltig vorhandenen Klassifikationsalgorithmen. Diese lassen sich grob nach ihren Funktionsprinzipien in unterschiedliche *Familien* clustern. Jede Familie, aber auch jeder Algorithmus, hat individuelle Vor- und Nachteile in bestimmten Anwendungsfällen. Die Performanz und die Eignung eines Algorithmus können deswegen selbst mit viel Erfahrung oft nicht abschließend von vornherein beantwortet werden. Je nach Größe und Dimensionalität des zu untersuchenden Datensatzes kann es deswegen sinnvoll sein, diesen einer nicht zu kleinen Auswahl an Algorithmen vorzulegen. Anschließend wird dann der effektivste Algorithmus ausgewählt.

Ein solcher Ansatz wird auch in dieser Arbeit verfolgt. Dementsprechend breit ist die Auswahl an Familien und groß die Anzahl der aus ihnen gewählten Klassifikationsalgorithmen. Aus diesem Grund können die im Folgenden vorgestellten Algorithmen nur eine oberflächliche Betrachtung erfahren. Eine genaue Vorstellung würde den Rahmen dieser Arbeit bei weitem sprengen. Der Leser sei jedoch auf die Quellen zu dieser Arbeit verwiesen. Diese stellen in der großen Menge an zur Verfügung stehender Literatur einen guten Einstieg in das Thema Data Mining und ML dar.

Die Benennung der im Folgenden vorgestellten Klassifizierer-Familien und deren Algorithmen orientiert sich an den in der genutzten ML-Software *Weka* (siehe Abschnitt 3.4.1) verwendeten Namen.

2.6.3.1 Bayessch

Ein bayesscher Klassifizierer ist ein wahrscheinlichkeitsbasiertes Klassifikationsverfahren, das auf dem *Satz von Bayes* basiert. Dieses Theorem beschreibt die Berechnung bedingter Wahrscheinlichkeiten. Mit ihm kann die Wahrscheinlichkeit für ein Ereignis berechnet werden, wenn bekannt ist, dass ein zweites Ereignis bereits eingetreten ist.

Den meisten Algorithmen dieser Familie ist gemeinsam, dass sie von einer stochastischen Unabhängigkeit der Attribute einer vorgelegten Instanz ausgehen. Diese kann in praktischen Fällen nicht gegeben sein. Es entstehen dann falsche Ergebnisse, die auf den multiplikativen Einfluss der voneinander abhängigen Attribute auf das Endergebnis zurückzuführen sind. Somit muss besonderes Augenmerk auf die Attributselektion gelegt werden. Ein Vorteil dieser Verfahren ist, dass die Trainingsdaten meist nur einmal durchlaufen werden müssen und die Algorithmen auch mit Lücken in den Attributen der Instanzen umgehen können. Auch der Umfang an Trainingsdaten ist vergleichsweise gering, um mit ihnen gute Ergebnisse zu erzielen [40].

Die Familie von ML-Algorithmen erfährt umfangreiche Nutzung bei der Klassifikation von Dokumenten bspw. zu bestimmten Themen [37].

In dieser Arbeit werden folgende zwei Implementierungen der bayesschen ML-Algorithmen genutzt:

- **NaiveBayes**: Die Implementierung eines einfachen ML-Algorithmus, welcher jeder Klassenzuordnung eine Wahrscheinlichkeit zuweist. Er geht von einer inhärenten Unabhängigkeit der Attribute aus.
- **BayesNet**: Eine Erweiterung des NaiveBayes-Algorithmus, welcher auch mit inhärenten Abhängigkeiten in den vorgelegten Datensätzen umgehen kann. Er erzeugt ein bayessches Netz, in dem die Knoten Zufallsvariablen und die Kanten bedingte Abhängigkeiten zwischen den Variablen beschreiben.

2.6.3.2 Funktionsbasiert

Funktionsbasierte ML-Algorithmen lassen sich auf natürliche Weise als mathematische Gleichungen darstellen. Dies gilt für viele andere Klassifizierer-Familien nicht (z. B. Entscheidungsbäume und Regelbasierte Klassifizierer; der NaiveBayes-Algorithmus bildet wiederum eine Ausnahme, da er eine einfache mathematische Formulierung zulässt) [37]. Viele Algorithmen dieser Familie basieren auf der *linearen Dis-*

kriminanzanalyse. Sie versuchen also eine optimale lineare Grenze in Form einer Geraden (im höherdimensionalen auch *Hyperebene* genannt) zwischen die in einem Merkmalsraum abgebildeten Instanzen zweier Klassen zu legen. Diese Grenze kann im Folgenden genutzt werden, um Vorhersagen zur Klassenzugehörigkeit neuer Instanzen zu treffen [40].

Funktionsbasierte Klassifizierer lassen sich hervorragend auf alle Instanzen mit numerischen Attributen anwenden. Bedingung hierfür ist aber, dass die Klassengebiete, also die Abschnitte des Merkmalsraums, welche eindeutig einer Klasse zugeordnet werden können, Gauß-verteilt sind. Bei anderen Verteilungen lässt sich häufig keine eindeutige lineare Grenze zwischen die Klassen legen, sodass die Performanz des Klassifizierers ungenügend ist [40]. Diese Herangehensweise ist dafür jedoch vergleichsweise effizient.

Eine Besonderheit in dieser Familie von Klassifizierern stellen die *Support Vector Machines* (SVM) dar. Sie basieren ebenfalls auf der linearen Diskriminanzanalyse, erlauben es aber über den sogenannten *Kernel-Trick,* auch nichtlineare Klassengrenzen zu finden. Das Wort Kernel oder Kern wird in der Mathematik verwendet, um eine Gewichtungsfunktion für eine Summe oder ein Integral zu bezeichnen. Mittels des Kernels werden die Trainingsdaten implizit in Daten mit höherer Dimension transformiert. Dies vereinfacht ihre Struktur und erlaubt es, in dem höher-dimensionalen Raum eine lineare Klassengrenzen zu finden, die in dem nieder-dimensionalen Raum nicht vorhanden waren. Die Position der Daten im höher-dimensionalen Raum muss jedoch nie konkret berechnet werden. Der Aufwand für die Diskriminanzanalyse beim Einsatz eines Kernels steigt deswegen nicht so stark, wie die Erhöhung der Dimensionen vermuten lässt. SVMs sind jedoch grundsätzlich rechenaufwändiger als rein lineare Klassifizierer [40].

In dieser Arbeit werden folgende vier Implementierungen der funktionsbasierten ML-Algorithmen genutzt:

- **Logistic**: Basiert auf der aus der Statistik bekannten logistischen Regression, welche boolesche Werte bei der Vorhersage auf Basis vorgelegter Daten zurückgibt.
- **Sequential Minimal Optimization (SMO)**: Eine Implementierung des SMO-Algorithmus zum Training von SVMs, wenn gewünscht unter Einsatz unterschiedlicher Kernels.
- **LibSVM**: Ein Zusatz-Modul für Weka, welches über den *Package-Manager* nachgeladen werden kann. Ermöglicht die Erstellung von nichtlinearen Klassifizierern auf Basis von SVMs. Hat eine höhere Flexibilität z. B. bezüglich Parameter-Wahl als Wekas Standard-SMO-Implementierung und rechnet schneller als diese. Unterstützt unterschiedliche Kernels.

- **LibLINEAR**: Eine Implementierung des SMO-Algorithmus als Zusatz-Modul für Weka, welches über den Package-Manager nachgeladen werden kann. Ermöglicht üblicherweise schnellere Berechnungen im Vergleich zum vorinstallierten SMO. Arbeitet auf Basis linearer und logistischer Regression.

2.6.3.3 Meta

Wie der Name vermuten, lässt sind in dieser Familie Verfahren versammelt, welche die Algorithmen anderer Familien erweitern und verbessern. So können z. B. Probleme ausgeglichen werden, die sich aus zu kleinen Trainingsdatensätzen oder der inhärenten Instabilität der Algorithmen gegen kleine Veränderungen in deren Attributen ergeben [37].

Modelle, die ohne Behandlung dieser Probleme erstellt werden, neigen häufig zu einer *Über-* oder *Unteranpassung* (engl. *over-/underfitting*). Einer der Gründe für Überanpassung ist ein Modell, welches den Trainingsdatensatz *zu gut* nachbildet. Das heißt, eine Generalisierungsleistung, die für eine korrekte Klassifikation neu vorgelegter Instanzen häufig notwendig ist, kann nicht mehr geleistet werden. Bei der Unteranpassung wiederum kann der Algorithmus aus den Trainingsdaten kein Modell erstellen, welches ausreichend genau genug ist, um erfolgreich eingesetzt zu werden [41].

Um solcherlei Probleme zu vermeiden, wird auf das *Lernen im Kollektiv* zurückgegriffen (engl. *ensemble learning*). Das heißt, es werden viele unterschiedliche Modelle auf Basis der Trainingsdaten erzeugt. Ihr Unterschied liegt entweder in den genutzten ML-Algorithmen oder in der Auswahl der Instanzen für das Training. Bei der Klassifikation neu vorgelegter Instanzen wird dann eine (gewichtete) Mehrheitsentscheidung aller Modelle genutzt. Daneben gehört auch das iterative Verbessern eines einzelnen Modells zur Familie der Meta-Klassifizier [37].

Ganz ähnlich zu Situationen, in denen mehrere Menschen gemeinsam versuchen ein Problem zu lösen und jeder mit seinen Stärken zu Lösungsfindung beiträgt, ist die Performanz von Lösungen auf Basis von Meta- Klassifizierern besser als diejenige individueller Modelle. Herausforderungen ergeben sich heutzutage häufig noch, bei der Erklärung der Ergebnisse. Besonders, wenn eine Vielzahl unterschiedlicher Algorithmen in einem Meta-Klassifizierer zusammengefasst wird, kann es schwierig sein, die Entscheidung der Modelle zu verstehen [37].

In dieser Arbeit werden folgende drei Implementierungen von Meta-ML-Verfahren genutzt:

- **AdaBoostM1**: Ist ein Verfahren, welches über das iterative Training desselben Modells eine Verbesserung seiner Vorhersage-Performanz erreichen möchte. Dazu wird ein Modell erstellt und anhand des Trainingsdatensatzes geprüft.

Die dabei eventuell fehlerhaft klassifizierten Instanzen werden mit einem höheren *Gewicht* (einem hierfür eingeführten numerischen Wert) versehen und das Training erneut durchgeführt. Die höher gewichteten Instanzen haben nun mehr Einfluss auf die Erstellung des Modells. Der Vorgang wird wiederholt, bis eine definierte Fehlerschwelle unterschritten wird oder sich nicht mehr signifikant verändert. Dieses Verfahren wird häufig bei *Unteranpassung* angewandt.

- **LogitBoost**: Während der AdaBoost-Algorithmus die Fehlerhäufigkeit bei der Vorhersage der Klassen im Trainingsdatensatz vermindert, versucht LogitBoost die Zuverlässigkeit der Vorhersage zu erhöhen, indem es den Durchschnitt über beliebig viele erstellte Modelle desselben Regression-Algorithmus erstellt und dabei eine logistische Verlust-Funktion minimiert. Die Modelle werden, wie bei AdaBoost, mit individuell unterschiedlich gewichteten Instanzen trainiert. Ihr Gewicht wird bei falscher Vorhersage erhöht. Dieses Verfahren wird häufig bei *Unteranpassung* aufgrund von schwachen Klassifizierern angewandt.
- **Bagging** (kurz für *bootstrap aggregation*): Ist ein Verfahren, bei dem eine Vielzahl von Modellen desselben ML-Algorithmus mit unterschiedlichen Aufteilungen des Trainingsdatensatzes erstellt werden. Dabei kann der komplette Trainingsdatensatz gleichmäßig auf die Anzahl der lernenden Klassifizierer verteilt werden oder es können zufällige Stichproben aus ihm erstellt werden. Die Klassifikation wird im Anschluss über eine Mehrheitsentscheidung der Modelle gefällt. Dieses Verfahren wird häufig bei *Über-* oder *Unteranpassung* angewandt.

2.6.3.4 Entscheidungsbäume

Entscheidungsbaum-basierte Klassifizierer erstellen auf Basis des Trainingsdatensatzes meist nicht-zirkuläre Baumstrukturen. Als Knoten enthalten diese den Namen eines Attributes. Die Anzahl der Kinder des Knotens ist abhängig davon, welche Werte das Attribut des Vater-Knotens annehmen kann. Bei nominalen Attributen entspricht die Zahl der Kinder meist der Anzahl an möglichen Werten des Attributs. Im Falle von numerischen Attributen gibt es eine Zwei-Wege-Teilung, wenn sein Wert *größer* bzw. *kleiner* als eine vom Algorithmus vorgegebene Konstante ist. Alternativ kann auch eine Drei-Wege-Teilung verwendet werden (z. B. im Falle von *größer*, *kleiner* und *gleich*). Die Blätter des Baumes enthalten die Namen der möglichen Klassen oder eine Wahrscheinlichkeitsverteilung über alle Klassen. In welcher Ebene ein bestimmtes Attribut als Knoten in den Baum aufgenommen wird, obliegt dem Algorithmus. Sie versuchen meist die Attribute als Erstes zu prüfen, welche den größtmöglichen Informationsgewinn (im Sinne von Einschränkung auf weniger Klassen) bieten [37].

Bei der Klassifikation wird eine Instanz durch den Entscheidungsbaum hindurchgeführt. An jedem Knoten wird auf Basis ihrer Attribute entschieden, welchen Weg die Instanz weiter durch den Baum nimmt. Zum Schluss erreicht die Instanz ein Blatt des Baumes, welches ihr eine Klasse zuweist [37].

Ein wichtiges Konzept, welches in Zusammenhang mit Entscheidungsbaum-basierten Klassifizierern hier genannt werden soll, ist das *Stutzen oder Vereinfachen der Bäume* (engl. *pruning*). Da voll entwickelte Entscheidungsbäume häufig unnötige Äste besitzen, ist es sinnvoll, diese zu entfernen, bevor sie produktiv eingesetzt werden. Unnötige Äste entstehen beim Training, sind aber z. B. der Klassifikation bisher ungesehener Instanzen abträglich (siehe Überanpassung im Abschnitt 2.6.3.3). Auch wenn ein Stutzen des Baumes die Performanz auf dem Trainingsdatensatz verringert, kann seine Performanz im produktiven Betrieb so wesentlich verbessert werden. Grundsätzlich wird zwischen dem *Stutzen während der Trainings* (engl. *pre-pruning*) und dem Stutzen nach *Erstellung des Baums* (engl. *post-pruning*) unterschieden. Das *pre-pruning* wird bspw. durch eine vorgegebene Tiefe des Baumes durchgeführt, bis zu der der Baum wachsen darf. Das *post-pruning* wird über unterschiedliche Verfahren zur Bestimmung der Relevanz von Ästen gesteuert und anschließend z. B. über die Ersetzung eines Asts durch einen Knoten umgesetzt [37].

Ein großer Vorteil von Entscheidungsbaum-basierten Algorithmen ist ihre leichte Verständlichkeit. Die erzeugten Bäume können für den Benutzer ausgegeben und die Klassifikation sogar manuell nachvollzogen werden. Ihre Schwächen liegen vor allem bei Datensätzen mit numerischen Attributen, da das diskrete Regelwerk der Bäume meist zu schlechterer Performanz führt als andere Algorithmen des ML [37].

In dieser Arbeit werden folgende zwei Implementierungen von Entscheidungsbaum-Algorithmen genutzt:

- **J48**: Dieser Algorithmus ist eine Java-Implementierung des bekannten quelloffenen C4.5-Algorithmus. Seine Funktionsweise entspricht den obigen Darstellungen dieses Abschnitts.
- **Random Forest**: Hierbei handelt es sich eigentlich um die Implementierung eines Bagging-Verfahrens. Random Forests werden von *Weka* aber in der Entscheidungsbaum-Kategorie geführt, da sie nur ML-Algorithmen aus dieser Kategorie in dem von ihnen erstellten Kollektiv zulassen. Beim Training wird eine beliebig große Anzahl von Entscheidungsbäumen erstellt. Grundlage sind jeweils zufällige Stichproben des Trainingsdatensatzes. Zusätzlich kann beim Erstellen des einzelnen Baums an jedem Knoten ein zufälliges Attribut aus einer Gruppe gewählt werden. Die Gruppe wird gebildet, indem die Attribute nach ihrem Informationsgewinn für den Baum geordnet und nur die obersten *n* Elemente bei der Auswahl für den Knoten betrachtet werden. Dies verringert

die Ähnlichkeit zwischen den Bäumen des Random Forests. Die Klassifikation wird, wie beim Bagging üblich, über eine Mehrheitsentscheidung aller Bäume getroffen.

2.6.4 Maße für die Leistungsfähigkeit eines Klassifizierers

Um eine Aussage über die Leistungsfähigkeit eines Modells treffen zu können, muss es getestet werden. Im CRISP-DM-Prozess (siehe Abb. 2-4) wird dies in Phase 4 „*Modellierung*“ erledigt, also direkt, nachdem das Modell erstellt wurde. Ein Unterschied dieses Arbeitsschrittes besteht zur Phase 5 *„Evaluation“*. Dort wird überprüft, ob das Modell für den Anwendungszweck angemessen ist. Die Leistungskennzahlen sind ein erster Indikator dafür, aber erst eine Evaluation mit dem Nutzer des Modells bringt hier ein abschließendes Ergebnis [39].

Den Kern dieser Arbeit bilden Klassifizierer des Maschinellen Lernens. Aus diesem Grund wird hier nur auf Kennzahlen eingegangen, welche im Data Mining die Performanz eines Klassifizierers abbilden. Die Fehlerrate (engl. *error rate*) ist ein typisches Maß in diesem Bereich. Andere Lerntypen, wie z. B. die Regression, werden häufig auch mit anderen Kennzahlen bewertet.

Die *Resubstitutionsfehlerrate* (engl. *resubstitution error rate*) ist ein Maß für die Leistung des Modells bei der Klassifikation des eigenen Trainingsdatensatzes. Hier werden dem Modell also dieselben Daten zur Klassifikation vorgelegt, mit denen es davor trainiert wurde. Der Klassifizierer sagt die Klasse jeder Instanz voraus: Wenn sie korrekt ist, wird dies als Erfolg gewertet; wenn nicht, ist es ein Fehler. Die Fehlerquote ist der Anteil der Fehler, die über eine Reihe von Instanzen gemacht werden, und sie misst die Gesamtleistung des Klassifizierers [37].

Wie bereits in Abschnitt 2.6.2 angesprochen, ist die Klassifikation von bisher unbekannten Daten die größte Herausforderung für ein ML-Modell. Die Erfolgsraten bei der Klassifikation von bekannten Daten und unbekannten Daten unterscheiden sich daher meist wesentlich. Somit kann der Resubstitutionsfehler nur der erste Schritt hin zu einer Aussage über die Leistungsfähigkeit eines Modells sein. Die Qualität eines Modells zeigt sich erst, wenn ihm Testdaten (also unbekannte Daten) vorgelegt werden.

Im Allgemeinen gilt, je größer der Trainingsdatensatz, desto besser der Klassifizierer, obwohl die Verbesserung ab einer bestimmten Menge an Trainingsdaten zu sinken beginnt. Wenn viele Daten verfügbar sind, gibt es keine Probleme für Training und anschließendes Testen der Modelle. Die eigentliche Herausforderung liegt darin, mit begrenzten Daten sowohl einen *guten* Klassifizierer anzulernen als auch gleichzeitig noch genügen Testdaten übrig zu haben. Da die Datenvorbereitung, z. B. aufgrund einer manuellen Kategorisierung der Daten, den größten Anteil (ca. 50%- 70%, siehe

Abschnitt 2.6.2 - CRISP-DM Phase 3) am Data-Mining-Projekt einnimmt, ist das Vorhandensein von genügend Daten in vielen Fällen nicht gegeben. Eine übliche Methode, dieser Herausforderung zu begegnen, ist das Aufteilen des vorhandenen Datensatzes in zwei Drittel Trainings- und ein Drittel Testdaten. Dieses Vorgehen wird als *Zurückhalten* (engl. *holdout*) bezeichnet [37].

2.6.4.1 Zurückhalten, Schichtung und Kreuzvalidierung

Damit das Training erfolgreich sein kann, müssen beide Datensätze jedoch repräsentativ sein. Beispielsweise könnten sich bei zufälliger Aufteilung alle Instanzen einer Klasse im Testdatensatz befinden. Dies würde zwangsläufig zu einer schlechten Leistung des Modells führen. Durch geschichtete (engl. *stratified*) Zufallsproben, also Proben, in denen alle Klassen gleichmäßig vorkommen, kann diesem Problem begegnet werden. Dies gilt jedoch nur für die Klassen-Verteilung. Damit die beiden Datensätze wirklich repräsentativ sind, müssen jedoch auch alle anderen Merkmale der Instanzen gleichmäßig verteilt werden [37].

Die geschieht durch ein wiederholtes Zurückhalten der geschichteten Zufallsproben. Der Ausgangsdatensatz wird dazu mehrmals zufällig in die beiden benötigten Datensätze aufgeteilt und wiederholt getestet. Schließlich wird der Durchschnitt über alle Leistungskennzahlen der Tests gebildet. Dieses Endergebnis führt zu einer statistisch wesentlich repräsentativeren Aussage über die Leistungsfähigkeit des Modells [37].

Die dafür im Bereich ML meist angewandte Methode ist die sogenannte *geschichtete zehnfache Kreuzvalidierung* (engl. *stratified 10-fold cross-validation*). Dazu wird der Datensatz zufällig in 10 Teile zerlegt (engl. *folds*), von denen jeder einmal für den Test zurückgehalten wird, während der Rest für das Anlernen des Modells Verwendung findet (siehe Abb. 2-5: Iteration 1 bis 10). Eine Schichtung der Daten bezüglich der Klassen wird ebenfalls berücksichtigt (siehe Abb. 2-5: Gleichverteilung der Klassen im Testdatensatz). Dieser Vorgang wird somit insgesamt zehnmal durchgeführt und die Ergebnisse zum Schluss gemittelt. Ausführliche Tests mit zahlreichen verschiedenen Datensätzen und unterschiedlichen Techniken des ML haben gezeigt, dass Zehn die richtige Anzahl von Teilen ist, um die beste Fehlerabschätzung zu erhalten. Gleichzeitig es gibt auch einige theoretische Beweise, die dies unterstützen. Obwohl diese Argumente keineswegs schlüssig sind und in Kreisen des Maschinellen Lernens und des Data Mining weiterhin eine Debatte darüber geführt wird, welches das beste Bewertungsschema ist, ist die zehnfache Kreuzvalidierung in der Praxis zur Standardmethode geworden. Die angewandte Schichtung der Daten führt ebenfalls zu einer leichten Verbesserung des Ergebnisses. Durch die Randomisierung des Datensatzes kann sich das Ergebnis dieses Prozesses durchaus verändern, je häufiger man es

durchführt. In der Praxis wird deswegen häufig der Durchschnitt von zehn Wiederholungen der *geschichteten zehnfachen Kreuzvalidierung* herangezogen, um eine abschließende repräsentative Aussage über einen Klassifizierer zu treffen [37].

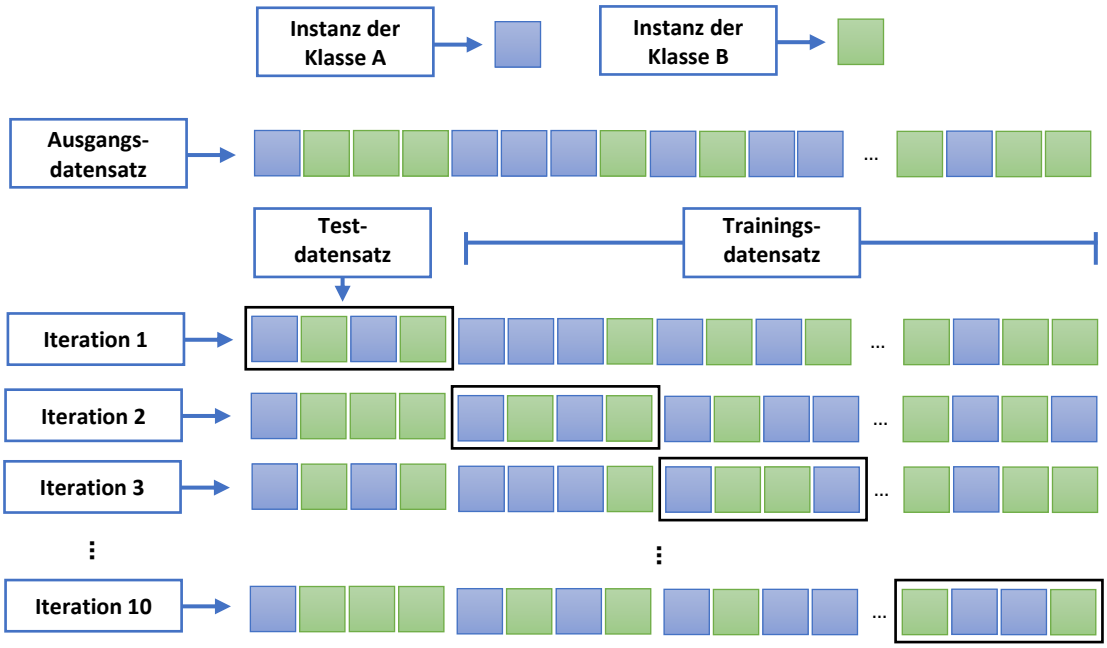


Abb. 2-5: Visualisierung der geschichteten zehnfachen Kreuzvalidierung

Nachdem nun erläutert wurde, wie der Prozess des Testens eines Modells konkret aussehen kann, wird in den folgenden Abschnitten dieses Kapitels genauer auf die Leistungskennzahlen eingegangen, die dabei gemessen werden.

2.6.4.2 Die Wahrheitsmatrix und abgeleitete Leistungsmaße

Im Maschinellen Lernen gibt es viele Leistungsmaße für die Modelle. So gut wie alle leiten sich aus der sogenannten *Wahrheitsmatrix* (engl. *confusion matrix*) ab. Diese stellt im Falle der Klassifikation die vorhergesagte Klasse einer Instanz ihrer wahren Klasse gegenüber (siehe Tab. 2-1).

Tab. 2-1: Wahrheitsmatrix beispielhaft für Klasse A

		Wahre Klasse	
		Klasse A (Instanz positiv)	Andere Klasse (Instanz negativ)
Vorhergesagte Klasse	Klasse A (Test positiv)	*Richtig Positiv*	*Falsch Positiv*
	Andere Klasse (Test negativ)	*Falsch Negativ*	*Richtig Negativ*

Für die Aufstellung der Matrix werden alle Instanzen eines Testdatensatzes durch ein Modell klassifiziert und ihre Zugehörigkeit zu einer bestimmten aller vorhersagbaren Klassen wird prognostiziert. Eine Instanz, deren Zugehörigkeit zu dieser Klasse vorhergesagt wird, wird als *positiv* bezeichnet. Andernfalls ist die Instanz *negativ*. Die Ergebnisse des Tests werden anschließend mit der wahren Zugehörigkeit einer Instanz verglichen und als Häufigkeiten in der Matrix eingetragen. Es ergeben sich somit vier mögliche Fälle für die Ergebnisse der Vorhersage [41]:

- **Richtig Positiv** (RP): Der Test sagt die Zugehörigkeit zur betrachteten Klasse voraus und die Instanz gehört zu dieser Klasse.
- **Falsch Positiv** (FP): Der Test sagt die Zugehörigkeit zur betrachteten Klasse voraus, aber die Instanz gehört zu einer anderen Klasse.
- **Falsch Negativ** (FN): Der Test sagt die Zugehörigkeit zu einer anderen als der betrachteten Klasse voraus, aber die Instanz gehört zur betrachteten Klasse.
- **Richtig Negativ** (RN): Der Test sagt die Zugehörigkeit zu einer anderen als der betrachteten Klasse voraus und die Instanz gehört auch zu einer anderen Klasse.

Häufig angewendete Maße zur Beurteilung von Klassifizierern, welche aus der Wahrheitsmatrix abgeleitet werden, sind die *Richtig-Positiv-Rate* (auch *Sensitivität* genannt, engl. *recall* oder *hit rate*) und die *Falsch-Positiv-Rate* (engl. *fallout*). Beide können als bedingte Wahrscheinlichkeiten für das Eintreten des entsprechenden Ereignisses interpretiert werden [37].

1. **Richtig-Positiv-Rate** (RPR): Gibt den Anteil der korrekt als positiv klassifizierten Instanzen an der Gesamtheit der in Wahrheit positiven Instanzen an. Ist ein Maß dafür, wie sensitiv der Klassifizierer gegenüber den relevanten positiven Instanzen ist, also wie viele er als solche erkennt.

$$\boldsymbol{RPR}\colon P(positiver\ Test \mid Instanz\ positiv) = \frac{RP}{RP + FN} \qquad (2.1)$$

2. **Falsch-Positiv-Rate** (FPR): Gibt den Anteil der fälschlicherweise als positiv klassifizierten Instanzen an der Gesamtheit der in Wahrheit negativen Instanzen an.

$$\boldsymbol{FPR}\colon P(positiver\ Test \mid Instanz\ negativ) = \frac{FP}{FP + RN} \qquad (2.2)$$

Diese beiden Maße ermöglichen es, wichtige Eigenschaften eines Klassifizierers in einer Zahl darzustellen. So ist z. B. gerade in der Erkennung von Viren die FPR eine wichtige Kennzahl. Sie gibt an, wie häufig der Virentest Alarm schlägt, obwohl kein Befall des Systems vorliegt. Für den Anwender spielt dies eine große Rolle, da jeder Alarm bearbeitet werden muss. Die RPR auf der anderen Seite stellt in diesem Beispiel ein hervorragendes Maß dafür dar, wie gut sich ein Virenscanner im Vergleich mit

anderen bei der Erkennung von Bedrohungen schlägt. Sie ist also ein Indiz dafür, wie sehr sich die Nutzer auf den Scanner verlassen können.

Zusammen stellen sie die Ausgangsgrößen zur Berechnung der *Grenzwertoptimierungskurve* (auch *ROC-Kurve* genannt, engl. *receiver operating characteristic* (ROC)) dar, welche in Abschnitt 2.6.4.3 genauer erläutert wird.

Als letzte direkt aus der Wahrheitsmatrix abgeleitete Größe soll hier der *positive Vorhersagewert* (engl. *precision*) vorgestellt werden. Er ist bspw. ein Maß dafür, welcher Anteil aller Alarme eines Virenscanners auf echte Bedrohungen zurückzuführen ist [41].

1. **Positiver Vorhersagewert** (PVW): Gibt den Anteil der korrekt als positiv klassifizierten Instanzen an der Gesamtheit der als positiv klassifizierten Instanzen an.

$$\boldsymbol{PVW}: P(Instanz\ positiv \mid positiver\ Test) = \frac{RP}{RP + FP} \qquad (2.3)$$

Zusammen mit der Sensitivität ist der positive Vorhersagewert eines Klassifizierers Ausgangspunkt zur Berechnung des F-Maßes, welches in Abschnitt 2.6.4.4 vorgestellt wird.

Es gibt viele weitere Maße für die Leistungsfähigkeit von ML-Modellen. In dieser Arbeit werden jedoch nur diejenigen genutzt, welche auch Cohen und Nissim [13] herangezogen haben. So soll eine Vergleichbarkeit der beiden Arbeiten erreicht und damit dem Replikationscharakter dieser Arbeit genüge getan werden.

2.6.4.3 Die Grenzwertoptimierungskurve

Neben rein numerischen Maßen für die Entscheidungsgüte eines Klassifizierers gibt es grafische. Zu diesen gehört die oben bereits kurz erwähnte *Grenzwertoptimierungskurve* (auch *ROC-Kurve* genannt). Der englische Begriff *receiver operating characteristic* stammt aus der Signalentdeckungstheorie und beschreibt den Kompromiss zwischen RPR und FPR bei der Signalsuche über einem verrauschten Medium [37].

Das Verhältnis zwischen falsch positiven und richtig positiven Vorhersagen wird dabei über einen *Schwellwert* (engl. *threshold*) beeinflusst, welcher bei vielen ML-Algorithmen leicht einführbar ist. Wenn der Klassifizierer bspw. eine Wahrscheinlichkeit für die Zugehörigkeit einer Instanz zu einer betrachteten Klasse produziert, ist der Schwellwert die konkrete Wahrscheinlichkeit, ab der die Zugehörigkeit als gegeben angesehen wird. Bei den Meta-Klassifizierern, welche per Mehrheitsentscheidung auswählen, ist der Schwellwert bspw. die Größe der Mehrheit, die für ein positives Testergebnis vorhanden sein muss. Hieran lässt sich erkennen, dass der Schwellwert beliebig verändert werden kann. So kann ein Klassifizierer im Bereich der

Schadcodeerkennung besonders empfindlich eingestellt werden, wenn die Sicherheitsanforderungen hoch sind und der Bediener mit einer hohen FPR umgehen kann [41].

Um das optimale Verhältnis zwischen FP- und RP-Vorhersagen zu finden, wird die bei unterschiedlichen Schwellwerten gemessene prozentuale FPR auf der Abszissenachse und die RPR auf der Ordinatenachse eines Graphen aufgetragen. Der Schwellwert, welcher zur höchsten RPR bei niedriger FPR führt, kann somit als Optimum angenommen werden. Je nach Einsatzszenario für den Klassifizierer kann jedoch ein anderer Schwellwert sinnvoller sein (z. B. aufgrund von Sicherheitsanforderungen). Die Diagonale dieses Graphen kann als die ROC-Kurve eines zufällig entscheidenden Klassifizierers angenommen werden. Liegt die ROC-Kurve also unter der Diagonalen, so klassifiziert das betrachtete Modell schlechter als ein Münzwurf [41].

Über eine Messung der Fläche unter der ROC-Kurve (engl. *area under curve* (AUC)) kann die Aussagekraft des Graphen auf eine einzelne Kennzahl verdichtet werden, welche dann zwischen 0,0 und 1,0 liegt. Hierbei ist 0,5 das schlechtestmögliche Ergebnis, da dies eine Kurve nahe der Diagonalen beschreibt. Je näher die Größe der AUC an 1,0 liegt, desto besser der Klassifizierer, da sich die ROC-Kurve der Ordinate annähert. Eine gegen 0,0 laufende Fläche ist aber ebenfalls als optimal zu bewerten, wenn man die Vorhersagen vertauscht, also die positiven Tests als negative Tests interpretieren lässt [37].

2.6.4.4 Das F-Maß

Das F-Maß (engl. *f-measure* oder *f1 score*) ist das harmonische Mittel aus Sensitivität bzw. Richtig-Positiv-Rate und positivem Vorhersagewert. Bei einem F-Maß von 1 sind sowohl Sensitivität als auch PVW optimal. Umgekehrt liegt die untere Grenze des F-Maßes bei einem Wert von 0.

$$\mathbf{F-Maß} = 2 \times \frac{RPR \times PVW}{RPR + PVW} \qquad (2.4)$$

Wie an Formel (2.4) zu erkennen, wird die Anzahl der richtig negativen Vorhersagen nicht durch das F-Maß abgebildet. Diese können in manchen Anwendungsfällen jedoch durchaus relevant für die Bewertung eines Klassifizierers sein. Ebenso kann es nötig sein, RPR gegenüber PVW stärker zu gewichten, da die unterschiedlichen Arten von falscher Vorhersage (FN und FP) ein Aspekt der zu lösenden Aufgabe sein können [42]. Somit ist die Geeignetheit des F-Maßes stark vom Einsatzzweck des bewerteten Klassifizierers abhängig. In dieser Arbeit wird es in seiner grundlegenden Form (siehe Formel (2.4)) genutzt, um einen Vergleich zu [13] herzustellen.

3 Material und Methode der Analyse

Cohen und Nissim stellen in [13] eine neuartige Methode vor, um auf einem vertrauenswürdigen Weg Merkmale direkt aus dem Arbeitsspeicher einer Virtuellen Maschine zu extrahieren und mittels ML-Klassifizierern auf einen Befall durch Verschlüsselungstrojaner zu überprüfen.

Die Vertrauenswürdigkeit dieses Verfahrens basiert vor allem darauf, dass die Abbilder des Arbeitsspeichers auf Basis des Hypervisors der Virtuellen Maschine gewonnen werden. Daher kann der Trojaner die Analyse des Systems nicht erkennen und auch keine spezifischen anti-forensischen Gegenmaßnahmen treffen. Die Virtuelle Maschine wird zudem als Ganzes geschützt, weil der komplette Adressbereich des Arbeitsspeichers überwacht wird. Lösungen, welche auf dynamischer und statischer Analyse basieren (z.B. mittels *yara*-Regeln, siehe dazu [1]), müssen sich immer auf einzelne verdächtige Dateien konzentrieren.

Cohen und Nissim beantworten auf Basis ihres Verfahrens folgende Forschungsfragen:

CuN-F1: Ist es möglich, eine Virtuelle Maschine effizient anhand eines handhabbaren Satzes allgemein gültiger Merkmale zu überwachen, die von Verschlüsselungstrojanern im Hauptspeicher hinterlassen werden?

CuN-F2: Sind Klassifizierer, welche auf Grundlage der allgemein gültigen Merkmale erstellt wurden, effektiv bei der Erkennung von Verschlüsselungstrojanern?

CuN-F3: Ist es möglich, basierend auf bekannten Verschlüsselungstrojanern, das Vorhandensein unbekannter Trojaner im Hauptspeicher einer VM zu erkennen?

CuN-F4: Welcher Klassifizierer bietet die besten Erkennungsergebnisse (NaiveBayes, BayesNet, J48, RandomForest, Logistic, LogitBoost, SMO, Bagging oder AdaBoost)?

CuN-F5: Können mit dieser Methode auch unbekannte Remote Access Trojaner auf einer VM erkannt werden?

(Aufgrund der besonderen Anforderungen von RAT an die Infrastruktur der Experimentumgebung, dies ist u.a. das Vorhandensein von Command-and-Control-Servern, wird diese Forschungsfrage nicht im Rahmen dieser Arbeit behandelt.)

Cohen und Nissim beantworten diese Forschungsfragen durch mehrere Experimente. Zu deren Vorbereitung werden Virtuelle Maschinen mit Verschlüsselungstrojanern infiziert bzw. ein gutartiges Programm wird auf ihnen installiert. Deren Ausführung auf der VM wird anschließend über das regelmäßige Erstellen von Hauptspeicher-Abbildern dokumentiert. Die Abbilder werden mit Volatility ausgewertet und die Er-

gebnisse mit Python-Skripten zu Attributvektoren verdichtet, um so Instanzen zu erhalten. Aus der Menge der Instanzen werden je nach Experiment unterschiedliche Test- und Trainingsdatensätzen erstellt. Mit diesen wird die Leistungsfähigkeit der ML-Algorithmen untersucht.

Die Methode von Cohen und Nissim wird in dieser Arbeit repliziert und erweitert. Ziel ist es, die in Abschnitt 1 aufgestellten eigenen Forschungsfragen F1 – F4 zu beantworten. Dazu wird vom Autor folgendes Vorgehen gewählt:

Abb. 3-1: Vorgehen des Autors

Die folgenden Abschnitte dieses Kapitels differenzieren das praktische Vorgehen des Autors. Besonderes Augenmerk liegt auf der Darstellung aller Abweichungen von der replizierten Methode sowie auf Annahmen, welche an einigen Stellen aufgrund fehlender Details [13] getroffen werden mussten.

Die erzielten Ergebnisse werden im nachfolgenden Kapitel 4 vorgestellt und im Kapitel 5 diskutiert.

3.1 Experimentumgebung

Um die in Abschnitt 1 aufgestellten Forschungsfragen beantworten zu können, müssen die von Cohen und Nissim [13] durchgeführten Experimente Eins bis Vier repliziert werden. Dazu ist im ersten Schritt der Aufbau einer Experimentumgebung notwendig. Sie besteht aus sieben Teilen:

1. Der verwendeten Hardware
2. VirtualBox in der Version 6.0.12 als Virtualisierungslösung
3. Beispiele für Verschlüsselungstrojanern
4. Beispiele für gutartige Programme
5. Selbst entwickelte Automatisierungsbibliothek in Python
6. Volatility in der Version 2.6.1 als Hauptspeicher-Analysewerkzeug
7. Weka in der Version 3.8.4 als Lösung für das Maschinelle Lernen

Die Experimentumgebung soll dazu dienen, die benötigten Hauptspeicher-Abbilder zu erzeugen und anschließend so auszuwerten, dass ein Training und der Test der ML-Algorithmen möglich werden. Gleichzeitig soll gemessen werden, wie lange die Auswertung dauert, um Aussagen über die Effizienz der Methode treffen zu können.

Die verwendete Hardware ist ein *Lenovo X1 Carbon* der sechsten Generation, mit folgender Ausstattung:

- **CPU**: Intel Core i7-8550U – Basisgeschwindigkeit: 1,99 GHz
- **RAM**: 16 GB
- **Betriebssystem**: Windows 10 Pro 64 bit
- **Massenspeicher**: 512 GB SSD

Als nächstes soll die Virtualisierungslösung *VirtualBox* vorgestellt werden. Virtualisierung bietet einige Vorteile gegenüber Experimenten direkt auf der Hardware.

3.1.1 VirtualBox

Der schon seit einigen Jahren anhaltende Trend, IT-Services in die *Cloud* zu verlegen, macht diese zu einem besonders interessanten Angriffsziel für Kriminelle. Die Basis für den Erfolg der Cloud bildet die *Virtualisierung*, welche es ermöglicht, IT-Ressourcen von der benötigten Hardware zu abstrahieren und damit wesentlich flexibler zu betreiben. Grundlage dafür ist der sogenannte *Hypervisor* (auch *virtual machine monitor*). Er wird auf dem *Host-System* (dem gastgebenden IT-System) installiert und präsentiert den *Gast-Systemen* (den Virtuellen Maschinen) eine virtuelle Ausführungsumgebung. Auf diese Weise ist es möglich, mehrere Betriebssysteme gleichzeitig auf einer Hardware laufen zu lassen. Über den Hypervisor kann frei konfiguriert werden, welche Ressourcen des Host-Systems, wie z. B. Hauptspeicher- und Festplat-

tenplatz, vom Gast-System genutzt werden können. Auch können die genutzten Ressourcen können von außen überwacht werden. Software, die innerhalb der Virtuellen Maschine läuft, kann den Hypervisor nicht verändern. Dadurch kann auch potenziell schädliche Software im Gast-System ausgeführt werden, ohne dass die Gefahr eines unkontrollierten Zugriffes auf die Hardware des Host-Systems besteht [13].

Es werden zwei Arten von *Hypervisor* unterschieden [13]:

- *Nativ* oder engl. *bare-metal*: Diese laufen direkt auf der Hardware des Host-Systems. Beispiele dafür sind *Microsoft Hyper-V* oder *VMware ESX/ESXi*.
- Engl. *Hosted*: Hierbei handelt es sich um eine Software, die in einem herkömmlichen Betriebssystem installiert wird. Beispiele dafür sind *VMware Workstation* und *Oracle VM VirtualBox*.

In [13] wird *VMware ESXi* als Virtualisierungslösung genutzt. Abweichend dazu soll in dieser Arbeit erprobt werden, ob eine Umsetzung auch mit einem *hosted Hypervisor* möglich ist. Deswegen wird in dieser Arbeit *VirtualBox* (Version 6.0.12) genutzt [43]. Diese quelloffene Software der Firma *Oracle*, bietet ähnliche Funktionen (bezüglich Steuerung des Gast-Systems, aber auch der Automatisierungsschnittstellen) wie *ESXi*, kann jedoch ohne Entrichtung von Lizenzgebühren beliebig genutzt und sogar verändert werden.

Virtualisierungslösungen sind ein interessanter Forschungsgegenstand, da sie weit verbreitet sind und daher ein beliebtes Angriffsziel abgeben. Bedient man sich jedoch einiger ihrer Funktionen, kann gleichzeitig die Analyse von in VMs ausgeführtem Schadcode wesentlich profitieren. Manche Schadcode-Familien sind in der Lage zu erkennen, ob sie in einer virtuellen Umgebung ausgeführt werden. Sie reagieren darauf, meist indem sie Teile ihres Codes nicht ausführen, um eine Analyse zu erschweren. Solcher Schadcode ist nicht Bestandteil der hier gemachten Betrachtungen.

Hypervisoren, wie der von VirtualBox, bieten z. B. die Möglichkeit, über *Schnappschüsse* (engl. *snapshots*) den Zustand des Gast-Systems zu speichern. Dies umfasst alle Ressourcen, wie z. B. Inhalt des Haupt- und Massenspeichers. Darüber hinaus werden auch die aktuellen Konfigurationsparameter des Hypervisors gespeichert. Dies sind u.a. die angeschlossenen Geräte, wie. z. B Netzwerkkarten, aber auch, ob die Virtuelle Maschine gerade lief oder ausgeschaltet war. Über die Schnappschüsse können frühere Zustände der Virtuellen Maschine einfach wiederhergestellt werden. So wird die Durchführung der im Folgenden beschriebenen Experimente erheblich erleichtert.

Der Hauptspeicher einer Virtuellen Maschine kann ebenfalls leicht ausgelesen werden. VirtualBox bietet dazu das Erstellen sogenannter *Core-Dumps* an (mittels Methode `dump_guest_core()` der Klasse `debugger`). Dies ist für jedes in VirtualBox

lauffähige Betriebssystem möglich. Ein Core-Dump ist eine Datei, die detaillierte Informationen über den Zustand der VM zu einem bestimmten Zeitpunkt in einer Form enthält, die von Entwicklern genutzt werden kann. Sie sind besonders hilfreich, um die Gründe für Anwendungsabstürze aufzuspüren, weshalb viele Systeme angewiesen werden können, einen solchen automatisch zu erstellen, wenn ein Anwendungsabsturz auftritt [45]. Das Dateiformat entspricht dem offenen und in diesem Kontext häufig genutzten *Executable and Linkable Format 64* (*ELF-64*). Es ist umfangreich in [46] dokumentiert. Die von VirtualBox erstellte ELF-Datei hat den in Code 3-1 dargestellten Aufbau und enthält einige für dieses Programm spezifische Sektionen. Am Ende der Datei befindet sich das Hauptspeicher-Abbild der VM (in Code 3-1: `[Memory Dump]`). Dieses kann anschließend aus der ELF-Datei extrahiert werden (siehe Abschnitt 3.3.1) und mit Volatility ausgewertet werden. Hierzu wird die Methode `prepare_for_analysis()` im Modul `vbox_manager` entwickelt (siehe Anhang C).

```
[ ELF 64 Header]
[ Program Header, type PT_NOTE ]
        → offset to COREDESCRIPTOR
[ Program Header, type PT_LOAD ] - one for each contiguous physical
memory range
        → Memory offset of range
        → File offset
[ Note Header, type NT_VBOXCORE ]
[ COREDESCRIPTOR ]
        → Magic
        → VM core file version
        → VBox version
        → Number of vCPUs etc.
[ Note Header, type NT_VBOXCPU ] - one for each vCPU
[ vCPU 1 Note Header ]
        [ DBGFCORECPU - vCPU 1 dump ]
[ Additional Notes + Data ] - currently unused
[ Memory dump ]
```

Code 3-1: Der Aufbau eines VirtualBox Core-Dumps [44]

Cohen und Nissim benutzen *Windows Server 2012 R2* in ihren VMs. In dieser Arbeit wird hingegen *Windows 10 Pro* aus den in Kapitel 1 dargelegten Gründen verwendet. Bei den Vorbereitungen der Experimentumgebung wird festgestellt, dass Volatility nicht mit allen Windows 10 Versionen umgehen kann. Die Analyse des Hauptspeichers war in diesen Fällen nicht möglich oder es konnten nicht alle benötigten Volatility Plugins fehlerfrei genutzt werden. Darüber hinaus ist die Analyseperformance von Volatility bei den 64 bit Versionen schlechter als bei den 32 bit Versionen. Da dies die geplante Zeit für die Analyse wesentlich verschlechtert hätte, werden sie ebenfalls nicht für diese Arbeit verwendet.

Die in Tab. 3-1 gelisteten Windows 10 Versionen werden nach aufsteigendem Alter getestet, um die jüngste vollständig und in angemessener Zeit analysierbare Version

zu finden. Für alle Versionen in Rot markierten Zeilen wird eine der obigen Einschränkungen festgestellt. Sie wird daher als nicht tauglich für diese Arbeit eingestuft. Die grünmarkierte Version ist die in dieser Arbeit (*Windows 10 Pro Update 1607 Build 14393* in *32 bit*) verwendete.

Tab. 3-1: Liste der mit Volatility auf Analysierbarkeit geprüften Windows 10 Versionen

Nummer	Windows Version	Bitness	Build-Nummer	Veröffentlichungs-datum
1	Windows 10 Pro - 1903	64 bit	18326.239	21. Mai 2019
2	„	32 bit	„	„
3	Windows 10 Pro - 1809	64 bit	17763	13. November 2018
4	„	32 bit	„	„
5	Windows 10 Pro - 1803	64 bit	17134	30. April 2018
6	„	32 bit	„	„
7	Windows 10 Pro - 1709	64 bit	16299	17. Oktober 2017
8	„	32 bit	„	„
9	Windows 10 Pro - 1703	64 bit	15063	5. April 2017
10	„	32 bit	„	„
11	Windows 10 Pro - 1607	64 bit	14393	2. August 2016
12	**Windows 10 Pro - 1607**	**32 bit**	**14393**	**2. August 2016**

Alle Experimente dieser Arbeit werden auf einer VM durchgeführt. Sie wird nach dem in Anhang A dargestellten Vorgehen aufgesetzt. Es wird darauf geachtet, dass die Maschine keine Verbindung zum Internet aufbauen kann, um ungewollte Updates oder ggf. spätere Zugriffe der Viren auf das Internet zu verhindern. Der Hauptspeicher der VM wird auf ein Gigabyte (GB) festgelegt. Heutzutage sind wesentlich größere Hauptspeicher die Norm. Mit ihrer Größe steigt jedoch auch die Zeit, welche für die Analyse mit Volatility benötigt wird. Um hier einen Ausgleich zu schaffen, wird eine Hauptspeicher-Größe festgelegt, bei der die VM noch einigermaßen flüssig zu bedienen ist und gleichzeitig die Zeit für die Analyse nicht zu lang ist. Cohen und Nissim nutzen ebenfalls diese RAM-Größe.

Im Anschluss werden die in Anhang B aufgeführten Konfigurationsschritte auf der VM durchgeführt. Hier wird u.a. der *Windows Defender* deaktiviert, damit die Ausführung der Trojaner nicht blockiert wird. Außerdem werden alle Einstellungen getroffen, welche ein Fernsteuern der VM mit der VirtualBox-API erlauben. Schließlich wird der *Windows Search* Dienst deaktiviert. Testläufe mit den Trojanern haben gezeigt, dass der Indexer dieses Dienstes sonst negative Auswirkungen auf die Performance der VM bei der Verschlüsselung durch den Trojaner hat.

Damit der Trojaner Dateien in der VM findet, die von ihm verschlüsselt werden können, wird die VM mit Beispieldaten befüllt. Aus dem Comon-Objects-in-Context-Datensatz (*COCO*) werden 200 Bilder (ca. 30 MB insgesamt) unter *C:\Users\masterthesis\Pictures* abgelegt. COCO ist ein umfangreicher Datensatz zur Erkennung, Segmentierung und Beschriftung von Objekten [47]. Er steht unter *Creative Commons Attribution 4.0 License* zur freien Benutzung. Zusätzlich werden zehn zufällige *dotx/dotm*-Dateien von der *Office-Templates-Website* unter *C:\Users\masterthesis\Documents* abgelegt [48]. Anschließend werden die VMs ausgeschaltet und ihre Netzwerkadapter deaktiviert.

Es wird ein Schnappschuss (Name: *base state*) der Maschine erstellt, welcher die Grundlage für alle weiteren in Abb. 3-2 dargestellten Schnappschüsse bildet.

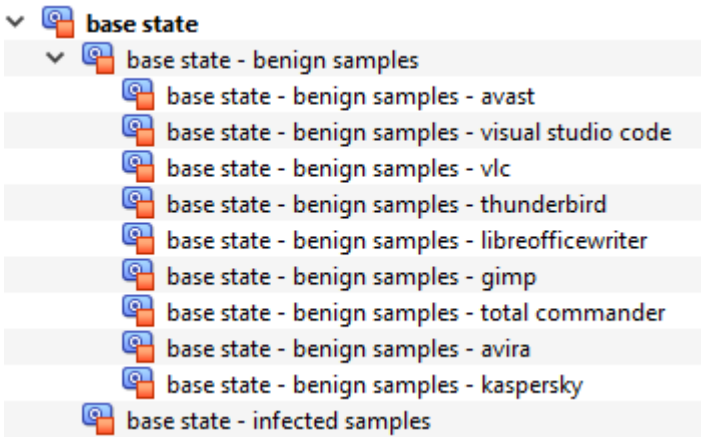


Abb. 3-2: Schema der Virtuellen Maschinen (Bildschirmfoto aus VirtualBox)

Für die gutartigen Programme (engl. *benign samples*) wird jeweils ein eigener Schnappschuss nach deren Installation angelegt. Die Verschlüsselungstrojaner (engl. *infected samples*) werden hingegen als ausführbare Dateien in einem Ordner auf dem Desktop der VM abgelegt, um sie von dort aus skriptgesteuert zu starten. Daher ist bei ihnen nur ein Schnappschuss notwendig.

3.1.2 Untersuchte Verschlüsselungstrojaner

Die Auswahl der Verschlüsselungstrojaner unterlag folgenden Anforderungen:

- **Der Trojaner ist lauffähig unter Windows 10 32bit.**
  Häufig ist Schadcode speziell für ein Betriebssystem programmiert und kann deswegen nicht auf jeder Plattform ausgeführt werden [13].
- **Der Trojaner ist als ausführbare Datei verfügbar.**
  Über diese Datei hinaus benötigte Programme zum Ausführen des Trojaners hätten die Experimente wesentlich verkompliziert.

- **Der Trojaner benötigt keine weiteren Nutzereingaben, nachdem er gestartet wurde.**
  Über den Start hinaus notwendige Nutzereingaben würden die Dokumentation der Verschlüsselung unterbrechen.
- **Der Trojaner benötigt keine Verbindung zum Internet.**
  Die heutigen hochentwickelten Verschlüsselungstrojaner sind unabhängiger und müssen vor dem Verschlüsseln von Dateien nicht mit ihrem C&C-Server kommunizieren (um einen für die Verschlüsselung zu nutzenden Schlüssel zu erhalten); eine Internetverbindung ist daher meist nicht erforderlich. Diese Anforderung ist wichtig, da auf das ständig wachsende Bedrohungspotential von Schadcode im Internet häufig mit *air-gaps* (deutsch *Luftlücken*) reagiert wird. Das heißt, besonders kritikale IT-Systeme werden vom Rest des Unternehmensnetzwerks und besonders vom Internet komplett isoliert. Doch auch diese Systeme können durch Wechseldatenträger infiziert werden [13].
- **Der Trojaner ist nicht zu alt.**
  Damit die Trojanerauswahl dem Aktualitätsanspruch dieser Arbeit gerecht wird, dürfen diese nicht zu alt sein. Die ersten Versionen der ältesten genutzten Trojaner (*Jigsaw* und *RedPetya*) sind aus dem Jahr 2016.
- **Der Trojaner muss einem determinierten Ausführungsablauf folgen.**
  Schadcode verfügt heutzutage über intelligente Methoden zur Ausführungsverschleierung und –verzögerung. So beginnen manche Verschlüsselungstrojaner ihre schändlichen Machenschaften erst nach mehrfachen Neustarts der infizierten Maschine und nach zufälligen Wartezeiten. Die Verwendung solches Schadcodes hätte das Erstellen der benötigen Hauptspeicher-Abbilder wesentlich erschwert (siehe Abschnitt 3.2). Sie werden daher ausgeschlossen.

Es werden Stichproben etlicher Trojaner beschafft und gegen die obigen Anforderungen getestet. Die Quellen der Trojaner sind *MalShare* [49] und *VirusShare* [50]. Dabei handelt es sich um Webseiten, die sich auf die Sammlung, Indexierung und das freie Angebot von Schadcode-Stichproben zu Forschungszwecken im Internet spezialisiert haben.

Viele Stichproben (u.a. von *Troldesh*, *UIWIX*, *Locky*, *Petwrap*, *Satan*, *Zusy*, *Ryuk*, *PewCrypt* und *samsam*) erfüllen die Anforderungen nicht und werden daher verworfen.

Tab. 3-2 listet die in dieser Arbeit untersuchten Verschlüsselungstrojaner auf und nennt die Hashes der verwendeten Dateien. Über den MD5-Hash können die Samples auf MalShare gefunden werden, über den SHA-256 auf VirusShare.

Tab. 3-2: In dieser Arbeit untersuchte Verschlüsselungstrojaner

Nr.	Verschlüsselungstrojaner	Hashes
1	**Dynamer**	**MD5:** 09b06fa3f1fadb21006c1fa79c75c536 **SHA-256:** 7eca26836fd75e64f83e810837dacc8b4c7b44ef1213e5872379a9b5c9a8a063
2	**GandCrab**	**MD5:** 26ea7a3076dc47bb078d05991087d75e **SHA-256:** 6f35196310894afed8b2ef6bdc8c9baa8802ec973f2f14eaee97bfe4be49b9d8
3	**Jigsaw**	**MD5:** a4bb3a5cb6835c089d769100d5461662 **SHA-256:** 52fb5d2e5555c38c4d0dd1bec893423761ed56ef1edcd3fddffd58cd507d7def
4	**Lockergoga**	**MD5:** e11502659f6b5c5bd9f78f534bc38fea **SHA-256:** c97d9bbc80b573bdeeda3812f4d00e5183493dd0d5805e2508728f65977dda15
5	**Occamy_C**	**MD5:** 76b640aa00354e46b29ca7ac2adfd732 **SHA-256:** 0b03bf1c7b596a862978999eebfa0703e6de48912c9a57e2fed3ae5cd747bea7
6	**RedPetya**	**MD5:** a92f13f3a1b3b39833d3cc336301b713 **SHA-256:** 4c1dc737915d76b7ce579abddaba74ead6fdb5b519a1ea45308b8c49b950655c
7	**Scarab**	**MD5:** 2b02a90ddedcfb709dcab08b454683a2 **SHA-256:** cf87bcf6dc5f1e199aa550caef92efa0a5c0476f20382719a8ea9b23cf489443
8	**Sodinokibi**	**MD5:** fc0383431f9867b043f6d58bd0f91242 **SHA-256:** 8ce95128cd1557dfb7d38883f9c6e94b8b74f56fdc162ecd67398111445ac7c6
9	**Wadhrama**	**MD5:** c2ec717719cc20e4527e5a4b43e48eb5 **SHA-256:** a1f3a173379286a85de310dc648c9040453571b130c63528f7856d1c3c8f0142
10	**Wannacry**	**MD5:** 84c82835a5d21bbcf75a61706d8ab549 **SHA-256:** ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa

Da sich die Namensgebung für Schadcode je nach Hersteller von Antiviren-Produkten unterscheidet, ist es schwierig, einen eindeutigen Namen festzulegen. Wenn Verschleierungsmaßnamen wie z. B. eigene *Packer* durch die Programmierer des Schadcodes eingesetzt werden, sind zudem häufig nur generische Namen für erkannte Viren und Trojaner vergeben. Die meisten Hersteller von Antiviren-Produkten versuchen jedoch den Namen der *Schadcode-Familie* auszugeben [25]. Dies sind Gruppen von Programmen, die auf ähnliche Art und Weise funktionieren.

Um eine eindeutige und richtige Benennung des in dieser Arbeit verwendeten Schadcodes zu ermöglichen, werden die Stichproben aus Tab. 3-2 mit *VirusTotal* [51] unterschiedlichen Antiviren-Produkten zur Überprüfung vorgelegt. Der Schadcode-Familienname, welcher von den meisten Produkten genannt wird, findet nun hier Anwendung.

3.1.3 Gutartige Programme in der Vergleichsgruppe

Damit der Klassifizierer lernen kann, welche Merkmale in welcher Höhe auf einen Trojaner hindeuten, ist es wichtig, ihm auch gutartige Programme als Vergleichsgruppe vorzulegen. Die hier vorgestellte Erkennungsmethode richtet sich vor allem an Virtuelle Maschinen von Organisationen, welche sie für Bürotätigkeiten einsetzen. Daher werden administrative, aber auch Sicherheitswerkzeuge ausgewählt, die in diesem Umfeld häufig anzutreffen sind. Die Programme werden von den Webseiten ihrer jeweiligen Hersteller heruntergeladen und auf der VM installiert. Anschließend wird jeweils ein Schnappschuss erstellt (siehe Abb. 3-2: Schema der Virtuellen Maschinen (Bildschirmfoto aus VirtualBox).

Folgende Tab. 3-3 listet die genutzten Programme, ihre Version und ihre Beschreibung auf. Der letzte Eintrag in dieser Tabelle bezieht sich auf den Grundzustand der VM. Dieser umfasst nur die nach den Anhängen A und B aufgesetzte Windows-Installation.

Tab. 3-3: In dieser Arbeit untersuchte gutartige Programme

Nummer	**Programmname** (Version)	**Beschreibung**
1	**Avast** (19.8.2393)	Antivirus-Programm [52]
2	**Avira** (15.0.2004.1825)	Antivirus-Programm [53]
3	**Kaspersky** (20.0.14.1085)	Antivirus-Programm [54]
4	**LibreOffice Writer** (6.3.4.2)	Textverarbeitungsprogramm [55]

Nummer	Programmname (Version)	Beschreibung
5	**Thunderbird** (68.3.1)	E-Mail-Client [56]
6	**VLC Player** (3.0.8)	Medien-Player [57]
7	**Visual Studio Code** (1.41.1)	Integrierte Entwicklungsumgebung [58]
8	**GIMP** (2.10.14)	Bildeditor [59]
9	**TotalCommander** (9.22a)	Dateimanager [60]
10	**Base State**	Grundzustand der VM

Der Anteil an Antivirus-Programmen wird so groß gewählt, da sie besonders tief ins Betriebssystem eingreifen und häufig Mechanismen nutzen, die in einem Hauptspeicher-Abbild Spuren ähnlich denen von Schadcode hinterlassen.

3.2 Datenerhebung

Für das Training der ausgewählten ML-Algorithmen, werden umfangreiche Datensätze benötigt. Die Erzeugung und Aufbereitung dieser Daten müssen in einer strukturierten und dokumentierten Weise erfolgen, um die Wiederholbarkeit der Experimente sicherzustellen. In dieser Arbeit wird unter Datenerhebung die Erzeugung der Hauptspeicher-Abbilder verstanden. Dieser Abschnitt erläutert das dazu angewandte Vorgehen im Detail.

Tab. 3-4: Ausführungszeit der Trojaner

Verschlüsselungstrojaner	Ausführungszeit (Laufzeit zwischen Abbilderstellung) in Sekunden
Dynamer	105 (1,05)
GandCrab	60 (0,6)
Jigsaw	52 (0,52)
Lockergoga	500 (5)
Occamy_C	23 (0,23)
RedPetya	13 (0,13)
Scarab	290 (2,9)
Sodinokibi	12 (0,12)
Wadhrama	3 (0,03)
Wannacry	30 (0,3)

Ausgangspunkt sind die nach dem in Abb. 3-2 dargestellten Schema konfigurierten Virtuelle Maschinen. Zu jeder Virtuellen Maschine werden 100 Hauptspeicher-Abbilder in festgelegten Intervallen über die Laufzeit des jeweils untersuchten Programms (entweder ein Verschlüsselungstrojaner oder ein gutartiges Programm) erstellt. Verschlüsselungstrojaner haben verschiedene Ausführungsphasen (siehe Abb. 2-1: Lebenszyklus Verschlüsselungstrojaner (nach [20])). Während jeder dieser Phasen ändert sich ihr Verhalten und damit die von ihnen im Hauptspeicher hinterlassenen Spuren. Damit eine Erkennung des Trojaners über seinen gesamten Lebenszyklus ermöglicht wird, müssen zu jeder Phase Abbilder erzeugt und später als einzelne Instanzen dem ML-Algorithmus vorgelegt werden.

Die Ausführungszeit der Verschlüsselungstrojaner wird in einem Testlauf gemessen. Sie beginnt mit dem Ausführen der infizierten Datei und endet, wenn der Trojaner alle Dateien in den Ordnern *Dokumente* und *Bilder* auf der VM verschlüsselt hat, er seine Lösegeldforderung anzeigt und die CPU-Auslastung seines Prozesses (überprüft im Task-Manager) auf unter 5% gesunken ist. Die CPU-Auslastung ist ein Hinweis darauf, welchen Umfang die Änderungen eines Programms am Hauptspeicher zu dem Zeitpunkt haben. Die jeweilige Ausführungszeit wird durch 100 geteilt und so die Länge der Intervalle zwischen der Erstellung der Hauptspeicher-Abbilder festgelegt (siehe Tab. 3-4: Spalte Ausführungszeit, Zahl in Klammern).

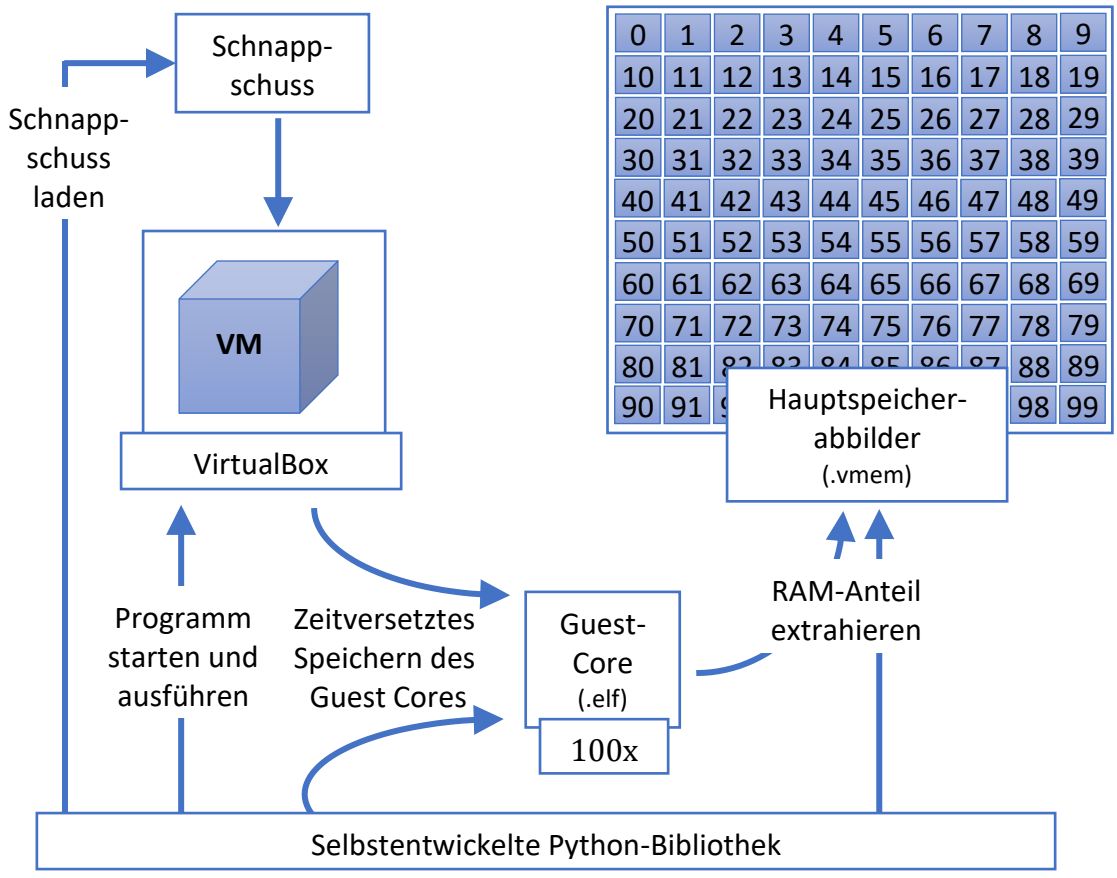


Abb. 3-3: Ablauf der Hauptspeicher-Abbild-Erstellung

Dieses Vorgehen unterscheidet sich von dem in [13]. Dort wird ein Abbild alle zehn Minuten erstellt. Dies erscheint aufgrund der teilweise kurzen Ausführungszeiten der Trojaner und der gutartigen Programme jedoch als nicht sinnvoll. Viele Trojaner durchlaufen alle ihre Ausführungsphasen in unter fünf Minuten.

Die Ausführungszeit aller gutartigen Programme wird pauschal auf 100 Sekunden festgelegt, da in Testläufen nach Ablauf dieser Zeit durch die Programme keine wesentliche CPU-Auslastung festgestellt werden kann. Es liegt also eine Sekunde Laufzeit der VM zwischen der Erstellung der einzelnen RAM-Abbilder. Die Zeitmessung beginnt in dem Moment, indem die Programme gestartet werden.

Möglichst viele Arbeitsschritte dieser Arbeit müssen automatisiert werden, da es sonst nicht möglich ist, genügend Daten innerhalb der zur Verfügung stehenden Zeit zu erzeugen. Daher wird das in Anhang C dokumentierte Modul in Python entwickelt. Dieses greift über ein *Application Programming Interface* (API) auf VirtualBox zu. Über dieses Modul wird ein bestimmter Schnappschuss geladen, die VM gestartet und Windows hochgefahren. Anschließend wird ein Abbild des Hauptspeichers der VM gezogen, bevor das untersuchte Programm ausgeführt wird (siehe Abschnitt 3.3.2 für die Gründe). Das Programm wird gestartet. Nach Ablauf der festgelegten Intervalle wird die VM *angehalten* (engl. *locked*). Alle in der VM laufenden Prozesse sind dann eingefroren. Es wird ein Guest-Dump erstellt. Aus diesem wird das Hauptspeicher-Abbild im *raw*-Format extrahiert und in einem Ordner abgelegt (vgl. Abb. 3-3). Währenddessen wird ein Bildschirmfoto der VM erstellt, welches den Ausführungsverlauf des untersuchten Programms dokumentiert. Danach wird die Maschine wieder gestartet und der Ablauf wiederholt sich.

Der manuelle Aufwand wird so auf ein Mindestmaß reduziert. Die Datenerhebung ist durch die Automatisierung darüber hinaus vollständig replizierbar. Insgesamt werden auf diese Weise 2020 Hauptspeicher-Abbilder erstellt.

3.3 Merkmalextraktion

Die Hauptspeicher-Abbilder können nicht als Ganzes von den ML-Algorithmen verarbeitet werden. Große Teile der Abbilder sind irrelevant für die Entscheidung, ob das IT-System infiziert wurde oder nicht. Zudem steigt die für das Training der Klassifizierer benötigte Zeit mit der Menge an Daten. Die Abbilder müssen daher ausgewertet werden. Ziel ist es, sie auf eine möglichst kleine Anzahl an Datenpunkten zu reduzieren, welche aber groß genug für eine valide Entscheidung über den Gesundheitszustand des Systems ist. Die Datenpunkte werden auch *Merkmale* genannt und bilden für jeweils eine Instanz zusammen die sogenannten *Merkmalsvektoren* (auch *Attri-*

butsvektor genannt). Dies ist ein Vektor dessen Skalare die Merkmale sind. Er beschreibt eine Instanz im Merkmalsraum, auf dem ein Algorithmus des Maschinellen Lernens trainiert und getestet wird.

In dieser Arbeit entspricht eine Instanz einem Hauptspeicher-Abbild. Der zur Extraktion der Merkmale und zum Aufbau der Merkmalsvektoren verwendete Ablauf wird in Abb. 3-4 dargestellt.

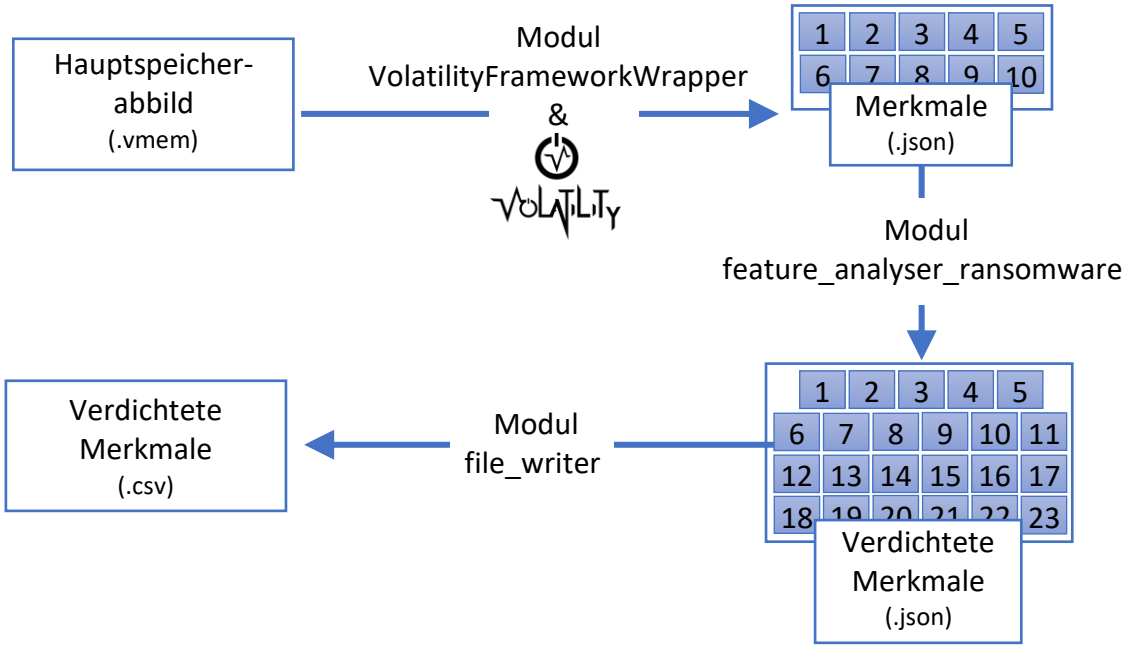


Abb. 3-4: Ablauf der Merkmalextraktion

3.3.1 Volatility

Der erste Schritt der Merkmalextraktion ist die Analyse jedes einzelnen Hauptspeicher-Abbilds mit dem Framework *Volatility*. Dieses ermöglicht über *Plugins*, bestimmte in der Arbeitsspeicher-Forensik häufig auftretende Fragestellungen automatisiert zu beantworten. Die für die Analyse benötigte Zeit hängt stark davon ab, ob das Betriebssystem 32 Bit- oder 64 Bit-basiert ist, wie groß der Arbeitsspeicher ist und welches Plugin eingesetzt wird.

Es werden zehn verschiedene Plugins eingesetzt. Ihre Ergebnisse werden im *JSON*-Format (*JavaScript Object Notation*, vgl. [61]) gespeichert. Dieses Format ermöglicht eine einfache automatisierte Weiterverarbeitung in den anschließenden Schritten der Merkmalsextraktion. Der Zwischenschritt über die JSON-Dateien wird gewählt, um die Analyse durch Volatility von der Extraktion der Merkmale zeitlich zu trennen. So können bei Fehlern in den Skripten die Extraktionen einfach wiederholt werden, ohne die wesentlich zeitaufwändigere Abbilderstellung und Volatility-Analysen wiederholen zu müssen.

Folgende Volatility-Plugins werden in dieser Arbeit genutzt (sie entsprechen den in [13] verwendeten):

- **callbacks**: Extrahiert eine Liste von *Kernel-Callbacks*. Dabei handelt es sich um Objekte, welche z. B. Treibern die Möglichkeit geben, Notifikationen anzufordern oder bereitzustellen, wenn bestimmte Bedingungen eintreten. Mit ihnen kann das Verhalten des Systems überwacht und auf Ereignisse durch eine Software reagiert werden [62].
- **dlllist**: Extrahiert eine Liste von geladenen *Dynamic-Link Libraries* (DLL) für jeden laufenden Prozess. Eine DLL ist ein Modul, das Funktionen und Daten enthält, die von einem anderen Modul (Anwendung oder DLLs) verwendet werden können [63]. Sie sind beliebtes Ziel von Schadcode, da eine infizierte DLL beim Aufruf durch ein privilegiertes Programm umfangreichen Zugriff auf das System gewähren kann [1].
- **handles**: Extrahiert eine Liste von offenen *Handles* für jeden laufenden Prozess. Eine Anwendung kann nicht direkt auf Objektdaten (z. B. eine Datei, ein Thread oder eine Grafik) oder die Systemressource, die ein Objekt repräsentiert, zugreifen. Stattdessen muss eine Anwendung ein *Objekt-Handle* erhalten, mit dem sie die Systemressource untersuchen oder ändern kann. Jedes Handle hat einen Eintrag in einer intern gepflegten Tabelle. Diese Einträge enthalten die Adressen der Ressourcen und die Mittel zur Identifizierung des Ressourcentyps [64]. In der Hauptspeicher-Forensik können über die Handles eines Prozesses diejenigen Systemressourcen identifiziert werden, auf welche aktuell Zugriff genommen wird[1].
- **ldrmodules**: Extrahiert eine Liste von *nicht geladenen* Modulen. Diese befinden sich nicht mehr in den üblichen Windows-internen Datenstrukturen, was auf den Versuch hindeuten kann, sie zu verstecken. Sie können jedoch über weitere Kernel-Datenstrukturen (*InInit*, *InMem* und *InLoad*) gefunden werden, die wesentlich schwieriger für Schadcode zu verändern sind [1].
- **modules**: Extrahiert eine Liste von *Treiber-Objekten*, die mit *Kernel-Modulen* assoziiert sind. Ein Modul ist eine ausführbare Datei oder DLL. Jeder *Prozess* besteht aus einem oder mehreren Modulen [65].
- **mutantscan**: Extrahiert eine Liste von *Mutex*-Objekten. Ein Mutex-Objekt ist ein Objekt zur Zugriffs-Synchronisation auf Ressourcen des Betriebssystems. Um zum Beispiel zu verhindern, dass zwei Threads gleichzeitig in den gemeinsamen Speicher schreiben, wartet jeder Thread auf den Besitz eines Mutex-Objekts, bevor er den Code ausführt, der auf den Speicher zugreift. Nach dem Schreiben in den gemeinsamen Speicher gibt der Thread das Mutex-Objekt frei [66]. Schadcode markiert Systeme häufig über System-weit erreichbare Mutexe, um eine Reinfizierung zu vermeiden [1].

- **privs**: Extrahiert die genutzten *Privilegien* aller Prozesse. Ein Privileg ist das Recht eines Kontos, z. B. eines Benutzer- oder Gruppenkontos, verschiedene systembezogene Operationen auf dem lokalen Computer durchzuführen, wie z. B. das Herunterfahren des Systems, das Laden von Gerätetreibern oder das Ändern der Systemzeit. Privilegien steuern den Zugriff auf Systemressourcen und systembezogene Aufgaben, während Zugriffsrechte den Zugriff auf sicherheitsrelevante Objekte steuern [67]. Über die Privilegien eines Prozesses kann auf seine Funktionalitäten und die Absichten seines Programmierers geschlossen werden [1].
- **psxview**: Extrahiert eine Liste von laufenden und auch versteckten *Prozessen*. Ein Programm besteht aus einem oder mehreren Prozessen. Ein Prozess ist im einfachsten Fall ein laufendes Programm [68]. Das Plugin nutzt dabei sechs verschiedene Ansätze zur Suche im Hauptspeicher, welche teilweise durch eigene Plugins abgebildet sind. Durch den Vergleich der Ergebnisse kann geschlussfolgert werden, dass Prozesse mutwillig versteckt werden, indem sie aus den Windows-üblichen Datenstrukturen zu ihrer Verwaltung entfernt werden [1].
- **svcscan**: Extrahiert eine Liste von *System-Diensten*. Ein solcher Dienst kann automatisch beim Systemstart, von einem Benutzer über das Systemsteuerungs-Applet "Dienste" oder von einer Anwendung, die die Dienstfunktionen verwendet, gestartet werden. Dienste können auch dann ausgeführt werden, wenn kein Benutzer am System angemeldet ist [69]. Auch dieses Plugin nutzt unterschiedliche Suchansätze, deren Vergleich mutwillig versteckte Dienste aufdecken kann. Dienste werden gerne von Schadcode-Entwicklern genutzt, um ihre Programme persistent auf einem System zu verankern [1].
- **thrdscan**: Extrahiert eine Liste von *Thread-Objekten*. Ein oder mehrere Threads laufen im Kontext eines Prozesses. Ein Thread ist die Basiseinheit, der das Betriebssystem Prozessorzeit zuweist. Ein Thread kann jeden Teil des Prozesscodes ausführen, einschließlich der Teile, die gerade von einem anderen Thread ausgeführt werden [68]. Dieses Plugin ist eine weitere Möglichkeit, versteckte Prozesse zu finden [1].

Um auch hier eine Automatisierung und vor allem eine Parallelisierung der Verarbeitung zu erreichen, wird der auf GitHub zur Verfügung stehende Volatility-Quellcode [34] direkt in die selbstentwickelte Python-Bibliothek eingebunden. Der Zugriff auf ihn wird über einen Wrapper (siehe Anhang E) abstrahiert. Mit diesen werden auch einige Fehler beseitigt, welche durch eine Ausführung vieler Volatility-Plugins auf einem geladenen Hauptspeicher-Abbild entstehen (siehe Anhang E: `get_threads()` und `get_timers()`). Die Analyse mit Volatility nimmt die längste Zeit bei der Verarbeitung der der Hauptspeicher-Abbilder ein. Das Extrahieren der Abbilder aus dem

RAM der VM sowie das Erstellen der Merkmalsvektoren und die Auswertung mit einer großen Anzahl von ML-Algorithmen fallen im Vergleich dazu kaum ins Gewicht. Deswegen wird in dieser Arbeit die Analyse mit Volatility als Maß für die Geschwindigkeit der Verarbeitung und damit für die Effizienz der genutzten Methode herangezogen.

Die Zeit, alle zehn Volatility-Plugins auf die 101 Hauptspeicher-Abbilder einer VM anzuwenden, wird mittels der `analyse_all_images_in_directory()`-Methoden gemessen (siehe Anhang F). Es werden vier unterschiedliche Ausführungsarten verglichen, die sich in der Anzahl der laufenden Analyseprozesse unterscheiden. Hiermit wird geprüft, ob es Engpässe bei der Abarbeitung gibt, die sich z. B. erst zeigen, wenn die CPU voll ausgelastet ist. Zum einen werden die Volatility-Plugins sequenziell auf die Abbilder angewandt. Das heißt, dass gleichzeitig nur ein Plugin auf ein bestimmtes Abbild angewandt werden kann. Zum anderen werden die Abbilder parallel analysiert. Hierbei laufen gleichzeitig zwei, vier oder acht Prozesse, welche Aufgaben (engl. *tasks*) abarbeiten. Eine Aufgabe entspricht der Analyse eines bestimmten Abbilds mit einem bestimmten Plugin. Da sogenannte *Task-Queues* (zu Deutsch *Aufgaben-Warteschlangen*) genutzt werden, ist es möglich, dass durch die Prozesse mehrere unterschiedliche Hauptspeicher-Abbilder gleichzeitig bearbeitet werden. Dies kommt vor, wenn die Prozesse die letzten Aufgaben zu einem Abbild abarbeiten. Ist nun ein Prozess vor den anderen fertig, holt er sich die nächste Aufgabe aus der Task-Queue. Da diese Aufgabe auf das nächste zu analysierende Abbild verweist, werden nun zwei unterschiedliche Abbilder gleichzeitig verarbeitet. Dies führt in der Theorie zu einer schnelleren Gesamtverarbeitungsgeschwindigkeit, während sich gleichzeitig die Laufzeiten der Plugins verlängern sollte.

Die Ergebnisse werden für die Abbilder einer VM sowie für alle Plugins, die nach einer Art der Ausführung abgearbeitet werden, gemittelt. So kann verglichen werden, wie sich die Laufzeiten für die Analyse eines Satzes an Hauptspeicher-Abbildern entwickeln, wenn diese sequenziell bzw. parallel durchgeführt wird.

3.3.2 Merkmale

Der zweite Schritt der Merkmalextraktion ist die Weiterverarbeitung, der durch die Volatility-Plugins produzierten JSON-Dateien zu Merkmalen. Erst diese eignen sich für das Training von Klassifizierern. Jedes Merkmal wird von einer eigenen in Python geschriebenen Methoden erzeugt. Diese Methoden lassen Wissen über typische Verhaltensmuster von Verschlüsselungstrojanern mit in die Extraktion einfließen. Die Informationsdichte und damit die Wahrscheinlichkeit für eine gute Performanz der Klassifizierer wird erhöht. Zum Beispiel gibt die Methode `get_number_of_false_rows_from_psxview()` zurück, wie viele Prozesse es in den Er-

gebnissen des psxview-Plugins gibt, welche von mindestens einem Prozess-Such-Ansatz nicht erkannt werden. Eine große Anzahl solcher Prozesse kann auf eine Infektion des Systems mit Schadcode hindeuten. Die Methoden werden im Python-Modul `feature_analyser_ransomware` gesammelt (siehe Anhang G). Ihre Namen orientieren sich an den durch [13] vergebene Merkmalsnamen. Da der Quellcode, welcher von Cohen und Nissim für die Erstellung ihrer Merkmalsvektoren genutzt wird, nicht bekannt ist, müssen die textuellen Beschreibungen der Autoren für den Entwurf einer eigenen Umsetzung genutzt werden. Insgesamt werden 23 Methoden zur Merkmalsextraktion geschrieben. Ihr Rückgabetyp ist immer ein *Integer*. Die folgende Tab. 3-5 gibt einen Überblick über die Methoden und jeweils eine kurze Beschreibung zu ihrer Funktionsweise sowie dazu, welche Volatility-Plugins jeweils genutzt werden.

Eine Besonderheit bildet die Methode `get_number_of_valid_exceptions_from_psxview()`. Sie vergleicht Prozesse, die während der Ausführung des untersuchten Programms laufen, mit solchen, die auf dem System vor dessen Ausführung liefen. Es liegt die Annahme zugrunde, dass alle vor der Ausführung laufenden Prozesse gutartig sind, auch wenn sie nicht durch einen der Prozess-Such-Ansätze von `psxview` gefunden werden. Es wird dazu ein Hauptspeicher-Abbild erstellt, kurz bevor der Trojaner oder das gutartige Programm auf der VM gestartet wird. Aus diesem werden die Namen aller laufenden Prozesse über das Volatility-Plugin `psxview` extrahiert. Das untersuchte Programm wird gestartet und die Namens-Extraktion für alle Abbilder wiederholt. Anschließend wird für alle Prozesse, die über mindestens einen der Prozess-Such-Ansätze des Volatility-Plugins `psxview` nicht gefunden werden, überprüft, ob diese in der Prozessnamensliste des ersten Abbilds erscheinen. Alle Prozesse, für die dies nicht der Fall ist, werden für das Ergebnis der Methode aufsummiert. Diese Methode ermöglicht also einen typischen Schritt der digitalen Forensik in der Bewertung von Prozessen zu automatisieren, nämlich den Vergleich von auffälligen unbekannten zu bekannten gutartigen Prozessen.

Dieser zweite Schritt der Merkmalsextraktion erfolgt ebenfalls in parallelisierter Form, um die Verarbeitung zu beschleunigen. Die Koordination der Abarbeitung übernimmt die Methode `get_all_fa_ransomware_outputs_in_parallel()`, welche zum Modul `memory_image_analysis` (siehe Anhang F) gehört. Die Ergebnisse werden über das Modul `file_writer` in einer CSV- (*comma seperated value*) und einer JSON-Datei abgelegt. Für jedes der 100 Abbilder, die während der Laufzeit des untersuchten Programms erstellt werden, entsteht ein Eintrag in der JSON-Datei und eine Spalte in der CSV-Datei. Über sie werden die Merkmalsvektoren der Instanzen (also der Hauptspeicher-Abbilder) dargestellt. Mit der Durchführung dieses Schritts sind alle Vorbereitungen getroffen, um die Klassifizierer des Maschinellen Lernens auf Basis dieser Daten zu trainieren.

Tab. 3-5: Genutzte Methoden zur Merkmalsextraktion

ID	Name der Extraktionsmethode	Beschreibung	Rückgabe-typ	Genutztes Volatility-Plugin
1	`get_number_of_callbacks()`	Gibt die Anzahl erkannter Callbacks zurück	Integer	callbacks
2	`get_number_of_exited_processes_from_psxview()`	Gibt die Anzahl erkannter beendeter Prozesse zurück	Integer	psxview
3	`get_number_of_false_columns_from_psxview()`	Gibt die Anzahl an Prozess-Such-Ansätzen zurück, die mindestens einen Prozess nicht entdeckt haben	Integer	psxview
4	`get_number_of_false_rows_from_psxview()`	Gibt die Anzahl an Prozessen zurück, die von mindestens einem Prozess-Such-Ansatz nicht erkannt werden	Integer	psxview
5	`get_number_of_handles()`	Gibt die Anzahl offener Handles zurück	Integer	handles
6	`get_number_of_ldrmodules()`	Gibt die Anzahl geladener Modulen zurück	Integer	ldrmodules
7	`get_number_of_modules()`	Gibt die Anzahl geladener Treiber-Objekte mit Verbindung zu Kernel-Modulen zurück	Integer	modules
8	`get_number_of_mutexes_from_mutantscan()`	Gibt die Anzahl genutzter Mutexe zurück	Integer	mutantscan
9	`get_number_of_not_default_enabled_privs()`	Gibt die Anzahl an Privilegien zurück, welche nicht das Attribut *Default* besitzen, aber aktiv sind	Integer	privs
10	`get_number_of_not_ininit_dll_paths_from_ldrmodules()`	Gibt die Anzahl an Modulen zurück, welche nicht in der InInit-Liste sind	Integer	ldrmodules
11	`get_number_of_not_ininitloadmem_dll_paths_from_ldrmodules()`	Gibt die Anzahl an Modulen zurück, welche in keiner der drei Listen vorkommen.	Integer	ldrmodules
12	`get_number_of_not_inload_dll_paths_from_ldrmodules()`	Gibt die Anzahl an Modulen zurück, welche nicht in der InLoad-Liste sind	Integer	ldrmodules
13	`get_number_of_not_inmem_dll_paths_from_ldrmodules()`	Gibt die Anzahl an Modulen zurück, welche nicht in der InMem-Liste sind	Integer	ldrmodules

ID	Name der Extraktionsmethode	Beschreibung	Rückgabe-typ	Genutztes Volatility-Plugin
14	`get_number_of_processes_from_psxview()`	Gibt die Anzahl erkannter Prozesse zurück	Integer	psxview
15	`get_number_of_pslist_processes_from_psxview()`	Gibt die Anzahl an Prozessen zurück, welche durch das *pslist*-Plugin entdeckt werden	Integer	psxview
16	`get_number_of_psscan_processes_from_psxview()`	Gibt die Anzahl an Prozessen zurück, welche durch das *psscan*-Plugin entdeckt werden	Integer	psxview
17	`get_number_of_running_services_from_svcscan()`	Gibt die Anzahl laufender Dienste zurück	Integer	svcscan
18	`get_number_of_services_from_svcscan()`	Gibt die Anzahl erkannter Dienste zurück (laufend und gestoppt)	Integer	svcscan
19	`get_number_of_stopped_services_from_svcscan()`	Gibt die Anzahl gestoppter Dienste zurück	Integer	svcscan
20	`get_number_of_threads_from_thrdscan()`	Gibt die Anzahl erkannter Threads zurück	Integer	thrdscan
21	`get_number_of_unique_dll_paths_from_dlllist()`	Gibt die Anzahl einzigartiger von Prozessen genutzter DLLs zurück, welche das *dlllist*-Plugin erkannt hat	Integer	dlllist
22	`get_number_of_unique_dll_paths_from_ldrmodules()`	Gibt die Anzahl einzigartiger von Prozessen genutzten DLLs zurück, welche das *ldrmodules*-Plugin erkannt hat	Integer	ldrmodules
23	`get_number_of_valid_exceptions_from_psxview()`	Gibt die Anzahl an Prozessen zurück, welche von mindestens einem Prozess-Such-Ansatz nicht entdeckt werden, aber bereits vor Ausführung des untersuchten Programms vorhanden waren	Integer	psxview

3.4 Algorithmen des Maschinellen Lernens

Auf Basis der erstellten Merkmalsvektoren werden Klassifizierer unterschiedlicher Art trainiert. Es werden die in Abschnitt 2.6.3 vorgestellten Algorithmen des Maschinellen Lernens angewandt. Die Leistung der Klassifizierer wird anschließend über die in Abschnitt 2.6.4 ausgewählten Leistungsmaße verglichen.

ML-Algorithmen lassen sich auf unterschiedliche Arten implementieren. Für bestimmte Algorithmen existieren sogar völlig unterschiedliche Umsetzungen, die nur auf einem gemeinsamen Grundprinzip basieren und daher einen gemeinsamen oder ähnlichen Namen besitzen (hierzu gehört bspw. der *NaiveBayes*-Algorithmus). Alle in dieser Arbeit verwendeten ML-Algorithmen entsprechen den Implementierungen in *Weka Version 3.8.4*.

3.4.1 Weka

Weka bietet Implementierungen von Lernalgorithmen, die sich leicht auf den erstellten Datensatz anwenden lassen. Es ist eine bewährte Open-Source-Software für Maschinelles Lernen, welches über eine grafische Benutzeroberfläche, Standard-Terminalanwendungen oder eine Java-API gesteuert werden kann. Für Lehre, Forschung und industrielle Anwendungen ist es weit verbreitet und enthält eine Fülle von eingebauten Werkzeugen für Standardaufgaben des Maschinellen Lernens. Weka ermöglicht es, einen Datensatz vorzuverarbeiten, ihn in ein Lernschema einzuspeisen und den daraus resultierenden Klassifizierer und seine Leistung zu analysieren, ohne Programmcode schreiben zu müssen [37].

In dieser Arbeit wird Weka als Rahmenwerk für das Maschinelle Lernen ausgewählt, um eine möglichst gute Vergleichbarkeit zur Arbeit von Cohen und Nissim zu gewährleisten. Die beiden Autoren wenden ebenfalls Weka an. So können Unterschiede in den Implementierungen der ML-Algorithmen, aber auch in der Auswertung der noch vorzustellen Experimente größtenteils ausgeschlossen werden.

Ein Großteil der angewendeten ML-Algorithmen wird von Wekas Standard-Installation bereits mitgeliefert. Zwei Klassifizierer werden über den eingebauten *Package Manager* aus dem offiziellen Weka-Repository nachgeladen. Sie haben folgende Versionsnummern:

- **LibSVM**: Version 1.0.10
- **LibLINEAR**: Version 1.9.8

Weka definiert ein eigenes Format für die Trainings- und Testdatensätze. Eine *ARFF*-Datei (*Attribute-Relation File Format*) enthält Instanzen und Merkmalsvektoren in textbasierter Form. Die Merkmalsnamen befinden sich im Kopf der Datei (mit dem

Bezeichner @*attribute* versehen) und weisen den Typ des Merkmals aus (z. B. *numeric*, *nominal* oder *class*). Die Merkmalsvektoren der Instanzen folgen zeilenweise. Die einzelnen Merkmale werden durch Kommas getrennt. Da ARFF eine einfachere Weiterverarbeitung der Datensätze in Weka ermöglicht, werden die in CSV vorliegenden Datensätze in dieses Format übersetzt. Dazu wird der von Weka mitgelieferte *ArffViewer* genutzt. Auch alle für die Experimente speziell zusammengestellten Trainings- und Testdatensätze werden in ARFF gespeichert (siehe Module `file_writer`, Anhang D).

3.4.2 Experimentkonfigurationen in Weka

Für die Durchführung der im Folgenden beschriebenen Experimente wird zum einen das sogenannte *KnowledgeFlow Environment* in Weka genutzt. Dieses bietet die Möglichkeit, alle notwendigen Schritte von der Vorverarbeitung der Datensätze über das Training der Klassifizierer bis hin zur Leistungsmessung und Ergebnisdokumentation auf grafischem Wege zusammenzustellen.

Anhang H zeigt den genutzten *Flow* (zu Deutsch *Ablauf*) für die Experimente, welche mit geschichteter zehnfacher Kreuzvalidierung durchgeführt werden (Experimente 1.1 bis 1.3). Durch den *ArffLoader* wird der Trainings- und Testdatensatz aus einer Datei geladen. Der *ClassAssigner* weist Weka auf die Datenfelder hin, in denen die Klasse der Instanzen aufgeführt ist. Anschließend werden die Schichtungen erzeugt und getrennt nach Trainings- und Testdaten auf die ML-Algorithmen verteilt. Mit diesen wird zuerst ein Klassifizierer trainiert, der anschließend mit den Testdaten evaluiert wird (siehe *ClassPerformanceEvaluator*). Die Ergebnisse werden von den *TextSavern* in Dateien abgelegt. Zusätzlich wird die ROC jedes Klassifizierers mittels *ModelPerformanceChart* berechnet und mit den *ImageSavern* in einer PNG-Datei (*Portable Network Graphics*) abgelegt.

Zum anderen wird Wekas *Experimenter* genutzt. Über ihn kann eine große Anzahl an Datensätzen einfacher verarbeitet werden als über den *KnowledgeFlow*. Er wird für alle Experimente eingesetzt, die nicht über eine Schichtung validiert werden, sondern über separate Trainings- und Testdatensätze (Experimente 2 und 3). Abb. 3-5 zeigt die Nutzerschnittstelle des *Experimenters* mit geladener Konfiguration für die funktionsbasierten Klassifizier von Experiment 3. Der Ausgabepfad für die Ergebnisse des Experiments wird im oberen Teil der Nutzerschnittstelle konfiguriert. Im unteren Teil werden links alle Datensätze geladen, welche für das Training und die anschließenden Tests notwendig sind. Im rechten unteren Teil werden die Klassifizierer geladen und konfiguriert.

Es werden kombinierte ARFF-Dateien verwendet, welche sowohl die Trainings- (die ersten 90% der Instanzen, also 1800 Stück) als auch Testdaten (die letzten 10 % der

Instanzen, also 200 Stück) enthalten. Die Aufteilung der Instanzen auf Trainings- und Testdatensatz kann dem entsprechenden Experiment in Abschnitt 3.5 entnommen werden. Über die Auswahl *Experiment Type* im mittleren Teil des Bildschirmfotos wird festgelegt, die Instanzen während der Durchführung zu trennen. Dabei wird keine Randomisierung durchgeführt, sondern die festgelegte Reihenfolge der Instanzen erhalten (engl. *order preserved*). Die genutzten ARFF-Dateien werden mittels der Methoden `make_experiment3_combined_trainingtestsets()` und `make_experiment3_splits()` im Modul `file_writer` (siehe Anhang D) erstellt.

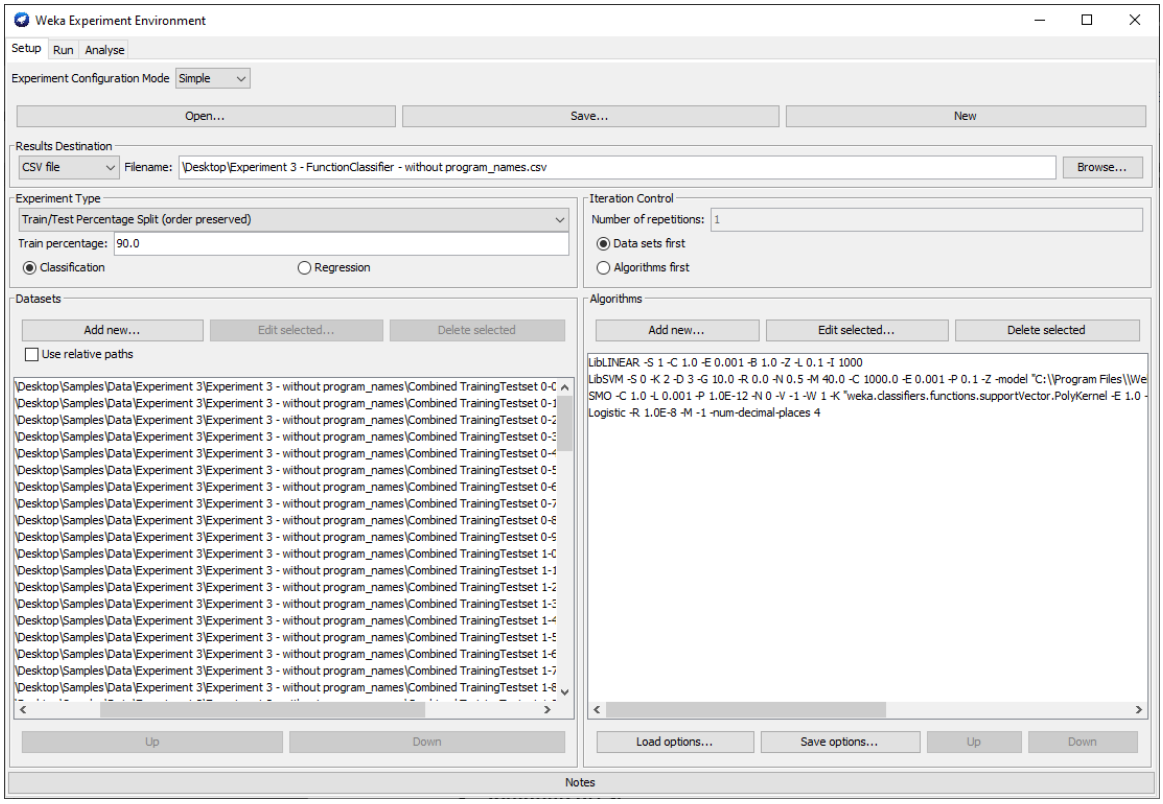


Abb. 3-5: Der Experimenter mit geladener Konfiguration (Bildschirmfoto aus Weka)

Für jede Klassifizierer-Familie werden die Experimente mit dieser Konfiguration wiederholt, um einzelne Ergebnis-Dateien zu erhalten. Diese werden im Anschluss mit *Microsoft Excel* weiterverarbeitet, um einen Ergebnisdurchschnitt über alle Durchführungen mit unterschiedlichen Datensätzen zu erhalten.

Alle Klassifizierer werden grundsätzlich mit den Weka-Standardeinstellungen trainiert. Die folgenden Konfigurationen werden genutzt:

- **RandomForest**:
  - `numIterations = 100` (Anzahl der Bäume im RandomForest)
  - `maxDepth = 0`

- **Bagging**:
  - `Classifier = REPTree`
- **LogitBoost**, **AdaBoostM1**:
  - `Classifier = DecisionStump`
- **Sequential Minimal Optimization**:
  - `Kernel = PolyKernel`
- **J48**:
  - `Unpruned = FALSE`
  - `confidenceFactor = 0.25`
- **LibSVM**:
  - `Normalization = TRUE`
  - `Cost = 1000` (mit GridSearch in Weka auf Basis der Datensätze von Experiment 1.1 ermittelt)
  - `Gamma = 10` (mit GridSearch in Weka auf Basis der Datensätze von Experiment 1.1 ermittelt)
  - `kernelType = radial basis function`
- **LibLINEAR:**
  - `Normalization = TRUE`

3.5 Experimentdurchführung

Um die Fähigkeiten der Klassifizierer möglichst umfassend zu erforschen, werden mehrere unterschiedliche Experimentdesigns entworfen. Jedes Experiment beantwortet dabei spezielle Fragen, die sich an den Forschungsfragen von Cohen und Nissim orientieren. So wird z. B. untersucht, wie die Leistung der Klassifizierer bei Infektion der Virtuellen Maschine mit bekannten, aber auch mit unbekannten Trojanern ausfällt.

Im Folgenden werden die Experimente detailliert vorgestellt. Sie entsprechen den in [13] angewandten. Die Komplexität steigt dabei von Experiment 1 zu Experiment 3 an. Dies wird insbesondere an der Zusammenstellung der Trainings- und Testdatensätze deutlich.

3.5.1 Experiment 1 - Erkennung von Anomalien und spezifischen Zuständen

Mit den folgenden drei untergeordneten Aufgaben wird die Fähigkeit der Klassifizierer untersucht, verschiedene bekannte Zustände des Servers zu erkennen. Ein Zustand wird durch die auf der VM laufenden Prozesse beschrieben. Alle Experimente werden mit 10-facher Kreuz-Validierung und daher nach dem in Anhang H dargestellten Ablauf durchgeführt.

3.5.1.1 Experiment 1.1 – Erkennung von Anomalien

Dieses Experiment untersucht die Leistungsfähigkeit der Klassifizierer bei der Erkennung von anomalen Zuständen der Virtuellen Maschine. Die Klassifizierer werden mit allen zur Verfügung stehenden Instanzen trainiert. Nur die Instanzen des Baseline-Schnappschusses werden der Klasse **normal** zugeordnet. Alle anderen Instanzen (sowohl die der Verschlüsselungstrojaner-Schnappschüsse als auch die der gutartigen Programme) erhalten die Klasse **abnormal**.

3.5.1.2 Experiment 1.2 – Unterscheidung zwischen infizierten und unauffälligen Zuständen

Dieses Experiment untersucht die Leistungsfähigkeit der Klassifizierer bei der Unterscheidung von infizierten und unauffälligen Zuständen der Virtuellen Maschine. Die Klassifizierer werden mit allen zur Verfügung stehenden Instanzen trainiert. Die Instanzen, welche mit den Schnappschüssen der gutartigen Programme (siehe Tab. 3-3) erstellt werden, werden der Klasse **benign** zugeordnet. Die anderen Instanzen (alle mit den Verschlüsselungstrojaner-Schnappschüssen erstellten, siehe Tab. 3-2) erhalten die Klasse **infected**.

3.5.1.3 Experiment 1.3 – Erkennung von spezifischen Zuständen

Dieses Experiment untersucht die Leistungsfähigkeit der Klassifizierer bei der Erkennung von spezifischen bekannten Zuständen der Virtuellen Maschine. In diesem Experiment geht es nicht darum, eine bestimmte Klasse zu erkennen, sondern die Leistungsfähigkeit der Klassifizierer bezogen auf die Erkennung jedes einzelnen vorhanden Programms zu bestimmen. Die Klassifizierer werden mit allen zur Verfügung stehenden Instanzen trainiert. Die Instanzen werden ihren Namen entsprechenden Klassen zugeordnet (bspw. erhalten Instanzen, die von der mit dem *Jigsaw*-Trojaner infizierten VM erstellt wurden, die Klasse **Jigsaw**).

3.5.2 Experiment 2 – Erkennung von unbekannten infizierten Zuständen

Dieses Experiment untersucht die Leistungsfähigkeit der Klassifizierer bei der Erkennung von unbekannten infizierten Zuständen der Virtuellen Maschine. Das Experiment wird für jeden der zehn in dieser Arbeit untersuchten Verschlüsselungstrojaner einmal durchgeführt. Die Ergebnisse aller zehn Durchführungen werden gemittelt. Es wird Wekas *Experimenter* genutzt und darin ein Experiment für jede Klassifizierer-Familie erstellt.

Tab. 3-6: Konfigurationen für die Durchführung von Experiment 2

Nummer der Durch-führung	Trainingsdatensatz		Testdatensatz	
	Benign	**Infected**	**Benign**	**Infected**
1	90 % der Instanzen von Avast, Avira, Kaspersky, LibreOffice Writer, Thunderbird, VLC Player, Visual Studio Code, GIMP, Total Commander, Base State	GandCrab, Jigsaw, Lockergoga, Occamy_C, RedPetya, Scarab, Sodinokibi, Wadhrama, Wannacry	10 % der Instanzen von Avast, Avira, Kaspersky, LibreOffice Writer, Thunderbird, VLC Player, Visual Studio Code, GIMP, Total Commander, Base State	Dynamer
2		Dynamer, Jigsaw, Lockergoga, Occamy_C, RedPetya, Scarab, Sodinokibi, Wadhrama, Wannacry		GandCrab
3		Dynamer, GandCrab, Lockergoga, Occamy_C, RedPetya, Scarab, Sodinokibi, Wadhrama, Wannacry		Jigsaw
4		Dynamer, GandCrab, Jigsaw, Occamy_C, RedPetya, Scarab, Sodinokibi, Wadhrama, Wannacry		Lockergoga
5		Dynamer, GandCrab, Jigsaw, Lockergoga, RedPetya, Scarab, Sodinokibi, Wadhrama, Wannacry		Occamy_C
6		Dynamer, GandCrab, Jigsaw, Lockergoga, Occamy_C, Scarab, Sodinokibi, Wadhrama, Wannacry		RedPetya
7		Dynamer, GandCrab, Jigsaw, Lockergoga, Occamy_C, RedPetya, Sodinokibi, Wadhrama, Wannacry		Scarab
8		Dynamer, GandCrab, Jigsaw, Lockergoga, Occamy_C, RedPetya, Scarab, Wadhrama, Wannacry		Sodinokibi
9		Dynamer, GandCrab, Jigsaw, Lockergoga, Occamy_C, RedPetya, Scarab, Sodinokibi, Wannacry		Wadhrama
10		Dynamer, GandCrab, Jigsaw, Lockergoga, Occamy_C, RedPetya, Scarab, Sodinokibi, Wadhrama,		Wannacry

Für jede Durchführung werden die Klassifizierer mit 90% der zur Verfügung stehenden Instanzen der gutartigen Programme und den Instanzen von neun der zehn untersuchten Verschlüsselungstrojaner trainiert. Die restlichen 10% der gutartigen Instanzen sowie die Instanzen des vorher ausgelassenen Verschlüsselungstrojaners bilden den Testdatensatz zu einer Durchführung. Die Instanzen, welche auf Basis der Schnappschüsse der gutartigen Programme (siehe Tab. 3-3) erstellt werden, werden der Klasse **benign** zugeordnet. Die anderen Instanzen (alle auf Basis der Verschlüsselungstrojaner-Schnappschüsse erstellten, siehe Tab. 3-2) erhalten die Klasse **infected**.

Die Instanzen der gutartigen Programme werden im Verhältnis 9:1 zwischen dem Trainings- und dem Testdatensatz aufgeteilt, damit diese eine einheitliche Größe haben und im Testdatensatz neben den Instanzen des Verschlüsselungstrojaners weitere gutartige Instanzen vorhanden sind. Jeder Trainingsdatensatz besteht aus 900 Instanzen gutartiger Programme (zufällige 90% der zur Verfügung stehenden 1000 Instanzen). Hinzu kommen je 100 Instanzen von 9 der 10 Verschlüsselungstrojaner. Der Testdatensatz umfasst 200 Instanzen. Dies sind die restlichen 10% der gutartigen Instanzen sowie die 100 Instanzen eines spezifischen Verschlüsselungstrojaners.

Tab. 3-6 listet die Konfigurationen für die einzelnen Durchführungen auf. Wichtig ist, dass bei diesem Experiment der Verschlüsselungstrojaner im Testdatensatz für die Klassifizierer unbekannt ist.

3.5.3 Experiment 3 – Erkennung von unbekannten Zuständen (infiziert und unauffällig)

Dieses Experiment untersucht die Leistungsfähigkeit der Klassifizierer bei der Erkennung von unbekannten infizierten Zuständen der Virtuellen Maschine, wenn gleichzeitig auch unbekannte gutartige Zustände vorhanden sind. Ziel ist es, die Falsch-Positiv-Rate der Klassifizierer, bezogen auf unbekannte gutartige Programme, genauer einschätzen zu können. Es werden alle Kombinationen von jeweils einem gutartigen Programm und einem Verschlüsselungstrojaner ausgewertet. Daher wird das Experiment insgesamt 100 Mal unter Anwendung unterschiedlicher Trainings- und Testdatensätze durchgeführt. Die Ergebnisse aller Durchführungen werden gemittelt. Es wird Wekas *Experimenter* genutzt und darin ein Experiment für jede Klassifizierer-Familie erstellt.

Tab. 3-7: Beispiel-Konfigurationen für die Durchführung von Experiment 3

Nummer der Durch-führung	Trainingsdatensatz		Testdatensatz	
	Benign	**Infected**	**Benign**	**Infected**
1	Avast, Avira, Kaspersky, LibreOffice Writer, Thunderbird, VLC Player, Visual Studio Code, Total Commander, Base State	GandCrab, Jigsaw, Lockergoga, Occamy_C, RedPetya, Scarab, Sodinokibi, Wadhrama, Wannacry	GIMP	Dynamer
2	Avast, Avira, LibreOffice Writer, Thunderbird, VLC Player, Visual Studio Code, GIMP, Total Commander, Base State	GandCrab, Jigsaw, Lockergoga, Occamy_C, RedPetya, Scarab, Sodinokibi, Wadhrama, Wannacry	Kaspersky	Dynamer
3	Avast, Avira, Kaspersky, Thunderbird, VLC Player, Visual Studio Code, GIMP, Total Commander, Base State	GandCrab, Jigsaw, Lockergoga, Occamy_C, RedPetya, Scarab, Sodinokibi, Wadhrama, Wannacry	Libre-Office Writer	Dynamer
...				
11	Avast, Avira, Kaspersky, LibreOffice Writer, Thunderbird, VLC Player, Visual Studio Code, Total Commander, Base State	Dynamer, Jigsaw, Lockergoga, Occamy_C, RedPetya, Scarab, Sodinokibi, Wadhrama, Wannacry	GIMP	GandCrab
12	Avast, Avira, LibreOffice Writer, Thunderbird, VLC Player, Visual Studio Code, GIMP, Total Commander, Base State	Dynamer, Jigsaw, Lockergoga, Occamy_C, RedPetya, Scarab, Sodinokibi, Wadhrama, Wannacry	Kaspersky	GandCrab
13	Avast, Avira, Kaspersky, Thunderbird, VLC Player, Visual Studio Code, GIMP, Total Commander, Base State	Dynamer, Jigsaw, Lockergoga, Occamy_C, RedPetya, Scarab, Sodinokibi, Wadhrama, Wannacry	Libre-Office Writer	GandCrab
...				
98	Avast, Kaspersky, LibreOffice Writer, Thunderbird, VLC Player, Visual Studio Code, GIMP, Total Commander, Base State	Dynamer, GandCrab, Jigsaw, Lockergoga, Occamy_C, RedPetya, Scarab, Sodinokibi, Wadhrama	Avira	Wannacry
99	Avast, Avira, Kaspersky, LibreOffice Writer, Thunderbird, VLC Player, GIMP, Total Commander, Base State	Dynamer, GandCrab, Jigsaw, Lockergoga, Occamy_C, RedPetya, Scarab, Sodinokibi, Wadhrama	Visual Studio Code	Wannacry
100	Avast, Avira, Kaspersky, LibreOffice Writer, Thunderbird, VLC Player, Visual Studio Code, GIMP, Total Commander	Dynamer, GandCrab, Jigsaw, Lockergoga, Occamy_C, RedPetya, Scarab, Sodinokibi, Wadhrama	Base State	Wannacry

Für jede Durchführung wird ein Trainingsdatensatz aus den Instanzen von neun der zehn gutartigen Programme und den Instanzen von neun der zehn Verschlüsselungstrojaner erstellt (insgesamt 1800 Instanzen). Der zugehörige Testdatensatz enthält alle Instanzen des übrigen gutartigen Programms und des Trojaners (insgesamt 200 Instanzen).

Tab. 3-7 stellt die Konfiguration von einigen Trainings- und Testdatensätzen der Durchführungen auszugsweise dar. Die Instanzen, welche auf Basis der Schnappschüsse der gutartigen Programme (siehe Tab. 3-3) erstellt werden, werden der Klasse **benign** zugeordnet. Die anderen Instanzen (alle auf Basis der Verschlüsselungstrojaner-Schnappschüsse erstellten, siehe Tab. 3-2) erhalten die Klasse **infected.**

3.6 Leistungskennzahlen für die Evaluation

Die Experimente werden mit den in Abschnitt 2.6.4 vorgestellten Maßen für die Leistungsfähigkeit von Klassifizierern evaluiert, um so eine Vergleichbarkeit herzustellen. Dies sind die *Richtig-Positiv-Rate, die Falsch-Positiv-Rate, die Größe der Fläche unter der Grenzwertoptimierungskurve* (engl. *area under receiver operating curve*) sowie das *F-Maß*. Diese Leistungsmaße verwenden Cohen und Nissim ebenfalls [13]. Die Leistungsmaße beziehen sich immer auf eine bestimmte Klasse der untersuchten Daten. Beispielweise muss die Richtig-Positiv-Rate für jede vergebene Klasse gesondert berechnet werden, da sie ein Wert für die Leistung eines Klassifizierers beim korrekten Zuordnen von Instanzen zu dieser Klasse ist. Alle Instanzen, die vom Klassifizierer zu dieser Klasse gezählt werden, aber in Wahrheit nicht dazugehören, erhöhen die Falsch-Positiv-Rate. Im Folgenden werden deswegen die jeweils betrachteten Klassen für jedes Experiment gesondert benannt.

4 Ergebnisse

In den folgenden Abschnitten werden die Ergebnisse der Experimente dargestellt. Die Klassifizierer werden nach der Fläche unter der ROC-Kurve geordnet (vom höchsten zum niedrigsten). Zur besseren Übersicht sind die Zellen der Tabellen farbkodiert nach der Höhe ihres eingetragenen Wertes (grün-gelb-rot). Die Farbe Grün steht für hohe Werte von RPR, AUC und F-Maß, aber für kleine Werte der FPR. Für die Farbe Rot verhält es sich andersherum.

Da die Leistungsmaße sich immer auf eine bestimmte Klasse des untersuchten Datensatzes beziehen, wird für jedes Experiment im Folgenden angegeben, welches diese Klasse ist. Für die Interpretation der Ergebnisse ist wichtig, dass die Klassenzuordnungen der Instanzen ebenfalls über die Experimente hinweg wechseln. So kann eine Instanz, die z. B. in Experiment 1.1 zur Klasse *abnormal* gehört, in Experiment 2 zur Klasse *infected* gehören. In Kapitel 3 wird diese Zuordnung im Detail erläutert.

4.1 Experiment 1.1

Die Werte in der Tab. 4-1 entsprechen den Erkennungsergebnissen für die Klasse **abnormal**.

Tab. 4-1: Ergebnisse von Experiment 1.1 - Erkennung von Anomalien

Klassifizierer	RPR	FPR	AUC	F-Maß
BayesNet	1	0	1	1
Logistic	1	0	1	1
SMO	1	0	1	1
LibSVM	1	0	1	1
LibLINEAR	1	0	1	1
LogitBoost	1	0	1	1
RandomForest	1	0	1	1
Bagging	1	0,01	1	1
AdaBoostM1	1	0,02	1	0,999
NaiveBayes	0,98	0	0,999	0,99
J48	1	0,03	0,981	0,999

Es ist zu erkennen, dass die Klassifizierer sehr gute Ergebnisse liefern (hohe RPR, AUC und F-Maß und kleine FPR). Mit Ausnahme der funktionsbasierten Klassifizierer gibt es jedoch in jeder Familie von ML-Algorithmen mindestens einen Vertreter, dessen Ergebnisse nicht perfekt sind (*Bagging*, *AdaBoostM1*, *NaiveBayes* und *J48*).

4.2 Experiment 1.2

Die Werte in der Tab. 4-2 entsprechen den Erkennungsergebnissen für die Klasse **infected**.

Tab. 4-2: Ergebnisse von Experiment 1.2 - Unterscheidung von infizierten und unauffälligen Zuständen

Klassifizierer	RPR	FPR	AUC	F-Maß
Logistic	1	0	1	1
LibSVM	1	0	1	1
LogitBoost	1	0	1	1
RandomForest	1	0	1	1
Bagging	1	0	1	1
AdaBoostM1	1	0	1	1
J48	0,999	0	1	0,999
BayesNet	0,99	0,033	0,999	0,979
LibLINEAR	1	0,02	0,99	0,99
SMO	1	0,058	0,972	0,971
NaiveBayes	0,968	0,13	0,964	0,923

Auch hier ist zu erkennen, dass die meisten Klassifizierer sehr gute Ergebnisse liefern. Die linearen *Support Vector Machines* (*LibLINEAR* und *SMO*) können jedoch ihre hervorragende Erkennungsleistung aus Experiment 1.1 nicht wiederholen. Insgesamt gibt es einen Anstieg von falschen Klassifikationen (siehe im Vergleich zu Experiment 1.1 gestiegene FPR).

4.3 Experiment 1.3

Die Werte in der Tab. 4-3 entsprechen dem ***gewichteten Mittel* aller Klassen**. Die Ergebnisse werden dazu nach der Anzahl vorhandener Instanzen in einer betrachteten Klasse gewichtet und anschließend über alle Klassen gemittelt.

AdaBoostM1 kann nur bei der Erkennung der Klassen *Avast*, *Dynamer* und *Lockergoga* Instanzen richtig zuordnen. Allen anderen 17 Klassen wird keine einzige Instanz zugeordnet. Aus diesem Grund werden die Ergebnisse für diesen Klassifizierer verworfen, da die gemittelten Werte der Leistungsmaße bei diesem geringen Erkennungsgrad keine Aussagekraft haben.

Insgesamt ist die Erkennungsleistung über alle eingesetzten Klassifizierer gesunken. Sie kann jedoch immer noch als gut eingeschätzt werden, da sich die Verschlechterungen häufig auf die dritte Nachkommastelle der Leistungsmaße beziehen.

Tab. 4-3: Ergebnisse von Experiment 1.3 - Erkennung von spezifischen Zuständen

Klassifizierer	RPR	FPR	AUC	F-Maß
Logistic	1	0	1	0,999
LogitBoost	0,999	0	1	0,999
RandomForest	1	0	1	1
Bagging	0,996	0	1	0,996
J48	0,998	0	1	0,997
BayesNet	0,99	0,001	1	0,99
LibSVM	0,999	0	0,999	0,999
NaiveBayes	0,947	0,003	0,998	0,946
SMO	0,946	0,003	0,996	0,944
LibLINEAR	0,963	0,002	0,981	0,962
AdaBoostM1	?	?	?	?

4.4 Experiment 2

Für die Ergebnisse in Tab. 4-4 werden die ***gewichteten Mittel* der betrachteten Klassen** einer Durchführung wiederum über alle zehn Durchführungen gemittelt.

Tab. 4-4: Ergebnisse von Experiment 2 - Erkennung von unbekannten infizierten Zuständen

Klassifizierer	RPR	FPR	AUC	F-Maß	K. Erg.
RandomForest	0,972	0,073	1	0,91	0
Logistic	0,966	0,33	0,992	0,965	0
BayesNet	0,908	0,015	0,987	0,892	0
LogitBoost	0,944	0,056	0,975	0,938	0
Bagging	0,975	0,024	0,975	0,973	0
AdaBoostM1	0,975	0,024	0,975	0,973	0
J48	0,975	0,024	0,975	0,973	0
LibLINEAR	0,917	0,082	0,917	0,903	0
SMO	0,894	0,105	0,894	0,88	0
NaiveBayes	0,799	0,201	0,871	0,775	0
LibSVM	0,869	0,13	0,869	0,894	1

Die Spalte *K. Erg.* (Keine Ergebnisse) gibt an, bei wie vielen der insgesamt 10 Durchführungen der Klassifizierer nicht in der Lage ist, einer Klasse überhaupt Instanzen richtig zuzuordnen. Die richtige Zuordnung von wenigstens einer Instanz ist Voraussetzung für die Berechnung des F-Maßes einer Klasse. Um das durchschnittliche gewichtete F-Maß aller Durchführungen zu bestimmen, muss das F-Maß jeder betrach-

teten Klasse in jeder einzelnen Durchführung bekannt sein. Somit sind die F-Maß-Ergebnisse aller Klassifizierer, die in Tab. 4-4 Eintragungen in der Spalte *K. Erg.* vorweisen, mit Vorsicht zu bewerten. Die Werte aller anderen Spalten sind vollständig vorhanden und werden korrekt berechnet.

Die Erkennungsleistung der Klassifizierer sinkt im Vergleich zu Experiment 1. Es fällt auf, dass die FPR wesentlich höher ist als in den bisherigen Experimenten. Das beste Erkennungsergebnis liefert der *RandomForest*-Klassifizierer.

Es wird versucht die schlechte Leistung des *LibSVM*-Klassifizierers durch erneutes Anwenden von *GridSearch* auf den Datensätzen dieses Experimentes zu verbessern. Die Ergebnisse der Parameter-Suche erbringen `cost = 1000` und `gamma = 1`. Die Leistung des Klassifizierer verbesserte sich damit auf: *RPR = 0,914, FPR = 0,086, AUC = 0,914* und *F-Maß = 0,897*. Dieses Ergebnis ist mit dem von *LibLINEAR* zu vergleichen.

4.5 Experiment 3

Für die Ergebnisse in Tab. 4-5 werden die ***gewichteten Mittel*** **der beiden betrachteten Klassen** einer Durchführung wiederum über alle 100 Durchführungen gemittelt.

Tab. 4-5: Ergebnisse von Experiment 3 - Erkennung von unbekannten infizierten und unauffälligen Zuständen

Klassifizierer	RPR	FPR	AUC	F-Maß	K. Erg.
RandomForest	0,883	0,117	0,981	0,908	9
AdaBoostM1	0,944	0,0559	0,969	0,939	0
LogitBoost	0,929	0,07	0,965	0,924	0
BayesNet	0,861	0,138	0,943	0,912	14
J48	0,937	0,06	0,937	0,932	0
Bagging	0,932	0,067	0,936	0,923	0
Logistic	0,846	0,153	0,917	0,835	1
LibLINEAR	0,881	0,118	0,8815	0,89	6
SMO	0,879	0,12	0,879	0,885	6
NaiveBayes	0,788	0,211	0,871	0,824	15
LibSVM	0,655	0,344	0,655	0,651	21

Die Spalte *K. Erg.* (Keine Ergebnisse) gibt an, bei wie vielen der insgesamt 100 Durchführungen der Klassifizierer nicht in der Lage ist, einer Klasse überhaupt Instanzen richtig zuzuordnen. Dies ist Voraussetzung für die Berechnung des F-Maß dieser Klasse. Um das durchschnittliche gewichtete F-Maß aller Durchführungen zu bestimmen, muss das F-Maß jeder betrachteten Klasse in jeder einzelnen Durchführung be-

kannt sein. Somit sind die F-Maß-Ergebnisse aller Klassifizierer, die in Tab. 4-5 Eintragungen in der Spalte *K. Erg.* vorweisen, mit Vorsicht zu bewerten. Die Werte aller anderen Spalten sind vollständig vorhanden und werden korrekt berechnet.

Die Erkennungsleistung der Klassifizierer sinkt im Vergleich zu Experiment 2. Es fällt auf, dass die FPR nochmals höher ist als in den bisherigen Experimenten. Der *RandomForest*-Klassifizierer liefert die beste AUC. Er hat jedoch im Vergleich mit *AdaBoostM1* eine kleinere RPR und höhere FPR.

Es wird versucht die schlechte Leistung des *LibSVM*-Klassifizierers durch erneutes Anwenden von *GridSearch* auf Basis der Datensätze dieses Experimentes zu verbessern. Die Ergebnisse der Parameter-Suche entsprechen jedoch der bereits genutzten *LibSVM*-Konfiguration. Somit ist davon auszugehen, dass der Klassifizierer bei diesem Experiment grundsätzlich schlechtere Leistung erbringt als die anderen, unabhängig von der Parameterwahl von `cost` und `gamma`.

4.6 Gemessene Ausführungszeiten der Analyse

Die Ausführungszeiten für die Volatility-Plugins auf den Hauptspeicher-Abbildern einer VM sind unabhängig von den Experimenten, da sie nur einmal am Anfang durchgeführt werden. Danach werden die Trainings- und Testdatensätze aus den erzeugten Merkmalvektoren abhängig vom Experiment zusammengesetzt. Somit genügt die Analyse eines Satzes an Abbildern, um Aussagen über die benötigte Zeit für die Analyse treffen zu können. Dabei ist natürlich zu beachten, dass diese Zeit stark von der genutzten Hardware abhängt.

Es werden vier unterschiedliche Ausführungsarten verglichen. Hiermit wird geprüft, ob es Engpässe bei der Abarbeitung gibt, die sich z. B. erst zeigen, wenn die CPU voll ausgelastet ist. Zum einen werden die Volatility-Plugins sequenziell auf die Abbilder angewandt. Zum anderen werden die Abbilder durch zwei, vier oder acht Prozesse parallel analysiert.

Abb. 4-1 zeigt, wie sich die Laufzeit der genutzten Volatility-Plugins bei den unterschiedlichen Ausführungsarten verändert. Wie erwartet steigt die Laufzeit für ein einzelnes Plugin an, je mehr gleichzeitig ausgeführt werden. Dies kann an der Lesegeschwindigkeit der Festplatte, aber auch an weiteren von der Hardware abhängigen Faktoren liegen.

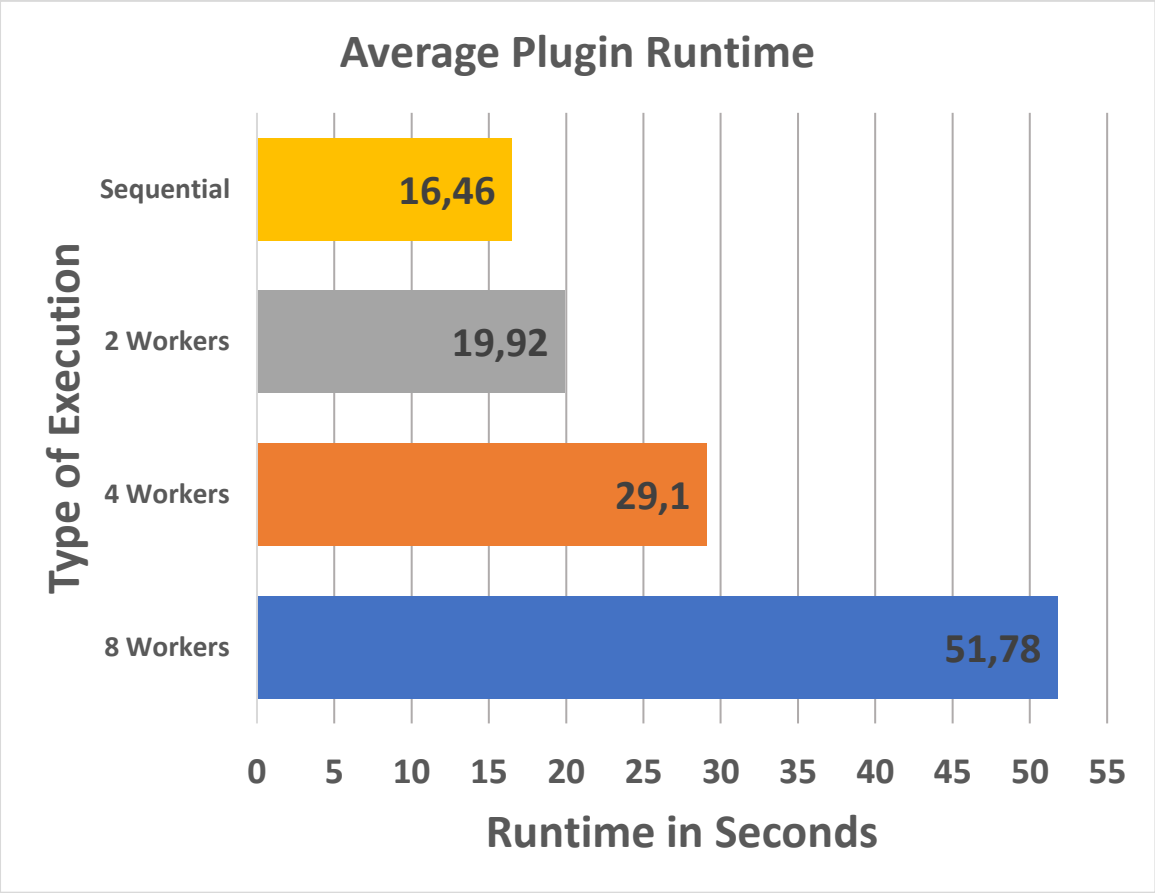


Abb. 4-1: Durchschnittliche Laufzeit eines Volatility-Plugins nach Ausführungsart

Mit acht gleichzeitigen Prozessen (auch *worker* genannt) liegt die Laufzeit für ein Plugin durchschnittlich bei ca. 50 Sekunden, während sie bei Ausführung nur eines Plugins gleichzeitig bei nur ca. 17 Sekunden liegt.

Abb. 4-2 zeigt die Gesamtdauer der Analyse von 101 Hauptspeicher-Abbildern für unterschiedliche Ausführungsarten. Auch hier bestätigt sich die theoretische Vermutung, dass die parallele Ausführung mehrerer Plugins gleichzeitig die Verarbeitungsgeschwindigkeit insgesamt verbessert. Mit acht gleichzeitigen Prozessen kann die benötigte Zeit auf ca. drei Stunden gesenkt werden. Beim sequenziellen Abarbeiten der Plugins muss noch ca. fünf Stunden länger auf die Ergebnisse gewartet werden.

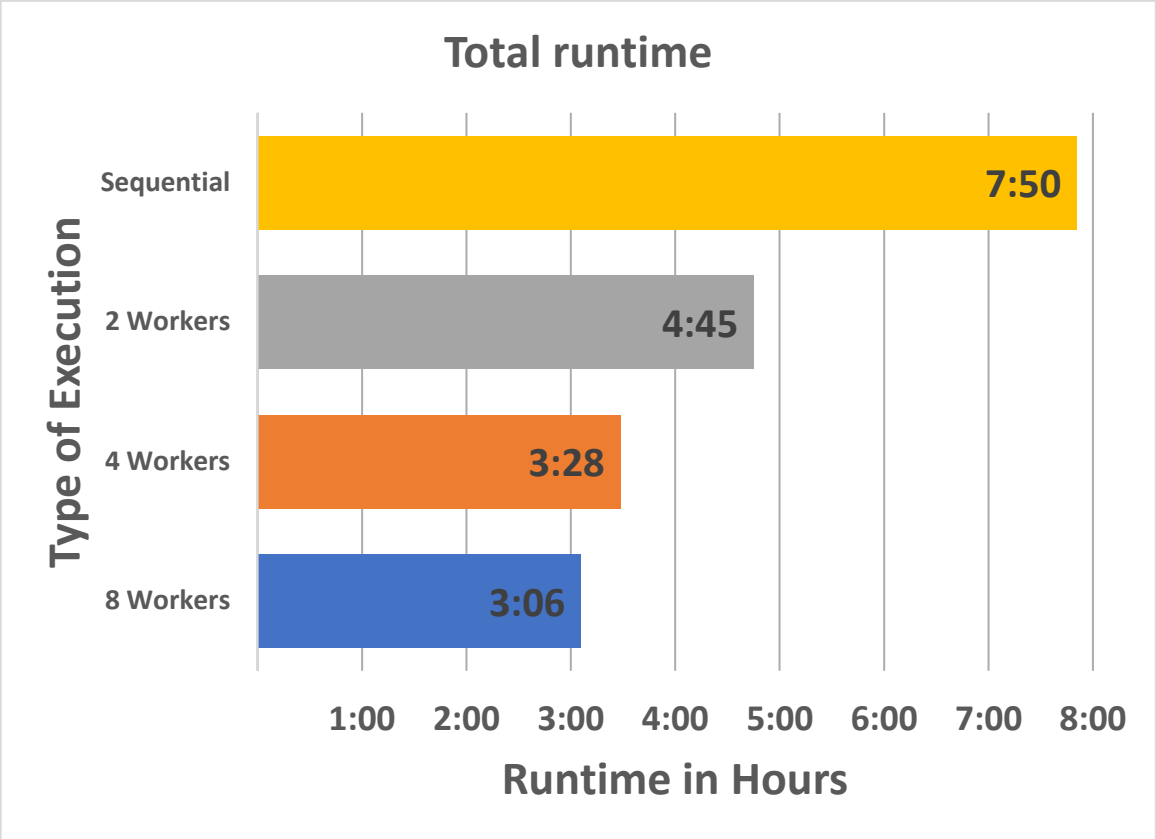


Abb. 4-2: Analysedauer für 101 Hauptspeicher-Abbilder einer VM nach Ausführungsart

Die Analyse eines einzelnen Abbilds dauert also mit acht Workern ca. 1,86 Minuten. Sequenziell liegt diese Zeit mit ca. 4,7 Minuten noch um das Zweieinhalbfache höher.

Die Verlängerung der Laufzeit eines einzelnen Plugins hat somit keine negativen Auswirkungen auf die Dauer der Analyse insgesamt, wenn die Zahl der parallelen Prozesse erhöht wird. Weitere Optimierungen können die benötigte Zeit sicherlich noch verkürzen.

5 Diskussion

In diesem Abschnitt werden die erzielten Ergebnisse mit denen der replizierten Arbeit von Cohen und Nissim verglichen. Für eventuelle Abweichungen werden Hypothesen aufgestellt. Anschließend wird die Anwendbarkeit der in dieser Arbeit verwendeten Methode in der Praxis besprochen. Eine mögliche technische Umsetzung wird hier genauso erläutert wie die Interpretation der Klassifizierungsergebnisse durch eine Software-Lösung. Abschließend werden die Grenzen aufgezeigt, welche entweder durch zukünftige Arbeiten adressiert und ggf. aufgehoben werden können oder der Methode inhärent sind und damit nur temporär umgangen werden können.

5.1 Experiment 1

Die Ergebnisse der Experimente 1.1 – 1.3 decken sich zum großen Teil mit denen von Cohen und Nissim. Die Klassifizierer erreichen sehr hohe Detektionsraten über alle drei untergeordneten Aufgaben des Experiments hinweg. Besonders hervorzuheben ist *RandomForest*. Dieser Klassifizierer liefert durchgehend eine perfekte Leistung (*RPR = 1, FPR = 0, AUC = 1* und *F-Maß = 1*). Dies deckt sich ebenfalls mit den Ergebnissen der replizierten Arbeit.

Insgesamt fällt jedoch auf, dass die restlichen Klassifizierer etwas schlechter arbeiten. Dies kann an dem im Vergleich zu Cohen und Nissim doppelt so großen Datensatz liegen. Dieser stellt gerade bei Experiment 1.3 wesentlich höhere Anforderungen an die Klassifizierer, was sich auch in den Ergebnissen dieses Experiments widerspiegelt. Mit einer größeren Anzahl an untersuchten Programmen wächst die Wahrscheinlichkeit dafür, dass die Merkmalsvektoren der Instanzen sich ähneln. Dies führt zwangsläufig zu einer gewissen Unschärfe bei der Klassifikation und damit zu einer schlechteren Detektionsrate.

Antiviren-Programme erfüllen ein ähnliches Aufgabenspektrum und haben daher auch ähnliche Funktionsweisen. Daher liegt die Vermutung nahe, dass die drei untersuchten Antivirus-Programme (*Avast*, *Avira* und *Kaspersky*) für die Fehlklassifizierungen sorgen. Durch Überprüfung der Wahrheitsmatrix für den am schlechtesten abschneidenden Klassifizierer *LibLINEAR* kann dies jedoch ausgeschlossen werden. Vielmehr hat der Klassifizierer bei den Trojanern *Dynamer, Jigsaw* und *Wannacry* die meisten Fehler gemacht. Interessanterweise werden die fehlklassifizierten Instanzen jedoch wiederum anderen Trojanern zugeordnet (acht *Wannacry*-Instanzen werden *Dynamer* zugeordnet). Dies ist ein Indiz für die Validität der Methode, da der ML-Algorithmus über die gewählten Merkmale effektiv zwischen Trojanern und gutartigen Programmen unterscheiden kann.

Das Verhalten des *AdaBoostM1*-Klassifzierer stellt eine Besonderheit dar. Dieses Modell verteilt alle vorhandenen Instanzen nur auf die Klassen *Avast, Dynamer* und *Lockergoga*. Dergleichen kann in [13] nicht beobachtet werden. Eine Veränderung der Parameter des Klassifizierers kann keine Verbesserung bei der Detektion erbringen. Erst der Austausch des geboosteten Algorithmus *DecisionStump* gegen *J48* führt zu einer perfekten Erkennungsrate (*RPR = 1, FPR = 0, AUC = 1* und *F-Maß = 1*). Es liegt die Vermutung nahe, dass der *DecisionStump*-Algorithmus nicht mit der großen Anzahl an Klassen dieses Experiments umgehen kann. Die Ergebnisse von Cohen und Nissim widersprechen dieser These jedoch. Hier sind weitere Untersuchungen notwendig, um den Grund für diese Abweichung zu eruieren.

In den drei Experimenten 1.1 – 1.3 wird die Leistungsfähigkeit der Klassifizierer bezüglich der Erkennung von bekannten Zuständen gemessen. Die Klassifizierer bewältigen diese Aufgabe überwiegend mit sehr guten Ergebnissen. Sind also Merkmalsvektoren von Verschlüsselungstrojanern oder gutartigen Programmen bekannt, ist die Wahrscheinlichkeit sehr hoch, dass diese auch als solche erkannt werden.

5.2 Experiment 2

Auch die Ergebnisse von Experiment 2 decken sich gut mit denen der replizierten Arbeit. Die Werte von RPR und AUC fallen insgesamt etwas besser aus. Dieser positive Trend wird jedoch durch die etwas schlechteren Ergebnisse der FPR geschmälert. Gerade die FPR ist ein wichtiger Indikator für die Güte eines Programms zur Erkennung von Schadcode. Eine hohe Anzahl von Fehlalarmen strapaziert die Nerven der Nutzer und kann schnell zum Ignorieren der Alarme und Deinstallation des Programms führen. Hier muss also besonderes Augenmerk auf der Konzeption von Lösungen für den Endanwender liegen.

Wiederum erbringt die Anwendung des *RandomForest*-Algorithmus die besten Ergebnisse (*RPR = 0,972, FPR = 0,073, AUC = 1* und *F-Maß = 0,91*). Die für die Höhe der FPR ausschlaggebenden Fehlklassifizierungen treten in zwei der zehn Durchführungen des Experimentes auf. Es werden beim Datensatz „*Combined TrainingTestset 1 – GrandCrab“* 97 Instanzen der Klasse „*infected*“ fälschlicherweise als „*benign*“ klassifiziert. Beim Datensatz *„Combined TrainingTestset 4 - Occamy_C“* werden 49 Instanzen von infizierten VMs nicht als solche erkannt. Auch der zweite Entscheidungsbaumbasierte ML-Algorithmus *J48* zeigt den gleichen Fehler im Umgang mit *Occamy_C* wie *RandomForest*. Betrachtet man die anderen Familien von Klassifizieren, zeigt sich, dass deren Fehlklassifikationen über alle 10 Durchführungen verteilt liegen. Somit scheinen *RandomForest* und *J48* explizit Probleme mit der Erkennung von Instanzen der mit *Occamy_C* und *GandCrab* infizierten VMs zu haben.

Die insgesamt etwas höhere FPR dieses Experiments können ggf. an der Funktionsweise der betrachteten Trojaner liegen. Die entscheidenden Abläufe, welche zu großen Veränderungen in den hier betrachteten Merkmalen der Hauptspeicher-Abbilder führen, werden häufig von Leerlaufphasen unterbrochen. In diesen Phasen ruht die Ausführung des Trojaners, was eine Ähnlichkeit der Merkmale zu denen der gutartigen Programme zur Folge haben könnte. Dies würde zwangsweise zu schlechteren Klassifikationsergebnissen und insbesondere einer höheren FPR führen. Hier sind weitergehende Prüfungen notwendig, um diese These zu überprüfen. Insbesondere sollte die Methode an VMs im tatsächlichen Büroeinsatz geprüft werden.

Es wird versucht die schlechte Leistung des *LibSVM*-Klassifizierers durch ein erneutes Anwenden von *GridSearch* zu optimieren. Die Parameter-Suche auf Basis des in diesem Experiment verwendeten Datensatzes *„Combined TrainingTestset 3 – Lockergoga“* erbringt ein um neun Punkte niedrigeres `gamma`. Dieser Datensatz wird ausgewählt, da der Klassifizierer hier besonders schlechte Einzelergebnisse erbringt. Eine Wiederholung des Experimentes mit `gamma = 1` verbessert die Ergebnisse erheblich. *LibSVM* schneidet nun nur unwesentlich schlechter als *LibLINEAR* ab.

In diesem Experiment wird die Leistungsfähigkeit der Klassifizierer bezüglich der Erkennung von unbekannten infizierten unter einer Reihe von bekannten unauffälligen Zuständen gemessen. Die Klassifizierer bewältigen diese Aufgabe überwiegend mit sehr guten Ergebnissen. Sind also Merkmalsvektoren von Verschlüsselungstrojanern unbekannt, ist die Wahrscheinlichkeit trotzdem sehr hoch, dass diese – bei Vorhandensein von bekannten unauffälligen Zuständen – als solche erkannt werden.

5.3 Experiment 3

Die Ergebnisse von Experiment 3 decken sich erneut gut mit denen der replizierten Arbeit. Die Werte von RPR, AUC und F-Maß fallen insgesamt etwas besser aus. Die FPR-Werte sind ebenfalls etwas besser (niedriger). Dieses schwierigste der durchgeführten Experimente scheint vom größeren Datensatz zu profitieren.

Wiederum erbringt die Anwendung des *RandomForest*-Algorithmus die besten Ergebnisse (*RPR = 0,883, FPR = 0,117, AUC = 0,981* und *F-Maß = 0,908*). Es fällt jedoch auf, dass einige Klassifizierer im Umgang mit bestimmten Datensätzen Probleme haben. *RandomForest* gehört zu dieser Gruppe. Er kann in neun von einhundert Durchführungen einer der beiden betrachteten Klassen (*infected* oder *benign*) keine einzige Instanz zuordnen. Der Klassifizierer kann in den Merkmalsvektoren der Instanzen also keine effektiven Unterscheidungskriterien für die Klassen finden. In einem solchen Fall gibt es für das F-Maß dieser Klasse keinen Wert. Abb. 5-1 stellt dies für die

Durchführung 44 mit dem Datensatz „*Combined TrainingTestset 4-4 - Occamy_C, TotalCommander*“ dar. Die Abbildung ist ein Bildschirmfoto des *Classify*-Reiters im *Weka Explorer*. In diesem wird das Experiment wiederholt, um eine detaillierte Übersicht über die Ergebnisse der Durchführung 44 zu erhalten. In der Tabelle „*Detailed Accuracy by Class“* wird in Spalte „*F-Measure“* und der Zeile „*Benign“* nur ein Fragezeichen ausgewiesen. Das F-Maß jeder Klasse wird jedoch benötigt, um das gewichtete durchschnittliche F-Maß aller Durchführungen zu berechnen. Somit sind die F-Maß-Ergebnisse aller Klassifizierer, die in der Ergebnis-Tabelle Eintragungen in der Spalte *K. Erg.* vorweisen, mit Vorsicht zu bewerten. Die Werte aller anderen Spalten sind vollständig vorhanden und werden korrekt berechnet. Es ist nicht bekannt, ob dieses Problem ebenfalls in den Ergebnissen von Cohen und Nissim auftritt, da dort keine Angaben darüber gemacht werden.

```
RandomForest

Bagging with 100 iterations and base learner

weka.classifiers.trees.RandomTree -K 0 -M 1.0 -V 0.001 -S 1 -do-not-check-capabilities

Time taken to build model: 0.16 seconds

=== Evaluation on test split ===

Time taken to test model on test split: 0 seconds

=== Summary ===

Total Number of Instances              200

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  F-Measure  ROC Area  Class
                 1,000    1,000    0,667      0,739     Infected
                 0,000    0,000    ?          0,739     Benign
Weighted Avg.    0,500    0,500    ?          0,739

=== Confusion Matrix ===

   a   b   <-- classified as
 100   0 |   a = Infected
 100   0 |   b = Benign
```

Abb. 5-1: Ergebnisse der Durchführung 44 von Experiment 3 mit RandomForest (Bildschirmfoto aus Weka)

Zusammengefasst bringt der RandomForest-Klassifizierer also in neun von einhundert Durchführungen des Experimentes 3 besonders schlechte Leistung. Er gleicht

dies jedoch mit den restlichen Durchführungen aus und kommt so auf ähnliche Ergebnisse wie in der replizierten Arbeit. Im Rahmen dieser Arbeit kann keine genauere Untersuchung der Gründe für dieses Verhalten erfolgen.

Es wird versucht die schlechte Leistung des *LibSVM*-Klassifizierers durch erneutes Anwenden von *GridSearch* auf Basis eines Datensatzes („*Combined TrainingTestset 1-4 - GrandCrab, TotalCommander*“) dieses Experimentes zu verbessern. Die Ergebnisse der Parameter-Suche entsprechen jedoch der bereits genutzten *LibSVM*-Konfiguration. Somit ist davon auszugehen, dass der Klassifizierer bei diesem Experiment grundsätzlich schlechtere Leistung erbringt als die anderen, unabhängig von der Wahl der Parameter `cost` und `gamma`.

In diesem Experiment wird die Leistungsfähigkeit der Klassifizierer bezüglich der Erkennung von unbekannten infizierten Zuständen bei gleichzeitigem Vorhandensein von unbekannten unauffälligen Zuständen gemessen. Die Klassifizierer bewältigen diese schwerste aller gestellten Aufgaben überwiegend mit sehr guten Ergebnissen. Sind also Merkmalsvektoren von Verschlüsselungstrojanern unbekannt, ist die Wahrscheinlichkeit trotzdem sehr hoch, dass diese – trotz gleichzeitigem Vorhandensein von unbekannten unauffälligen Zuständen – als solche erkannt werden.

5.4 Forschungsfragen

Für die Beantwortung der Forschungsfragen dieser Arbeit müssen die Experimente wiederholt werden, welche Cohen und Nissim in ihrer hier replizierten Arbeit vorschlagen. Dies geschieht unter Anwendung einiger Änderungen am Material (z. B. Auswahl der Verschlüsselungstrojaner) und an der Methode (z. B. Einsatz von *VirtualBox* und *Windows 10 Pro*). Sind die Experimente erfolgreich, ist es möglich, Cohen und Nissims Forschungsfragen 1-4 zu beantworten. Dies würde bedeuten, dass ihre vorgeschlagene Methode auch unter den gemachten Veränderungen erfolgreich angewandt werden kann. Im Folgenden wird daher für jede ihrer Forschungsfragen überprüft, ob dies der Fall ist. Anschließend werden daraus Schlüsse auf die vom Autor dieser Arbeit aufgestellten Forschungsfragen gezogen.

Die Auswertung eines Hauptspeicher-Abbilds von der Erstellung bis zum Speichern der Merkmalsvektoren dauert auf der genutzten Hardware durchschnittlich 1,86 Minuten. Das Auswerten eines Merkmalsvektors mit den Klassifizierern beansprucht anschließend nur wenige Sekunden. Dies sind Werte, die bei paralleler Ausführung der Volatility-Plugins und der Merkmalsextraktionsmethoden erreicht werden. Durch weitere Optimierungen (wie z. B. spezialisierter Hardware, neuer Volatility-Version usw.) lässt sich dieser Wert sicherlich noch verbessern. Dieses Ergebnis ist wesentlich besser als jenes in [13] mit 16 Minuten Analysedauer für ein Abbild von 1

GB Größe. Da die Autoren nicht genau beschreiben, wie sie Volatility benutzen (Anbindung an den Quellcode direkt oder Nutzung der kompilierten Version, Ausführung der Plugins parallel oder sequenziell), können die Gründe hierfür nicht genannt werden.

Die Auswertungsgeschwindigkeit und die erbrachten Ergebnisse zeigen, dass Virtuelle Maschinen auf Basis von VirtualBox und dem Gastbetriebssystem *Windows 10 Pro* anhand des verwendeten Merkmalssatzes effizient auf Befall durch Verschlüsselungstrojaner überwacht werden können (*Forschungsfrage CuN-F1*). Die Leistung der Klassifizierer bei Unterscheidung zwischen gutartigen Programmen und Verschlüsselungstrojaner liegt weit über einer Zufallsentscheidung. In vielen Fällen wird sogar eine hundert prozentige Detektionsrate erreicht. Die Effektivität der Klassifizierer, welche auf Basis des verwendeten Merkmalssatzes trainiert werden, ist bei der Erkennung von Verschlüsselungstrojanern somit gegeben (*Forschungsfrage CuN-F2*). Die Klassifizierer sind sogar in der Lage, von bekannten Trojanern auf unbekannte zu schließen und diese in Hauptspeicher-Abbildern zu erkennen (*Forschungsfrage CuN-F3*). Über alle Experimente hinweg liefert der Klassifizierer auf Basis des *RandomForest*-Algorithmus die besten Erkennungsergebnisse (*Forschungsfrage CuN-F4*).

Cohen und Nissims Forschungsfragen können erfolgreich mittels eines neuen Satzes an gutartigen Programmen und Verschlüsselungstrojanern beantwortet werden. Ihre in [13] gemachten Beschreibungen reichen aus, um alle notwendigen Schritte von der Infizierung der Virtuellen Maschine über die Merkmalsextraktion bis hin zum Klassifizieren mittels Maschinellen Lernens zu replizieren (*Forschungsfrage F1*). Auch die Verwendung des Betriebssystems *Windows 10 Pro* statt *Windows-Server-2012-R2* hat scheinbar keinen negativen Einfluss auf die Ergebnisse (*Forschungsfrage F2*). Im anspruchsvollsten Experiment 3 können sogar bessere Ergebnisse erzielt werden als in der replizierten Arbeit. In den anderen Experimenten liegt die Leistung etwas unter dieser. Insgesamt hält die Methode jedoch einer Veränderung der Datenbasis stand (*Forschungsfrage F3*). Der Einsatz der beiden Klassifizierer *LibLINEAR* und *LibSVM* scheint nur bei Betrachtung der Experiments 1.1 – 1.3 lohnenswert. Die Experimente 2 und 3, welche realitätsnäher sind, zeigen jedoch, dass insbesondere die Leistung von *LibSVM* nicht jener der anderen Klassifizierer entspricht. Selbst die durchgeführten Parameter-Optimierungen auf den Datensätzen der Experimente 2 und 3 erbringen keine oder nur geringe Verbesserungen. Auch *LibLINEAR* befindet sich unter den leistungsmäßig schlechtesten fünf Klassifizierern dieser beiden Experimente. Somit können Cohen und Nissims Ergebnisse nicht durch den Einsatz von *LibLINEAR* und *LibSVM* verbessert werden (*Forschungsfrage F4*). Die anderen genutzten Familien von

Klassifizierern sind wesentlich erfolgreicher im Umgang mit dem verwendeten Merkmalssatz.

Zusammengefasst lässt sich sagen, dass die von Cohen und Nissim vorgeschlagene Methode einer Veränderung des Gastbetriebssystems, einer Veränderung der Virtualisierungslösung sowie einer geänderten und größeren Auswahl von Verschlüsselungstrojanern und gutartigen Programmen standhält. Die Ergebnisse der replizierten Arbeit lassen sich wiederholen und sind bereits so gut, dass weitere Klassifizierer der SVM-Familie, selbst nach einer Optimierung der Parameter, nicht mithalten können.

5.5 Applikation der Methode in der Praxis

Die hier angewendete Methode dient der Überwachung von Virtuellen Maschinen durch Analyse von Hauptspeicher-Abbildern. Damit soll erkannt werden, ob Verschlüsselungstrojaner auf der Maschine laufen. Zur Anwendung dieser Lösung in einer Organisation, müssen ständig und kontinuierlich Momentaufnahmen der VM in einem vordefinierten Zeitintervall gemacht werden. Jedes dabei erstellte Abbild muss sofort analysiert werden. Dazu müssen die vorgeschlagenen Merkmale (siehe Tab. 3-5) extrahiert und mittels eines auf Maschinellem Lernen basierenden Modells klassifiziert werden. Das Endergebnis ist eine Einstufung, ob die VM infiziert ist oder ausschließlich gutartige Prozesse auf ihr laufen. Wird ein Trojaner erkannt, kann ein automatisierter Prozess angestoßen werden, bei dem z. B. die Netzwerkverbindung deaktiviert oder die Maschine als Ganzes angehalten wird [13].

Im gesamten Prozess kommt der Absicherung des Wirtrechners der überwachten VMs und eines ggf. separaten Analyserechners eine große Bedeutung zu. Werden diese kompromittiert, erfolgt die Erstellung der Hauptspeicher-Abbilder und deren Auswertung nicht mehr auf einem sicheren Weg. Den so erzeugten Aussagen über den Infektionsstatus der VM kann nicht mehr vertraut werden.

Die Hauptspeicher-Abbilder, welche im Büroumfeld häufig 8 GB und aufwärts groß sind, müssen nach der Merkmalsextraktion nicht weiter gespeichert werden. Für VMs, die mit viel RAM ausgerüstet werden, spart dies Ressourcen. Die extrahierten Merkmale können jedoch z. B. in einer SQL-Datenbank zur späteren Überprüfung oder für ein Forttrainieren der Klassifizierer abgelegt werden [13].

Die Modelle des Maschinellen Lernens sind auf eine bestimmte VM trainiert. Eine Übertragbarkeit auf eine andere VM, gerade wenn sich ihr Anwendungszweck unterscheidet (z. B. Büroumfeld versus Softwareentwicklungsumfeld), muss Gegenstand

zukünftiger Forschung sein. Das Modell sollte kontinuierlich auf so vielen gutartigen Instanzen der VM wie möglich trainiert werden [13].

Für die Erstellung des Hauptspeicher-Abbilds muss die VM für einen kurzen Zeitraum angehalten werden. Die Zeit, in der sie nicht reagiert, hängt von der Rechenleistung und den Ressourcen des Hypervisors sowie der Größe des Arbeitsspeichers der Maschine ab. Der Benutzer bemerkt dies durch eine kurze Verzögerung beim Bedienen der VM. Er wird jedoch nicht abgemeldet oder die Maschine heruntergefahren. Sobald das Abbild erstellt wurde, kann er mit seiner Arbeit an der Stelle fortfahren, an der er zuvor unterbrochen wurde. In Zukunft können Virtualisierungsprodukte möglicherweise ein Abbild einer VM erstellen, ohne dass diese angehalten werden muss [13].

5.6 Grenzen der Methode

Die vorgestellte Methode hat Grenzen, welche bei ihrer Applikation in der Praxis zu beachten sind. Diese betreffen u.a. den Umgang des Betriebssystems mit dem RAM.

Ein laufendes System unter Windows besteht aus hunderten von Prozessen, welche im RAM abgebildet sind. Der Inhalt des RAMs ändert sich ständig in Abhängigkeit von den laufenden Prozessen, aber auch dem zur Verfügung stehenden Speicherplatz und den zugehörenden *Auslagerungsrichtlinien* (engl. *paging policy*). Aufgrund der dort getroffenen Einstellungen lagert Windows automatisch Teile des Hauptspeichers auf den Massenspeicher aus. So wird Platz für neue Prozesse geschaffen. Werden diese Teile später wieder benötigt, kann Windows sie unter kurzer Verzögerung zurück in den RAM laden und die Ausführung der Prozesse fortsetzen. Über diese Funktion ist es Betriebssystemen möglich, wesentlich mehr und umfangreichere Prozesse gleichzeitig auszuführen. Eine Auswirkung ist jedoch, dass nicht zu jedem Zeitpunkt alle Informationen über laufende Prozesse im RAM vorliegen. Ein Abbild des Hauptspeichers enthält dann keine Merkmale ausgelagerter Prozesse [13].

Die vorgestellte Methode bezieht sich auf das Betriebssystem *Windows 10 Pro*. Andere Betriebssysteme können eine abweichende Struktur des Hauptspeichers aufweisen und somit auch andere Merkmale enthalten. Sollen diese analysiert werden, müssen ggf. Anpassungen am verwendeten Merkmalssatz gemacht werden.

Ähnliches gilt auch für den Schadcode, der mit dieser Methode entdeckt werden kann. Viren, Trojaner oder Würmer nutzen unterschiedliche Vorgehensweisen zur Ausbreitung und zur Erreichung ihrer Ziele. In dieser Arbeit werden Verschlüsselungstrojaner, eine besondere Subkategorie der Trojaner, untersucht. Ob die vorgeschlagenen

Merkmale auch bei anderem Schadcode effektiv sind, kann auf Basis der hier vorgestellten Untersuchungen nur vermutet werden. Auch die Reaktion auf erkannten Schadcode durch z. B. automatisierte Routinen ist nicht Bestandteil dieser Arbeit.

Setzt man den vorgeschlagenen Weg zur Datenerhebung in der Praxis ein, leidet das Benutzererlebnis des überwachten Systems durch die Erstellung der Hauptspeicher-Abbilder und das dafür notwendige Anhalten der VM. Die Dauer dieses Vorgangs ist, genauso wie die Analyse der Abbilder mit Volatility, stark abhängig von der Speichergröße des RAMs [13]. Ein Transport der Abbilder über das Netzwerk verbietet sich bei hier gut ausgestatten VMs.

Auch muss die Zeitverzögerung durch die Analyse beachtet werden, wenn man die Reaktion auf eine Infektion plant. Diese kann durch den Autoren dieser Arbeit von 16 Minuten (siehe [13]) auf 1,86 Minuten für ein Abbild von 1 GB Größe gesenkt werden. Dies ist jedoch immer noch viel, wenn man die Ausführungszeiten der Trojaner betrachtet (siehe Tab. 3-4).

Wie alle Lösungen auf Basis von Maschinellem Lernen unterliegt auch die hier vorgestellte Methode einer laufenden *Konzeptveränderung* (engl. *concept drift*). So kann sich im Zeitverlauf die Interpretation der Merkmale ändern. Ein Merkmal, welches früher auf eine Infektion hingedeutet hat, könnte dann durch ein gutartiges Programm verursacht werden oder andersherum. Eine Konzeptveränderung entsteht, wenn sich die Datengrundlage ändert. Ausgelöst wird sie z. B. durch Veränderungen der Trojaner, welche nun neue Vorgehensweisen für die Infektion oder die Verschlüsselung nutzen. Hierdurch wird die Detektionsleistung der Modelle reduziert, wenn keine Gegenmaßnahmen in Form eines ständig fortgesetzten Trainings bzw. eines regelmäßigen komplett neuen Trainings der Klassifizierer auf Basis eines aktuellen Datensatzes erfolgen.

6 Zusammenfassung und Ausblick

Verschlüsselungstrojaner stellen einen noch relativ jungen Trend im Bereich *Schadcode* dar, der jedoch aufgrund seiner Lukrativität massenhaften Einsatz gefunden hat und immer noch findet. Sie bedrohen nicht nur die heimischen PCs von privaten Nutzern, sondern auch die *Cloud*-Umgebungen von Unternehmen. Die *Cloud* hat aufgrund weitgehend virtualisierter Systeme besondere Sicherheitsanforderungen. Gleichzeitig bietet sie aber auch besondere Möglichkeiten zur Bekämpfung von Trojanern dieser Art. Diese Arbeit knüpft an die bestehende Forschung auf diesem Gebiet an. Die Grundlage dazu bildet die von Cohen und Nissim [13] vorgestellte Methode zur Erkennung von Verschlüsselungstrojanern in Hauptspeicher-Abbildern virtualisierter Server mittels Maschinellen Lernens. Auf diesem Weg kann ein System als Ganzes geschützt werden. Heutzutage übliche Antiviren-Programme sind dazu häufig nicht in der Lage. Sie nutzen dynamische und statische Analysen und können deswegen zu jedem Zeitpunkt nur einzelne auffällige Dateien auf Schadcode überprüfen. Zukünftig könnte Cohen und Nissims Methode diese herkömmlichen Schutzmaßnahmen von IT-Systemen ergänzen oder sogar ersetzen. Bevor die Methode jedoch in der Praxis appliziert werden kann, sind weitergehende Untersuchungen nötig.

6.1 Zusammenfassung

Ziel dieser Arbeit ist es, zu überprüfen, ob die vorgestellte Methode reproduzierbar ist und auch in einer veränderten Umgebung, sowie beim Vorlegen eines neuen und erweiterten Satzes von Verschlüsselungstrojanern ihre Wirksamkeit bei der Detektion von Schadcode erhalten kann. Der in [13] vorgeschlagene Satz von neun Klassifikations-Algorithmen wurde zusätzlich um zwei weitere (*LibSVM, LibLINEAR*) ergänzt, um deren Leistungsfähigkeit in diesem Umfeld zu überprüfen und so ggf. Kandidaten für eine Verbesserung der Detektionsleistung zu finden.

Dazu wurde die von den Autoren vorgeschlagenen Datenpipeline, beginnend mit der Datenerhebung in einer Virtuellen Maschine, repliziert. Zusätzlich wurden dabei folgende Änderungen und Erweiterungen vorgenommen:

- Untersuchung eines neuen und erweiterten Satzes von jeweils zehn Verschlüsselungstrojanern und zehn gutartigen Programmen
- Einsatz von *VirtualBox* statt *ESXi* als Hypervisor
- Einsatz von *Windows 10 Pro* statt *Windows-Server-2012-R2*

Schadcode unterliegt einem ständigen Wandel. Eine Methode zur Erkennung von Trojanerbefall muss daher robust gegenüber diesem Wandel sein und auch zuverlässig

arbeiten, wenn neue Angriffsvektoren genutzt werden. In dieser Arbeit wurde mit *Dynamer*, *GandCrab*, *Jigsaw*, *Lockergoga*, *Occamy_C*, *RedPetya*, *Scarab*, *Sodinokibi, Wadhrama* und *Wannacry* eine Auswahl von professionellen und teilweise noch im Einsatz befindlichen Verschlüsselungstrojanern der letzten Jahre untersucht. Ihnen wurden die im Büro-Umfeld üblichen gutartigen Programme *Avast*, *Avira*, *Kaspersky*, *LibreOffice Writer*, *Thunderbird*, *VLC Player*, *Visual Studio Code*, *GIMP* und *TotalCommander* gegenübergestellt.

Das Betriebssystem wurde mit *Windows 10 Pro* an die in privaten und öffentlichen Organisationen üblichen Computer-Systeme für Endanwender angepasst. Der Schutz der Endgeräte vor Verschlüsselungstrojanern ist in diesem Bereich von besonderer Bedeutung. Auch die Virtualisierungsumgebung muss austauschbar sein, ohne dass die Methode hierdurch in ihrem Detektionsvermögen eingeschränkt wird, da sonst der potenzielle Anwenderkreis massiv zusammenschrumpft. In dieser Arbeit wurde deswegen *VirtualBox* als Hypervisor eingesetzt.

Für die automatisierte Erhebung der Hauptspeicher-Abbilder wurde eine Bibliothek in Python entwickelt, welche die API von *VirtualBox* nutzt. Auch die weitere Verarbeitung der Abbilder in der Analyse mit *Volatility* und der anschließenden Merkmalsextraktion wurde automatisiert und parallelisiert. Durch diese Maßnahmen konnte die Zeit, welche für die Analyse eines Abbilds notwendig ist, von 16 Minuten in [13] auf durchschnittlich 1,86 Minuten gesenkt werden.

Es wurden die von Cohen und Nissim vorgeschlagenen Merkmale genutzt. Diese nehmen zum großen Teil sowohl auf einzelne wichtige Elemente des Betriebssystems Bezug als auch auf die Anzahl von laufenden Prozessen und Services, die geladenen DLLs oder die von Prozessen genutzten Privilegien. Daneben wird von den Autoren aber auch ein automatisierter Vergleich von auffälligen Prozessen gegen eine Whitelist angestellt. Damit wird eine typische Aufgabe der Hauptspeicher-Forensik automatisiert und in den Merkmalssatz aufgenommen. Für die Auswertung der erzeugten Merkmalsvektoren wurde *Weka* eingesetzt. Dieses Rahmenwerk ermöglicht ein effizientes Training und Testing der elf angewandten Algorithmen des Maschinellen Lernens.

Die Methode wurde in drei umfassenden Experimenten mit steigendem Schwierigkeitsgrad für die Klassifizierer untersucht. Das erste Experiment bestand wiederum aus drei untergeordneten Aufgaben (Experimente 1.1-1.3) mit dem Ziel, verschiedene bekannte Zustände auf der VM zu erkennen. In der ersten Aufgabe sollten Abweichungen von einem Normal-Zustand identifiziert werden. Die zweite Aufgabe verfolgte die Absicht, zwischen infizierten und unauffälligen Zuständen zu unterscheiden. Danach sollten in der dritten Aufgabe spezifische bekannte Zustände, charakterisiert durch die untersuchten Programme und Trojaner, bestimmt werden. Die Ergebnisse waren

bei so gut wie allen Klassifzierern sehr gut. Besonders hervorzuheben ist der auf dem *RandomForest*-Algorithmus basierende Klassifizierer, welcher bei allen drei Aufgaben eine perfekte Erkennungsrate erreichte (*RPR = 1, FPR = 0, AUC = 1* und *F-Maß = 1*). Bis auf das Verhalten des *AdaBoostM1*-Klassifizierers in Experiment 1.3 decken sich die Ergebnisse mit denen von Cohen und Nissim. Die Ergebnisse zeigen, dass mit der genutzten Methode zuverlässig bekannte Verschlüsselungstrojaner und bekannte gutartige Programme detektiert werden können.

Im zweiten Experiment wurde bewertet, ob die Methode geeignet ist, unbekannte infizierte Zustände auf der VM zu erkennen. Insgesamt fielen die Werte von RPR und AUC etwas besser. Die Ergebnisse der FPR dagegen waren etwas schlechter (höher) als in der replizierten Arbeit. Der *RandomForest*-Algorithmus lieferte auch hier das beste Modell für die Daten (*RPR = 0,972, FPR = 0,073, AUC = 1* und *F-Maß = 0,91*). Auf Basis der vorliegenden Ergebnisse kann festgestellt werden, dass unbekannte Verschlüsselungstrojaner bei gleichzeitigem Vorhandensein von bekannten gutartigen Programmen mit hoher Wahrscheinlichkeit erkannt werden.

Im dritten und anspruchsvollsten Experiment sollten unbekannte infizierte Zustände unter unbekannten unauffälligen Zuständen festgestellt werden. Die Ergebnisse von Experiment 3 deckten sich erneut gut mit denen in [13]. Die Werte von RPR, AUC und F-Maß fielen insgesamt sogar etwas besser aus als bei Cohen und Nissim. Die FPR-Werte waren ebenfalls etwas besser (niedriger). Dieses schwierigste der durchgeführten Experimente scheint vom größeren Datensatz profitiert zu haben. Es wurde jedoch festgestellt, dass die Klassifizierer bei einzelnen der 100 Durchführungen dieses Experimentes Schwierigkeiten hatten, einer der beiden Klassen Instanzen zuzuordnen. Es ist nicht bekannt, ob dieses Problem ebenfalls in den Experimenten von Cohen und Nissim auftrat, da in [13] keine Angaben darübergemacht werden. Eine genaue Untersuchung zu den Ursachen für diese Abweichung hätte den Rahmen dieser Arbeit gesprengt und wurde daher nicht durchgeführt. Wiederum erbrachte die Anwendung von *RandomForest* die besten Ergebnisse (*RPR = 0,883, FPR = 0,117, AUC = 0,981* und *F-Maß = 0,908*).

Cohen und Nissims Experimente konnten somit erfolgreich auf einem neuen Satz von gutartigen Programmen und Verschlüsselungstrojanern durchgeführt werden. Ihre in [13] gemachten Beschreibungen reichten aus, um alle notwendigen Schritte von der Infizierung der Virtuellen Maschine über die Merkmalsextraktion bis hin zum Klassifizieren mittels Maschinellen Lernens zu replizieren. Auch die Verwendung des Betriebssystems *Windows 10 Pro* statt *Windows-Server-2012-R2* und der Wechsel der Virtualisierungslösung zu *VirtualBox* hatten keinen negativen Einfluss auf die Ergebnisse. In den ersten beiden Experimenten wurden zwar etwas schlechtere Ergebnisse

erzielt als in der ursprünglichen Arbeit. Dafür wurde die Erkennungsleistung im anspruchsvollsten Experiment 3 verbessert. Die Methode kann insgesamt einer Veränderung der Datenbasis standhalten. Dies wird vor allem daran deutlich, dass die Forschungsfragen der replizierten Arbeit auf Basis der hier erbrachten Ergebnisse erneut beantwortet werden konnten und derselbe Algorithmus (*RandomForest*) über alle Experimente die beste Detektionsleistung erbrachte.

Der Einsatz der beiden Klassifizierer *LibLINEAR* und *LibSVM* scheint nur bei Betrachtung des Experiments 1 mit seinen Unteraufgaben 1.1 – 1.3 lohnend. Die Experimente 2 und 3, welche realitätsnäher sind, zeigten jedoch, dass insbesondere die Leistung von *LibSVM* nicht jener der anderen Klassifizierer entspricht. Auch eine Optimierung der *LibSVM*-Parameter mit *GridSearch* auf dem jeweiligen Datensatz brachte keine signifikanten Verbesserungen. *LibLINEAR* befindet sich ebenfalls unter den leistungsmäßig schlechtesten fünf Klassifizierern dieser beiden Experimente. Somit können Cohen und Nissims Ergebnisse nicht durch den Einsatz von *LibLINEAR* und *LibSVM* verbessert werden. Die ursprünglich genutzten Algorithmen sind wesentlich erfolgreicher im Umgang mit dem verwendeten Merkmalssatz.

Vor dem Hintergrund der Schäden durch Verschlüsselungstrojaner, welche Organisationen weltweit in den letzten Jahren zugefügt wurden, ist klar, dass eine solche Methode benötigt wird. Diese Arbeit kann nachweisen, dass die Methode umfassend genug beschrieben ist, um sie zu replizieren. Sie ist robust gegenüber einer Veränderung der Umgebung und auch bezüglich der Verschlüsselungstrojaner und gutartigen Programme. Auch die Auswahl der Klassifizierer ist der Aufgabe gut gewachsen. Durch Optimierungen kann die Analyse der Daten noch wesentlich beschleunigt werden. Insgesamt stellt die Methode somit einen effizienten und vertrauenswürdigen Weg zum Schutz von Virtuellen Maschinen vor Verschlüsselungstrojanern dar.

6.2 Ausblick

Die Ansatzpunkte für Verbesserungen und die Weiterentwicklung der Methode entlang ihrer Datenpipeline sind groß. Dies liegt vor allem an der Vielfalt von unterschiedlichen betroffenen wissenschaftlichen Disziplinen (*IT-Forensik, Betriebssystementwicklung, Big Data, Data Science* usw.).

Betrachtete man den ersten Teil der Datenpipeline, so ist es von besonderem Interesse, nach Wegen zu suchen, wie die Erstellung der Hauptspeicher-Abbilder ohne ein Anhalten der Virtuellen Maschine möglich ist. Dies würde die Anwendbarkeit der Methode in der Praxis enorm steigern, da die Nutzer der VM in ihrer Arbeit nicht gestört werden. Auch die Untersuchung, wie die benötigten Volatility-Plugins kontinuierlich

auf dem RAM der VM arbeiten können, ist interessant. Zwischen den einzelnen Hauptspeicher-Abbildern liegt aktuell viel Zeit, in der keine Aussage über den Zustand der VM möglich ist. Werden diese Zeiten minimiert bzw. komplett beseitigt, kann schneller auf eine Trojaner-Infektion reagiert werden.

Bisher werden nur die Hauptspeicher-Abbilder der VM untersucht. Es ist jedoch möglich, dass viele wichtige Hinweise für das Vorhandensein eines Verschlüsselungstrojaners nicht im RAM vorliegen, sondern vom Betriebssystem auf die Festplatte ausgelagert wurden. Das Einbeziehen der Auslagerungsdatei in die Analyse des Systems sollte daher in Zukunft einen Schwerpunkt der Weiterentwicklung der Methode bilden.

Da Verschlüsselungstrojaner unterschiedliche Phasen der Ausführung haben, ist eine genaue Untersuchung dahingehend notwendig, in welchen Phasen die Methode die Trojaner sicher erkennt und bei welchen anderen Phasen eine Anpassung der Methode notwendig ist. Neben Verschlüsselungstrojanern gibt es viele weitere Schadcode-Arten, welche eine Antivirus-Lösung nicht unberücksichtigt lassen kann. Die Methode sollte daher auf ihre Leistungsfähigkeit bezüglich der Detektion dieser Arten hin untersucht werden.

Aktuell werden die Algorithmen des Maschinellen Lernens auf einer bestimmten VM trainiert und mussten in den anschließenden Tests in den Hauptspeicher-Abbildern dieser VM die Trojaner erkennen. Es gibt noch keine Erkenntnisse darüber, wie leistungsfähig die Klassifizierer sind, wenn sie auf anderen VMs getestet werden. Es ist zu vermuten, dass die Erkennungsleistung der trainierten Modelle sinkt, wenn sie auf VMs mit einem anderen Einsatzzweck getestet werden. Auch die Parameter der eingesetzten Algorithmen wurden bisher nicht optimiert. Gegebenenfalls ließen sich dadurch deutliche Leistungspotentiale in noch schwierigeren Experimenten freilegen.

Bisherige Untersuchungen der Methode sind nur im Labor und unter künstlichen Bedingungen abgelaufen. Die Methode in der Praxis bzw. in einer praxisnahen Umgebung zu untersuchen, stellt große Anforderungen an das Experimentdesign, aber auch an die dafür notwendige Vorbereitung. Dieser Aufwand würde sich jedoch dahingehend lohnen, dass ein direkter Vergleich zu herkömmlichen Methoden der Schadcode-Erkennung möglich ist.

Literaturverzeichnis

[1] M. H. Ligh, A. Case, J. Levy, und Aa. Walters, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. 2014.

[2] M. H. Weik und M. H. Weik, „Terminate and Stay Resident Viruses“, *Comput. Sci. Commun. Dict.*, Bd. 1, Nr. 11, S. 1766–1766, 2000.

[3] S. M. Pg Scholar und M. Krishnan, „Forensic Recovery of Fully Encrypted Volume“, *Int. J. Comput. Appl.*, Bd. 91, Nr. 7, S. 975–8887, 2014.

[4] Bundesamt für Sicherheit in der Informationstechnik, „Die Lage der IT-Sicherheit in Deutschland 2019“. Bonn, S. 1–80, 2019.

[5] Google, „Rekall Forensics“, 2019. [Online]. Verfügbar unter: http://rekall-forensic.com.

[6] The Volatility Foundation, „Volatility“, 2020. [Online]. Verfügbar unter: https://www.volatilityfoundation.org/.

[7] S. de Courcier, „How To Use AXIOM In Malware Investigations: Part I“, *Forensic Focus - Articles*, 2019. [Online]. Verfügbar unter: https://articles.forensicfocus.com/2019/11/11/how-to-use-axiom-in-malware-investigations-part-i/. [Zugegriffen: 14-Feb-2020].

[8] Belkasoft, „Forensic analysis of files and memory processes in Belkasoft Evidence Center 2017“, *Belkasoft Publications*, 2017. [Online]. Verfügbar unter: https://belkasoft.com/file-system-and-process-explorer. [Zugegriffen: 14-Feb-2020].

[9] X-Ways Software Technology AG, „Benutzerhandbuch X-Ways Forensics & WinHex“, 2019. [Online]. Verfügbar unter: http://www.x-ways.net/winhex/manual-d.pdf. [Zugegriffen: 14-Feb-2020].

[10] Gartner Group, „Big Data“, *Gartner Glossray*, 2020. [Online]. Verfügbar unter: https://www.gartner.com/en/information-technology/glossary/big-data. [Zugegriffen: 14-Feb-2020].

[11] Bibliographisches Institut GmbH, „Duden | Data-Mining | Rechtschreibung, Bedeutung, Definition, Herkunft“, *Duden Wörterbuch Online*, 2020. [Online]. Verfügbar unter: https://www.duden.de/rechtschreibung/Data_Mining. [Zugegriffen: 24-Feb-2020].

[12] S. Luber und N. Litzel, „Was ist Machine Learning?“, *BigData Insider Definitionen*, 01-Sep-2016. [Online]. Verfügbar unter: https://www.bigdata-insider.de/was-ist-machine-learning-a-592092/. [Zugegriffen: 14-Feb-2020].

[13] A. Cohen und N. Nissim, „Trusted detection of ransomware in a private cloud using machine learning methods leveraging meta-features from volatile memory“, *Expert Syst. Appl.*, Bd. 102, S. 158–178, Juli 2018.

[14] StatCounter, „Desktop Operating System Market Share Worldwide“, *StatCounter Global Stats*, 2020. [Online]. Verfügbar unter: https://gs.statcounter.com/os-market-share/desktop/worldwide. [Zugegriffen: 14-Feb-2020].

[15] StatCounter, „Desktop Windows Version Market Share Worldwide“, *StatCounter Global Stats*, 2020. [Online]. Verfügbar unter: https://gs.statcounter.com/os-version-market-share/windows/desktop/worldwide. [Zugegriffen: 14-Feb-2020].

[16] M. Field, „WannaCry cyber attack cost the NHS £92m as 19,000 appointments cancelled“, *The Telegraph Website*, 11-Okt-2018. [Online]. Verfügbar unter: https://www.telegraph.co.uk/technology/2018/10/11/wannacry-cyber-attack-cost-nhs-92m-19000-appointments-cancelled/. [Zugegriffen: 14-Feb-2020].

[17] N. A. Hassan, *Ransomware Revealed*. New York: Apress, 2019.

[18] F. Sinitsyn, „Petya: the two-in-one trojan“, *Securelist - Malware Descriptons*, 04-Mai-2016. [Online]. Verfügbar unter: https://securelist.com/petya-the-two-in-one-trojan/74609/. [Zugegriffen: 16-Feb-2020].

[19] V. C. Craciun, A. Mogage, und E. Simion, „Trends in design of ransomware viruses“, in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2019, S. 259–272.

[20] S. Morgan, „Global Cybercrime Damages Predicted To Reach $6 Trillion Annually By 2021“, *Cybercrime Magazine*, 07-Dez-2018. [Online]. Verfügbar unter: https://cybersecurityventures.com/cybercrime-damages-6-trillion-by-2021/. [Zugegriffen: 16-Feb-2020].

[21] Bundesamt für Sicherheit in der Informationstechnik, „Ransomware - Bedrohungslage, Prävention & Reaktion 2019“. Bonn, S. 1–35, 2019.

[22] J. Hwang, J. Kim, S. Lee, und K. Kim, „Two-Stage Ransomware Detection Using Dynamic Analysis and Machine Learning Techniques“, *Wirel. Pers. Commun.*, Nr. 0123456789, 2020.

[23] A. Souri und R. Hosseini, „A state-of-the-art survey of malware detection approaches using data mining techniques“, *Human-centric Comput. Inf. Sci.*, Bd. 8, Nr. 1, S. 22, 2018.

[24] Y. Ye, T. Li, D. Adjeroh, und S. S. Iyengar, „A survey on malware detection using data mining techniques“, *ACM Comput. Surv.*, Bd. 50, Nr. 3, S. 40, 2017.

[25] P. Szor, *The art of computer virus research and defense*. Upper Saddle River: Addison-Wesley Professional, 2005.

[26] F. Biondi, T. Given-Wilson, A. Legay, C. Puodzius, und J. Quilbeuf, „Tutorial: An overview of malware detection and evasion techniques“, in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2018.

[27] Randy Treit, „Detonating a bad rabbit: Windows Defender Antivirus and layered machine learning defenses - Microsoft Security“, *Microsoft Security Blog*, 11-Dez-2017. [Online]. Verfügbar unter: https://www.microsoft.com/security/blog/2017/12/11/detonating-a-bad-rabbit-windows-defender-antivirus-and-layered-machine-learning-defenses/. [Zugegriffen: 24-Feb-2020].

[28] Jeff Elder, „Avast Explains Cybersecurity AI at Enigma Conference | Avast“, *Avast Blog*, 27-Jan-2020. [Online]. Verfügbar unter: https://blog.avast.com/avast-explains-cybersecurity-ai-at-enigma-conference.
[Zugegriffen: 24-Feb-2020].

[29] B. für S. in der Informationstechnik, *Leitfaden „IT-Forensik“*, 1.0.1., Bd. 1, Nr. März. Bonn: Bundesamt für Sicherheit in der Informationstechnik, 2011.

[30] Ted Hudek und Nathan Bazan, „User mode and kernel mode - Windows drivers“, *Microsoft Docs - Hardware Dev Center*, 20-Apr-2017. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode.
[Zugegriffen: 02-März-2020].

[31] Ted Hudek und Nathan Bazan, „Virtual address spaces - Windows drivers“, *Microsoft Docs - Hardware Dev Center*, 20-Apr-2017. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/virtual-address-spaces.
[Zugegriffen: 02-März-2020].

[32] OpenText Corp., „Tableau Forensic USB 3.0 Bridge - T8u“, *Tableau Forensic USB 3.0 Bridge Marketing Material*, 2019. [Online]. Verfügbar unter: https://www.guidancesoftware.com/tableau/hardware//t8u.
[Zugegriffen: 04-März-2020].

[33] ACCESSDATA, „FTK Imager 4.2.0“, *FTK Imager Marketing Material*, 2020. [Online]. Verfügbar unter: https://marketing.accessdata.com/ftkimager4.2.0.
[Zugegriffen: 04-März-2020].

[34] Volatility Foundation, „GitHub - volatilityfoundation/volatility: An advanced memory forensics framework“, *Github Repository*, 03-März-2020. [Online]. Verfügbar unter: https://github.com/volatilityfoundation/volatility.
[Zugegriffen: 05-März-2020].

[35] Volatility Foundation, „GitHub - volatilityfoundation/volatility3: Volatility 3.0 development“, *Volatility 3.0 Repository*, 30-Jan-2020. [Online]. Verfügbar unter: https://github.com/volatilityfoundation/volatility3.
[Zugegriffen: 05-März-2020].

[36] G. Rebala, A. Ravi, und S. Churiwala, *An introduction to machine learning*, 1. Aufl. Cham, Schweiz: Springer Nature Switzerland, 2019.

[37] I. H. Witten, E. Frank, und M. A. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, 3. Aufl. Burlington, MA, USA: Morgan Kaufmann Publishers, 2011.

[38] E. Alpaydin, *Introduction to Machine Learning*, 2. Aufl. Cambridge, MA, USA: The MIT Press, 2010.

[39] C. Shearer, „The CRISP-DM model: The New Blueprint for Data Mining“, *J. Data Warehous.*, Bd. 5, Nr. 4, S. 13–22, 2000.

[40] T. A. Runkler, *Data-Mining : Methoden und Algorithmen intelligenter Datenanalyse*, 1. Aufl. Wiesbaden: Vieweg + Teubner, 2010.

[41] F. Provost und T. Fawcett, *Data science for business : What you need to know about data mining and data-analytic thinking*, 1. Aufl. Sebastopol: O'Reilly, 2013.

[42] D. Hand und P. Christen, „A note on using the F-measure for evaluating record linkage algorithms“, *Stat. Comput.*, Bd. 28, Nr. 3, S. 539–547, Mai 2018.

[43] „Oracle VM VirtualBox“. [Online]. Verfügbar unter: https://www.virtualbox.org/. [Zugegriffen: 06-Apr-2020].

[44] Oracle, „12.1.5. VM Core Format“, *VirtualBox Manual*. [Online]. Verfügbar unter: https://www.virtualbox.org/manual/ch12.html#ts_guest-core-format. [Zugegriffen: 07-Apr-2020].

[45] Oracle, „Core_dump – Oracle VM VirtualBox“, *VirtualBox Wiki*. [Online]. Verfügbar unter: https://www.virtualbox.org/wiki/Core_dump. [Zugegriffen: 07-Apr-2020].

[46] Open Watcom, „ELF-64 Object File Format Overview of an ELF file“, *ELF-64 Object File Format, Version 1.5 Draft 2*. S. 1–18, 1998.

[47] COCO Consortium, „COCO - Common Objects in Context“. [Online]. Verfügbar unter: http://cocodataset.org/#home. [Zugegriffen: 11-Apr-2020].

[48] Microsoft, „Word templates“, *Office Templates Website*. [Online]. Verfügbar unter: https://templates.office.com/en-us/templates-for-word. [Zugegriffen: 11-Apr-2020].

[49] The MalShare Project, „MalShare“. [Online]. Verfügbar unter: https://malshare.com/. [Zugegriffen: 13-Apr-2020].

[50] „VirusShare“. [Online]. Verfügbar unter: https://virusshare.com/. [Zugegriffen: 13-Apr-2020].

[51] „VirusTotal“. [Online]. Verfügbar unter: https://www.virustotal.com/gui/home. [Zugegriffen: 13-Apr-2020].

[52] Avast Software s.r.o., „Avast 2020“, *Produktwebseite Avast Antivirus*. [Online]. Verfügbar unter: https://www.avast.com/de-de/index#pc. [Zugegriffen: 13-Apr-2020].

[53] Avira Operations GmbH & Co. KG, „Avira“, *Produktwebseite Avira Antivirus*. [Online]. Verfügbar unter: https://www.avira.com/de. [Zugegriffen: 13-Apr-2020].

[54] AO Kaspersky Lab., „Kaspersky“, *Produktwebseite Kaspersky Antivirus*. [Online]. Verfügbar unter: https://www.kaspersky.de/. [Zugegriffen: 13-Apr-2020].

[55] The Document Foundation, „LibreOffice“, *Produktwebseite LibreOffice*. [Online]. Verfügbar unter: https://de.libreoffice.org/. [Zugegriffen: 13-Apr-2020].

[56] MZLA Technologies Corporation, „Thunderbird“, *Produktwebseite Thunderbird*. [Online]. Verfügbar unter: https://www.thunderbird.net/de/. [Zugegriffen: 13-Apr-2020].

[57] VideoLAN, „VLC Player“, *Produktwebseite VLC Player*. [Online]. Verfügbar unter: https://www.videolan.org/vlc/index.de.html. [Zugegriffen: 13-Apr-2020].

[58] Microsoft, „Visual Studio Code“, *Produktwebseite Visual Studio Code*. [Online]. Verfügbar unter: https://code.visualstudio.com/. [Zugegriffen: 13-Apr-2020].

[59] The GIMP Team, „GIMP“, *Produktwebseite GIMP*. [Online]. Verfügbar unter: https://www.gimp.org/. [Zugegriffen: 13-Apr-2020].

[60] Ghisler Software GmbH, „Total Commander“, *Produktwebseite TotalCommander*. [Online]. Verfügbar unter: https://www.ghisler.com/deutsch.htm. [Zugegriffen: 13-Apr-2020].

[61] I. ECMA, „The JavaScript Object Notation (JSON) Data Interchange Format“, *ECMA Int.*, Bd. 1st Editio, Nr. October, S. 8, 2014.

[62] Microsoft, „Callback Objects - Windows drivers“, *Microsoft Docs*. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/callback-objects. [Zugegriffen: 15-Apr-2020].

[63] Microsoft, „Dynamic-Link Libraries (Dynamic-Link Libraries) - Win32 apps“, *Microsoft Docs*. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/windows/win32/dlls/dynamic-link-libraries. [Zugegriffen: 15-Apr-2020].

[64] Microsoft, „Handles and Objects - Win32 apps“, *Microsoft Docs*. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/windows/win32/sysinfo/handles-and-objects. [Zugegriffen: 15-Apr-2020].

[65] Microsoft, „Module Information - Win32 apps“, *Microsoft Docs*. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/windows/win32/psapi/module-information. [Zugegriffen: 16-Apr-2020].

[66] Microsoft, „Mutex Objects - Win32 apps“, *Microsoft Docs*. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/windows/win32/sync/mutex-objects. [Zugegriffen: 16-Apr-2020].

[67] Microsoft, „Privileges - Win32 apps“, *Microsoft Docs*. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/windows/win32/secauthz/privileges. [Zugegriffen: 16-Apr-2020].

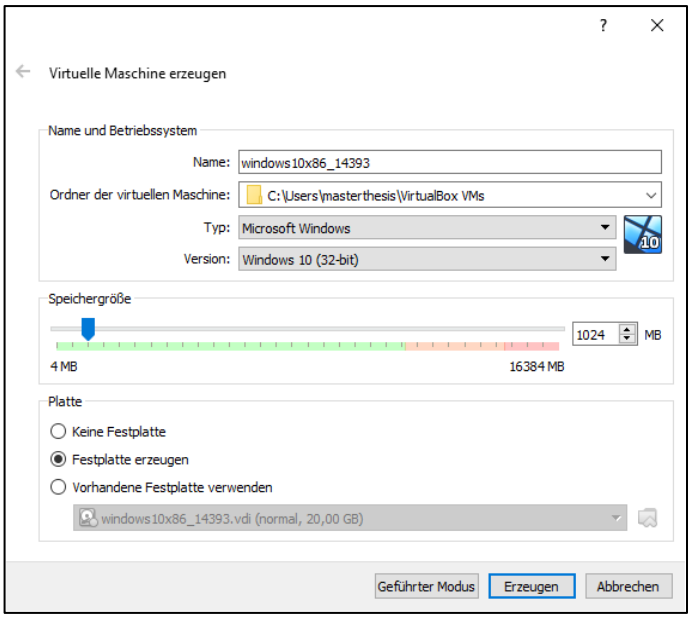
[68] Microsoft, „Processes and Threads - Win32 apps“, *Microsoft Docs*. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/windows/win32/procthread/processes-and-threads. [Zugegriffen: 16-Apr-2020].

[69] Microsoft, „Services - Win32 apps“, *Microsoft Docs*. [Online]. Verfügbar unter: https://docs.microsoft.com/en-us/windows/win32/services/services. [Zugegriffen: 16-Apr-2020].

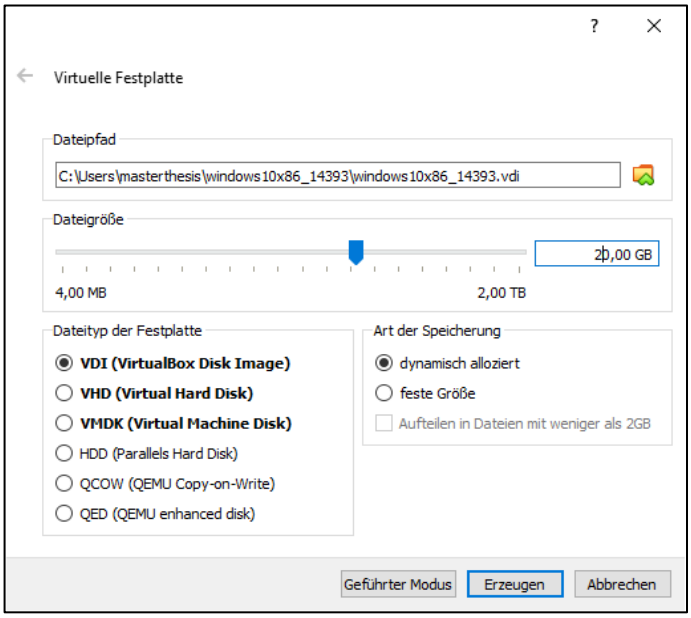
Anhang

A Installation der Virtuellen Maschinen

1. **Herunterladen der Windows 10 Pro Version vom offiziellen Microsoft Server**
2. **In VirtualBox eine neue VM anlegen:**
   a. Erste Maske im VM-erzeugen-Wizard:

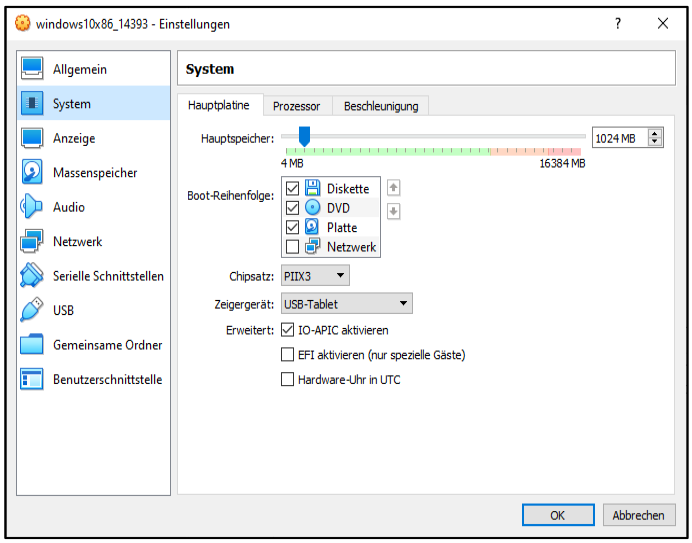


   b. Zweite Maske im VM-erzeugen-Wizard:

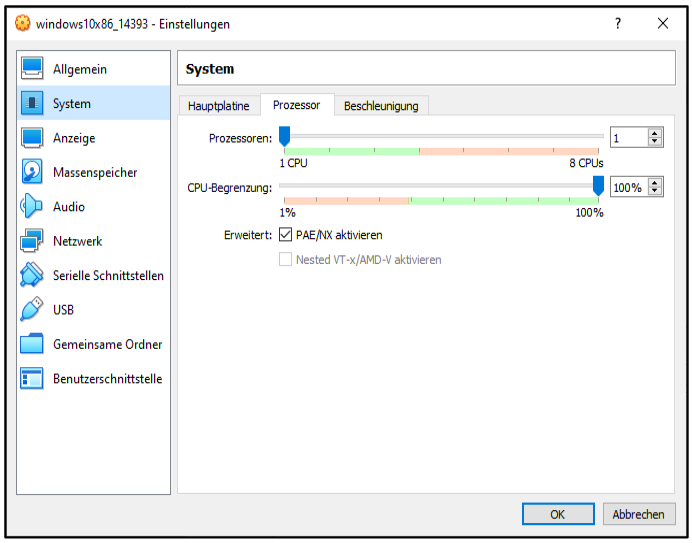


3. Erstellte VM ändern:

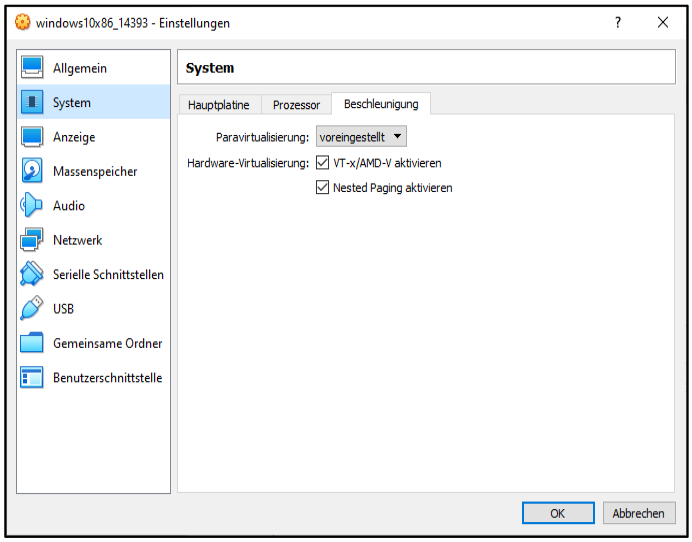
a. System: Reiter Hauptplatine:



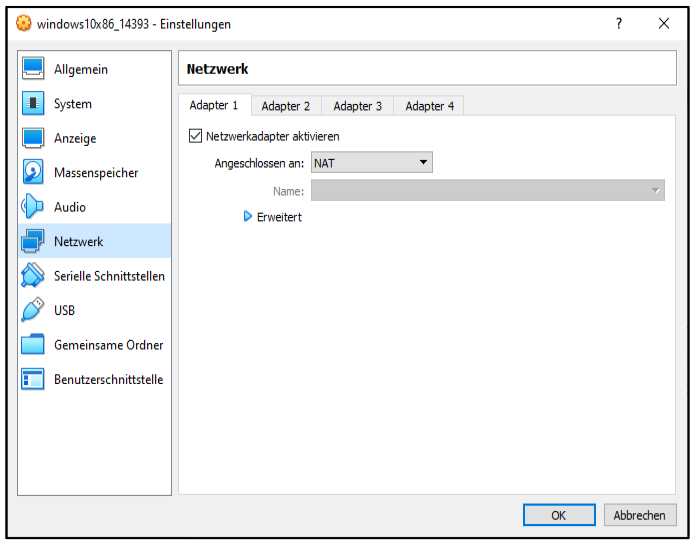
b. System: Reiter Prozessor:



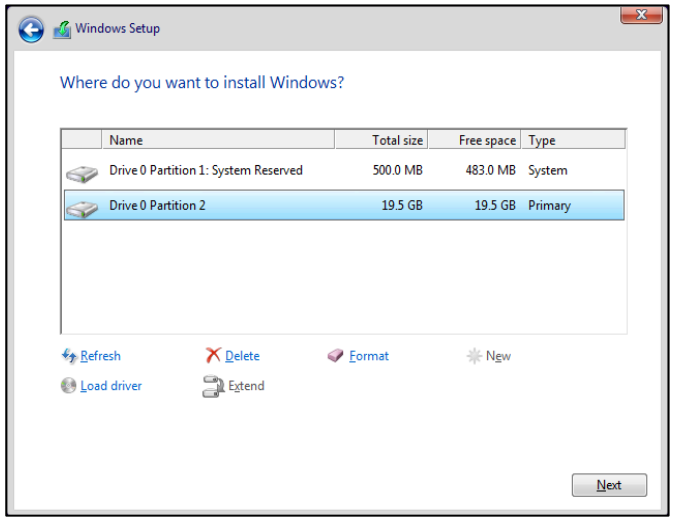
c. System: Reiter Beschleunigung:



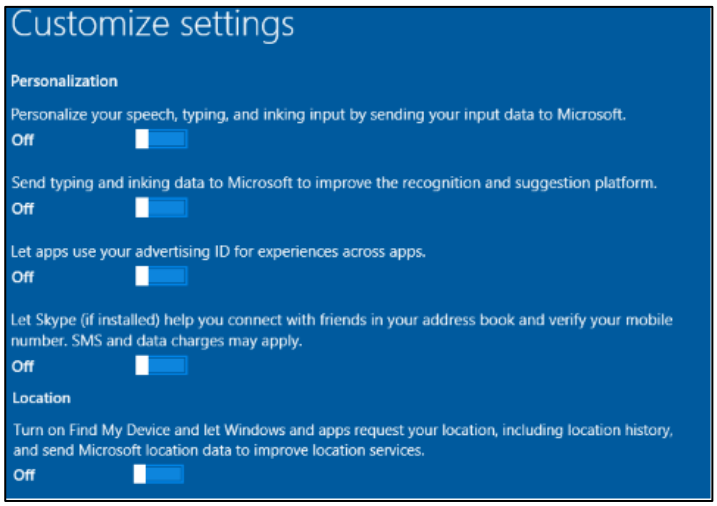
d. Netzwerk: Reiter Adapter 1:



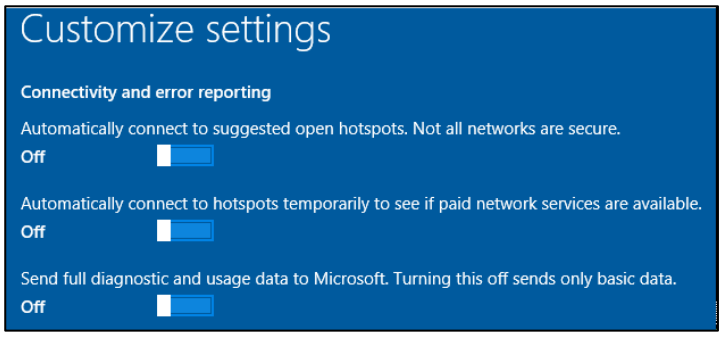
4. **VM starten und Windows 10 Pro Image in virtuelles CD-Laufwerk einlegen**
5. **Netzwerkverbindung der VM abschalten, damit keine Updates gezogen werden**
6. **Erster Windows 10 Installationsschritt:**
   a. Sprache: English
   b. Language: English
   c. Time and Currency: German
   d. Keyboard Layout German
   e. Install-Type: Windows 10 Pro x86 (Date modified: 16.07.2016)
   f. Partition Layout:



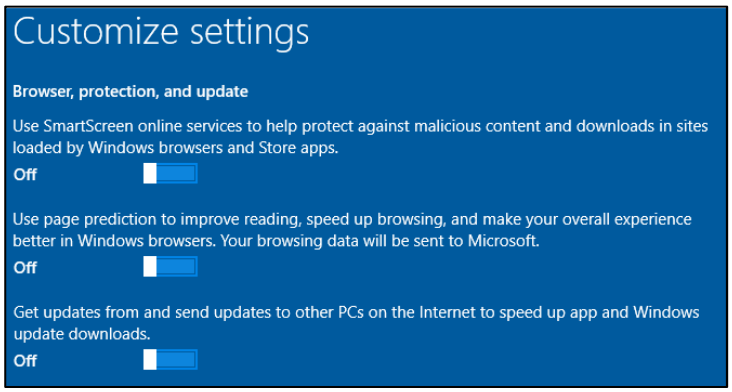
7. **Zweiter Windows 10 Installationsschritt:**
   a. Erste Maske:



   b. Zweite Maske:



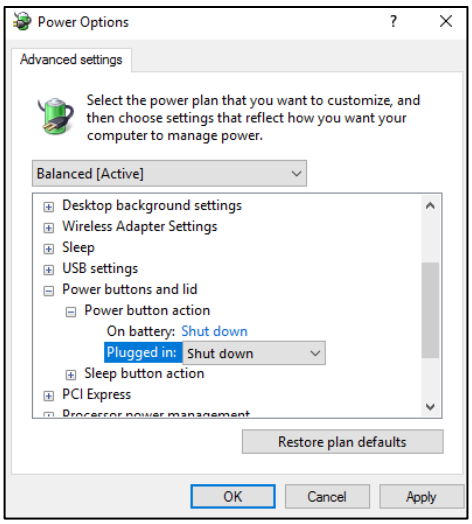
   c. Dritte Maske:



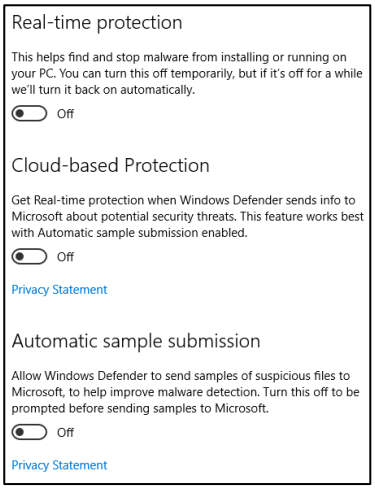
   d. Vierte Maske:
      i. User name: masterthesis
      ii. Password: [keines gesetzt]
      iii. Use Cortana: Nein

B Konfiguration der Virtuellen Maschinen

1. **Setzen der *Power button action*, um ein skriptgesteuertes Herunterfahren der VM zu ermöglichen:**



2. **Deaktivieren des Windows Update Dienstes:**
   a. Angehalten und deaktiviert über Windows Dienste (services.msc: Windows Update, Dienstname: wuauserv)
   b. Zusätzlich über die Group Policies deaktiviert: Unter „Computer Configuration“ > „Administrative Templates“ > „Windows Components“ > „Windows Update“: Klick auf „Configure Automatic Updates“
   c. Auswählen von „Deaktiviert“ in „Configure Automatic Updates“ auf der linken Seite: Klick auf „Übernehmen“ und „OK“
3. **Windows Defender abschalten, um ein Blockieren von bekannten Viren zu verhindern:**
   a. Unter „Settings“ > „Update & Security“ > „Windows Defender“ alle Optionen deaktiviert:



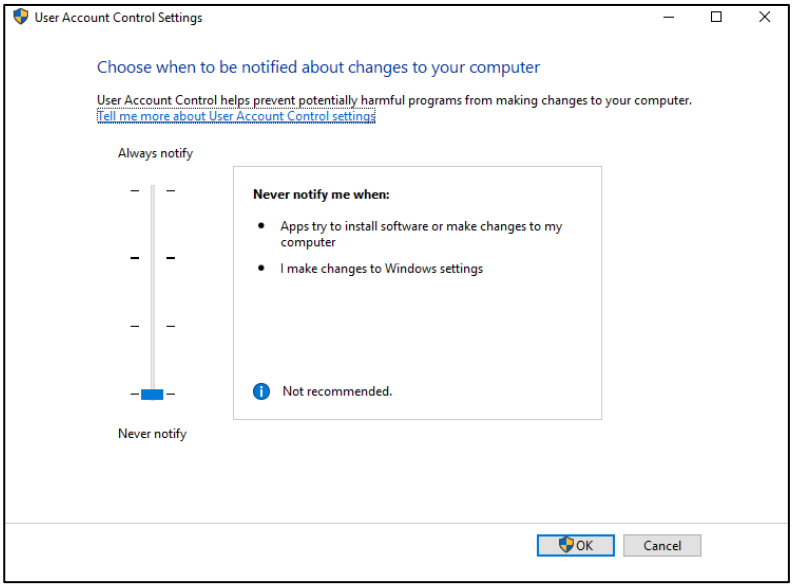
   b. Da Windows den Defender nach einem Neustart wieder einschaltet, wenn kein anderer Virenscanner installiert ist, wurde das Laufwerk C: von der Überwachung durch den Defender exkludiert:
      Unter „Settings“ > „Update & Security“ > „Windows Defender“: Klick auf „Exclude a Folder“: Laufwerk C: auswählen
   c. Zusätzlich über die *Group Policies* deaktiviert:
      Unter „Computer Configuration“ > „Administrative Templates“ > „Windows Components“ > „Windows Defender Antivirus“: „Turn off Windows Defender Antivirus“ auf „Enabled“ gesetzt
   d. Zusätzlich in der Registry deaktiviert:
      HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\Windows Defender: „DisableAntiSpyware“ auf einen Wert von „1“ gesetzt

4. **Windows Search deaktiviert, da sonst bei der Verschlüsselung der Dateien sehr viel CPU-Zeit für den Search-Indexer-Dienst benötigt wird. Dies verlangsamt die VM und die Verschlüsselung:**
   a. Angehalten und deaktiviert über Windows Dienste (services.msc: Windows Search, Dienstname: WSearch)
5. **Autostart-Programme deaktiviert, die den Start der VM verlangsamen:**
   a. Microsoft OneDrive über Task-Manager Reiter „Autostart“ deaktiviert
6. **.Net-Framework 3.5 installiert, da dies von einigen Viren benötigt wird:**
   a. Windows 10 Pro Installationsmedium einlegen
   b. In Powershell ausführen:

```
dism.exe /Online /Enable-Feature /FeatureName:NetFX3 /All
/Source:d:\sources\sxs /LimitAccess
```

   c. Windows 10 Pro Installationsmedium auswerfen und zwei Mal neustarten
7. **Das Remote-Konsole-Session-Passwort wurde auf [keines] gesetzt. Um trotzdem auf die VM per Remote-Konsole zugreifen zu können, müssen folgende Optionen gesetzt werden:**
   a. In den *Group Policies*:
      Unter „Windows Settings“ > „Security Settings“ > „Local Policies“ > „Security Options“: „Accounts: Limit local account use of blank passwords to console logon only“ auf „Disabled“ gesetzt
   b. VM Neustarten
8. **Windows UAC (User Account Control) deaktiviert, um die Sicherheitsabfrage zu vermeiden, wenn skriptgesteuert Programme auf der VM ausgeführt werden:**
   a. In der Registry deaktiviert:
      HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\System: „EnableLUA“ auf einen Wert von „0“ gesetzt

b. In den Systemeinstellungen deaktiviert:



9. **VirtualBox-Gasterweiterungen installiert, um die VM per VirtualBox-API fernzusteuern**
10. **In den Ordneroptionen Dateiendungen eingeblendet:**
    a. Haken bei „Hide Extensions from known files“ entfernt

C Dokumentation Modul vbox_manager

class *vbox_manager.VboxManager*(vm_name)

Wrapper-Class for all things concerning VirtualBox image capturing

- ***capture_image_of_running_vm*(output_filepath)**
  Captures the Guest Core of an running VM
  Expects a path (+filename) to save the GuestCore to, returns TRUE if capturing is successful, FALSE otherwise
- ***capture_program_execution*(vm_snapshot_name, dump_interval, output_filepath, filepath_to_executable_in_vm, program_name)**
  Executes a Program in the VM and captures its execution
  Expects the name of an vm snapshot, the dump interval in seconds, an output filepath, the filepath to the executable of the program in the vm and the name of the program returns True if capturing is successful, False otherwise
- ***execute_program_on_guest*(filepath_to_executable)**
  Executes a specified programm on the currently loaded vm. the vm has to be booted up and running
  Expects the filepath of the executable on the vm
- ***static* prepare_for_analysis(*input_image_path*, *output_image_path*)**
  Cuts a specific part from the guest dump of a VM to get the raw RAM image
  Expects the path to a guest dump image and an output_path, returns a raw memory dump
- **restore_snapshot(*snapshot_name*)**
  Restores the VM to the state of a snapshot
  Expects the name of an existing snapshot for the currently loaded vm
- **resume_vm()**
  Resumes the paused VM
- **start_vm()**
  Starts the currently loaded vm
- **stop_vm_by_force()**
  Powers down the running VM forcefully via Power_Down command
- **stop_vm_via_acpi()**
  Powers down the running VM via ACPI command
- **take_vm_screenshot(*output_filepath*)**
  Takes a screenshot of the currently loaded vm
  Expects a output-filepath to save the image to

D Dokumentation Modul file_writer

***class* file_writer.FileWriter**

This Class contains all methods needed for writing the Files of this Project.

- ***static* add_row_to_csv(*csv_filepath*, *data_array*)**
  Adds a new row to an already existing CSV-File
  Expects the full filepath to the CSV-File and an array containing the content of the new row
- ***static* change_all_arff_attributes_in_folderpath(*attributes_source_arff_filepath*, *target_folderpath*)**
  Changes exchanges the Attributes of all Arff-Files in a directory for the attributes of another Arff-file.
  Expectes an Arff-File to take the Attributes from and a folderpath to search for Arff-Files that will be changed
- **static *change_arff_attributes(* attributes_source_arff_filepath, target_arff_filepath*)***
  Takes two Arff-Files and writes the attribute of the first in the second file.
  Expects two Arff-Files
- ***static* create_fa_ransomware_result_csv(*csv_filepath*, *csv_filename='fa_ransomware_result.csv'*)**
  Creates a CSV-File that is prepared to contain the Results of all Ransomeware Feature Analyser Methods. The First Column contains the Method Names.
  Expects a path to save the CSV-File to and optionally a filename for the CSV-File
- **static *create_time_benchmark_csv(*csv_filepath, csv_filename='timing_benchmark.csv'*)***
  Creates a CSV-File that is prepared to contain the Results of all Timings of the Volatility Framework Functions used. The First Column contains the Function Names.
  Expects a path to save the CSV-File to and optionally a filename for the CSV-File
- ***static* make_experiment3_combined_trainingtestsets(*first_arff_filepath*, *second_arff_filepath*, *output_folderpath*)**
  Create a combined dataset from two datasets the meets the requirements of experiment 3 in Cohen et al. 2018, DOI: 10.1016/j.eswa.2018.02.039
  Expects two arff-file filepaths and an output filepath supplies an arff-file

- ***static* make_experiment3_splits(*first_arff_filepath*, *second_arff_filepath*, *output_folderpath*)**

  Takes two arff-Files, extracts one subject (with all its instances) from the either arff-file. writes both subjects to a single arff-file called testset. the remaining instances of both arff-files are written to a single arff-file called trainingset. this is done for all possible combinations of the subjects.

  Expects two arff-files and an output-folderpath writes all possible take-two–subjects-out combination to arff-files and the produced subject combinations to an Combinations-List.txt

- **static *rolling_combine_arff_files(***
  loaded_arff_files_list, loaded_base_arff_file, output_folderpath*)

  combines two arff-files by leaving out 1 subject from each and making a new arff-file from the rest.

  Expects two arff-files and an output-folderpath writes the result arff-file to the output folderpath

- ***static* save_to_json(*json_filepath*, *content*, *json_filename='new.json'*)**

  Creates a new JSON-File by dumping the supplied content-parameter.

  Expects a filepath to save the JSON-File to, the content to dump into the file and optionally a filename of the JSON-File

- **static *write_experiment_dataframe_to_arff_file(***
  dataframe, output_folderpath, output_filename, relation*)

  Write a Pandas Dataframe to an Arff-File that fulfills the requirements needed for the analysis with Weka afterwards.

  Expects a Pandas Dataframe, an outputpath to write the Arff-File to, the name of the created Arff-File and the realation that will be written in the Arff-File

E Dokumentation Modul volatility_framework_wrapper

***class* volatility_framework_wrapper.VolatilityFrameworkWrapper(*profile*, *filepath*)**

Extract Features from Physical Memory Image

- ***get_callbacks()***
  executes the volatility-callbacks-plugin on the loaded config
- ***get_dlllist()***
  executes the volatility-dlllist-plugin on the loaded config
- ***get_driverscan()***
  executes the volatility-driverscan-plugin on the loaded config
- ***get_extdevicetree()***
  executes the extended volatility-devicetree-plugin on the loaded config
- ***get_extdriverirp()***
  executes the extended volatility-driverirp-plugin on the loaded config
- ***get_handles()***
  executes the volatility-handles-plugin on the loaded config
- ***get_ldrmodules()***
  executes the volatility-ldrmodules-plugin on the loaded config
- ***get_modscan()***
  executes the volatility-modscan-plugin on the loaded config
- ***get_modules()***
  executes the volatility-modules-plugin on the loaded config
- ***get_mutantscan()***
  executes the volatility-mutantscan-plugin on the loaded config
- ***get_privs()***
  executes the volatility-privs-plugin on the loaded config we try to remove the REGEX options before running the privs-plugin to avoid collision with other plugins using the same option
- ***get_psxview()***
  executes the volatility-psxview-plugin on the loaded config
- ***get_ssdt()***
  executes the volatility-ssdt-plugin on the loaded config
- ***get_svcscan()***
  executes the volatility-svcscan-plugin on the loaded config
- ***get_thrdscan()***
  executes the volatility-thredscan-plugin on the loaded config

- ***get_threads()***
  executes the volatility-threads-plugin on the loaded config
- ***get_timers()***
  executes the volatility-timers-plugin on the loaded config if the threads-plugin is executed before on the same config there will be a bug with the volatility-options therefore we try to remove the LISTTAGS options before running the timers-plugin

F Dokumentation Modul memory_image_analysis

***class* memory_image_analysis.MemoryImageAnalyser(**

***volatility_profile*)**

This Class contains all necessary methods to analyse a virtualbox guest dump with the supplied volatility plugins (from VolatilityFrameworkWrapper) and afterwards analyse the result with the FeatureAnalyserRansomware

- ***analyse_all_images_in_directory4(*experiment_folderpath*)***
  Manager-method for analysing all guest dumps in a specified folderpath with get_all_volatility_plugin_outputs_in_parallel (uses 4 workers).
  Expects a folderpath in which the vmem-images are located supplies json-file containing the volatility-plugin-outputs
- ***analyse_all_images_in_directory_seq(*experiment_folderpath*)***
  Manager-method for analysing all guest dumps in a specified folderpath with get_all_volatility_plugin_outputs_sequentially.
  Expects a folderpath in which the vmem-images are located supplies json-file containing the volatility-plugin-outputs
- ***delete_vmem_files_in_folder_recursively(*experiment_folderpath*)***
  Deletes recursively all .vmem-Files in a directory.
  Expects a folderpath to search recursively, returns number of deleted files, false if no .vmem-files were found
- ***extract_ransomware_features_for_all_json_files(*experiment_filepath*)***
  manager-method that takes the folder in which the analysed guest dumps are located analyses the volatility-plugin-jsons and creates the feature vectors of each guest dump by applying get_all_fa_ransomware_outputs_in_parallel to each of them.
  Expects the folderpath in which the Before-Program-Execution and During-Program-Execution folders are located supplies a json and a csv file containing the feature vetors
- ***get_all_fa_ransomware_outputs_in_parallel(*before_program_execution_sample_folderpath, during_program_execution_sample_folderpath_list*)***
  Runs all FeatureAnalyserRansomware-methods in parallel over a list of folders containing the json-outputs of the volatility-plugins of get_all_vola-

tility_plugin_outputs_in_parallel or get_all_volatility_plugin_outputs_sequentially.

Expects the folderpath to the analysed benign image and the list of infected image folders Writes the results of the FeatureAnalyserRansomware-methods to the filepaths of the analysed image

- ***get_all_volatility_plugin_outputs_in_parallel(* image_list, number_of_workers*)***

  Runs all volatility plugins in parallel over a list of guest dump image filepaths.

  Expects the list of image filepaths Writes the results of the volatility plugins to the filepaths of the analysed image

- ***get_all_volatility_plugin_outputs_sequentially(* image_list*)***

  Runs all volatility plugins sequentially over a list of guest dump image filepaths.

  Expects the list of image filepaths Writes the results of the volatility plugins to the filepaths of the analysed image

- **static *get_list_of_json_filepaths_in_folder(* folder_path*)***

  Returns a list of JSON-filepaths for a specific folder (not recursive). Expects a folder path, returns a list of JSON-filepaths

- **get_vmem_filepaths_recursively(*folderpath*)**

  Searches recursively all filepaths to .vmem-Files in a directory. Expects a folderpath to search recursively, returns a dictionary of [folderpath: filename], false if nothing is found

- ***static* save_results_dict(*output_filepath*, *csv_column_length*, *dict_to_save*)**

  writes a dictionary to a csv-file by taking chunks of the dict and outputting each of them to a single column of specified length. Expects an output-filepath, the length of the columns to output to the dict to save to the csv-file supplies a csv-file at the specified output_filepath

***class* memory_image_analysis.FeatureExtractorRansomware-Task(*method_to_call*, *json_infected_sample*, *json_benign_sample*)**

Bases: `object`

This class represents a job-task for running a specific feature-extractor-ransomware-method over a previously analysed guest dump image

***class* memory_image_analysis.VolatilityPluginTask(*method_to_call*, *volatility_profile_to_use*, *image_filepath*, *image_filename*)**

Bases: `object`

This class represents a job-task for running a specific volatility-plugin over a guest dump image

class *memory_image_analysis.Worker(*task_queue, results_dict*)*

Bases: `multiprocessing.process.Process`

This Class is a wrapper for the standard multiprocessing.Process class. It allows to run process-workers over a list of tasks

- ***run*()**

G Dokumentation Modul feature_analyser_ransomware

class *feature_analyser_ransomware.FeatureAnalyserRansomware*

Analyse Features of a Physical Memory Image on Indicators of Compromise for Ransomware (see Cohen et al. 2018, DOI: 10.1016/j.eswa.2018.02.039)

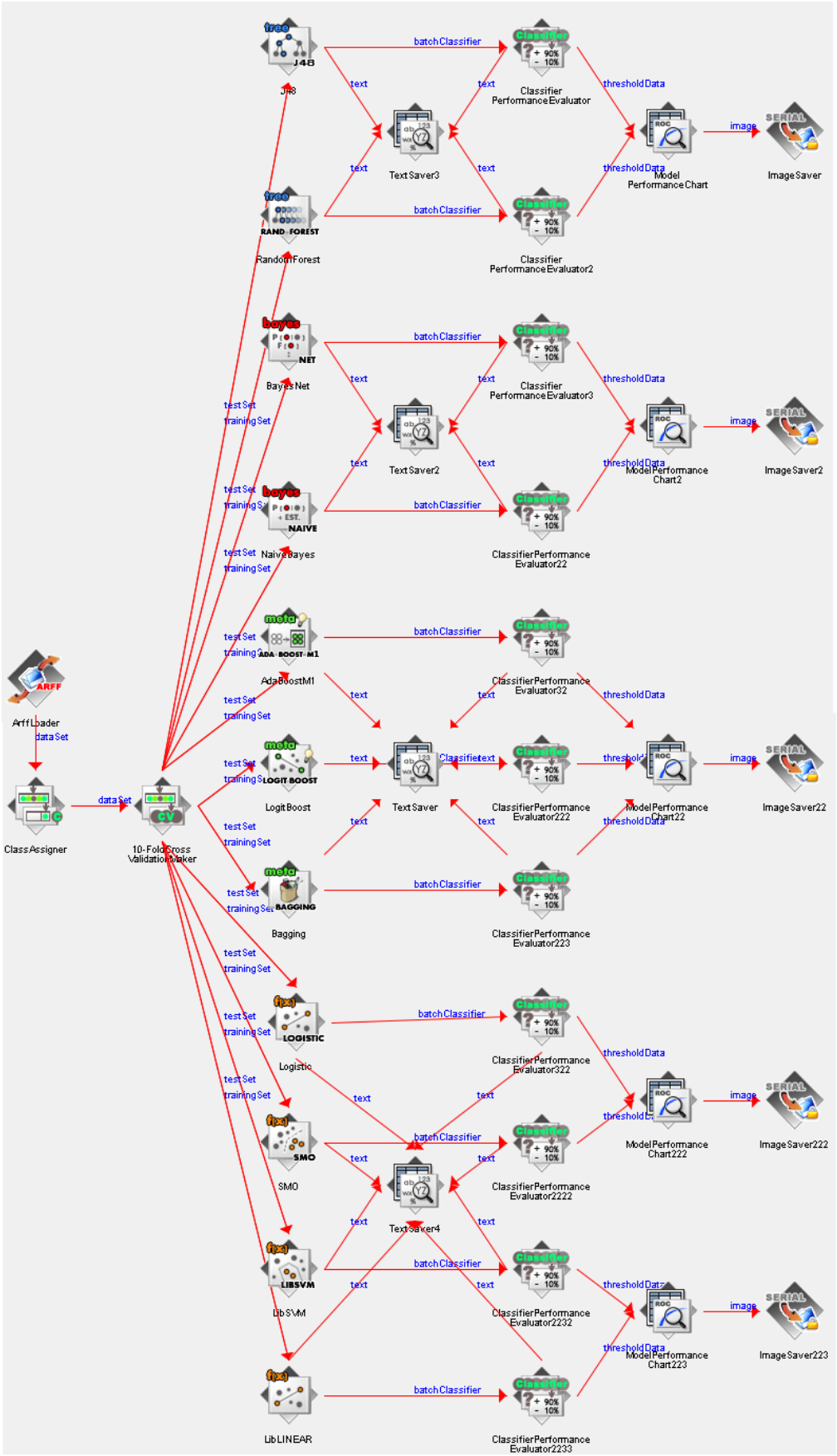
- **static *get_names_of_not_detected_processes_from_psxview(*psxview_json_file_path*)***
  Extracts the Names of the Processes from an PSXView JSON. Expects a Volatility PSXView Plugin JSON Filepath, returns a list of process names
- ***static* get_number_of_callbacks(*callback_json_file_path*)**
  Counts the Number of Callbacks.
  Expects a Volatility Callbacks Plugin JSON Filepath, returns an int
- **static *get_number_of_exited_processes_from_psxview(*psxview_json_file_path*)***
  Counts the Number of exited Processes.
  Expects a Volatility PSXView Plugin JSON Filepath, returns an int
- ***static* get_number_of_false_columns_from_psxview(*psxview_json_file_path*)**
  Counts the Number of Process Listing Techniques that don't detect at least one process.
  Expects a Volatility PSXView Plugin JSON Filepath, returns an int
- ***get_number_of_false_rows_from_psxview(*psxview_json_file_path*)***
  Counts the Number of Processes that aren't detect by at least one Listing Technique.
  Expects a Volatility PSXView Plugin JSON Filepath, returns an int
- ***static* get_number_of_handles(*handles_json_file_path*)**
  Counts the Number of open handels.
  Expects a Volatility Handels Plugin JSON Filepath, returns an int
- **static *get_number_of_ldrmodules(*ldrmodules_json_file_path*)***
  Counts the Number of DLLs.
  Expects a Volatility LDRModules Plugin JSON Filepath, returns an int
- ***static* get_number_of_modules(*modules_json_file_path*)**
  Counts the Number of loaded Modules.
  Expects a Volatility Modules Plugin JSON Filepath, returns an int

- ***static* get_number_of_mutexes_from_mutantscan(*mutantscan_json_file_path*)**
  Counts the Number of Mutexes.
  Expects a Volatility MutantScan Plugin JSON Filepath, returns an int
- **static *get_number_of_not_default_enabled_privs(*privs_json_file_path*)***
  Counts the Number of Privileges that are not Default, but Enabled.
  Expects a Volatility Privs Plugin JSON Filepath, returns an int
- ***static* get_number_of_not_ininit_dll_paths_from_ldrmodules(*ldrmodules_json_file_path*)**
  Counts the Number of DLL Paths not in the InInit List.
  Expects a Volatility LDRModules Plugin JSON Filepath, returns an int
- **static *get_number_of_not_ininitloadmem_dll_paths_from_ldrmodules(*ldrmodules_json_file_path*)***
  Counts the Number of DLL Paths not in the InInit,InLoad and InMem List.
  Expects a Volatility LDRModules Plugin JSON Filepath, returns an int
- ***static* get_number_of_not_inload_dll_paths_from_ldrmodules(*ldrmodules_json_file_path*)**
  Counts the Number of DLL Paths not in the InLoad List.
  Expects a Volatility LDRModules Plugin JSON Filepath, returns an int
- **static *get_number_of_not_inmem_dll_paths_from_ldrmodules(* ldrmodules_json_file_path*)***
  Counts the Number of DLL Paths not in the InMem List.
  Expects a Volatility LDRModules Plugin JSON Filepath, returns an int
- ***static* get_number_of_processes_from_psxview(*psxview_json_file_path*)**
  Counts the Number of Processes.
  Expects a Volatility PSXView Plugin JSON Filepath, returns an int
- **static *get_number_of_pslist_processes_from_psxview(* psxview_json_file_path*)***
  Counts the Number of Processes that are detected by PSList.
  Expects a Volatility PSXView Plugin JSON Filepath, returns an int
- ***static* get_number_of_psscan_processes_from_psxview(*psxview_json_file_path*)**
  Counts the Number of Processes that are detected by PSScan.

Expects a Volatility PSXView Plugin JSON Filepath, returns an int

- **static get_number_of_running_services_from_svcscan(*svcscan_json_file_path*)**
  Counts the Number of running Services.
  Expects a Volatility SVCScan Plugin JSON Filepath, returns an int
- ***static* get_number_of_services_from_svcscan(*svcscan_json_file_path*)**
  Counts the Number of Services.
  Expects a Volatility SVCScan Plugin JSON Filepath, returns an int
- **static *get_number_of_stopped_services_from_svcscan(* svcscan_json_file_path*)***
  Counts the Number of stopped Services.
  Expects a Volatility SVCScan Plugin JSON Filepath, returns an int
- ***static* get_number_of_threads_from_thrdscan(*thrd-scan_json_file_path*)**
  Counts the Number of Threads.
  Expects a Volatility ThrdScan Plugin JSON Filepath, returns an int
- **static *get_number_of_unique_dll_paths_from_dlllist(* dlllist_json_file_path*)***
  Counts the Number of unique loaded DLLs.
  Expects a Volatility DLLList Plugin JSON Filepath, returns an int
- ***static* get_number_of_unique_dll_paths_from_ldrmodules(*ldrmodules_json_file_path*)**
  Counts the Number of unique DLL Paths.
  Expects a Volatility LDRModules Plugin JSON Filepath, returns an int
- ***get_number_of_valid_exceptions_from_psxview(* inspected_psxview_json_file_path, benign_psxview_json_file_path*)***
  Counts the Number of Processes that aren't detect by at least one Listing Technique but are valid exceptions. This is determined by comparing the inspected_psxview_json to the psxview_json of a benign memory image.
  Expects two Volatility PSXView Plugin JSON Filepath (first the inspected, second the benign), returns an int

H Weka KnowledgeFlow – Classifiers - CrossValidation



I Struktur des Datenträgers zu dieser Arbeit

- **Weka Experiment Configuration**
  Enthält alle mit Weka erstellten Experiment- (.exp) und KnowledgeFlow-Dateien (.kf)
- **Trainings- and Testsets:**
  Enthält alle Datensätze, die in dieser Arbeit verwendet wurden
  - **All Samples.xlsx**
    Enthält alle Instanzen, welche im Rahmen dieser Arbeit erstellt wurden, in tabellarischer Form
  - **Experiment 1**
    Enthält alle für die Experimente 1.1 bis 1.3 genutzten Datensätze im ARFF und XLSX-Format
  - **Experiment 2**
    Enthält alle für die Durchführungen des Experiments 2 genutzten Datensätze im ARFF und XLSX-Format
  - **Experiment 3**
  - *Enthält alle für die Durchführungen des Experiments 3 genutzten Datensätze im ARFF-Format*
- **Results:**
  Enthält alle Ergebnisse, welche für diese Arbeit produziert wurden
  - **Experiment 1 - GridSearch - LibSVM.txt**
    Enthält die Ergebnisse der Parametersuche mit GridSearch für den LibSVM-Algorithmus
  - **Results - Machine Learning.xlsx**
    Enthält die zusammengefassten Ergebnisse aller Experimente
  - **Results - Parallel Processing with Python.xlsx**
    Enthält die Ergebnisse der Laufzeitmessungen für die Analyse von 101 Hauptspeicher-Abbildern mit Volatility
  - **Experiment 1.1**
    Enthält die Ergebnisse des Experiments 1.1 nach Klassifizierer-Familie getrennt (textuelle Beschreibung, sowie PNG der ROC)
  - **Experiment 1.2**
    Enthält die Ergebnisse des Experiments 1.2 nach Klassifizierer-Familie getrennt (textuelle Beschreibung, sowie PNG der ROC)
  - **Experiment 1.3**
    Enthält die Ergebnisse des Experiments 1.3 nach Klassifizierer-Familie getrennt (textuelle Beschreibung, sowie PNG der ROC)

    - **Experiment 2**
      Enthält die Ergebnisse des Experiments 2 nach Klassifizierer-Familie getrennt (originale CSV-Dateien, sowie aufbereitete XLSX-Dateien)
    - **Experiment 3**
      Enthält die Ergebnisse des Experiments 3 nach Klassifizierer-Familie getrennt (originale CSV-Dateien, sowie aufbereitete XLSX-Dateien)
- **Python Code Library:**
  Enthält den Quellcode, welcher für diese Arbeit produziert wurde
    - **docs**
      Enthält die Dokumentation des Quellcodes im HTML-Format
    - **extended_devicetree.py**
      Enthält den Quellcode des Modules ExtendedDeviceTree
    - **extended_driverirp.py**
      Enthält den Quellcode des Modules ExtendedDriverIrp
    - **feature_analyser_ransomware.py**
      Enthält den Quellcode des Modules FeatureAnalyserRansomware
    - **feature_analyser_rootkit.py**
      Enthält den Quellcode des Modules FeatureAnalyserRootkit
    - **file_writer.py**
      Enthält den Quellcode des Modules FileWriter
    - **main.py**
      Enthält den Quellcode der main()-Funktion
    - **memory_image_analysis.py**
      Enthält den Quellcode des Modules MemoryImageAnalyser
    - **README.md**
      Enthält den README-Text der Python-Projekts
    - **requirements.txt**
      Enthält die Liste der benötigten Bibliotheken für das Python-Projekt
    - **vbox_manager.py**
      Enthält den Quellcode des Modules VboxManager
    - **volatility_framework_wrapper.py**
      Enthält den Quellcode des Modules VolatilityFrameworkWrapper

Thesen

- Vor dem Hintergrund der Schäden durch Verschlüsselungstrojaner, welche Organisationen weltweit in den letzten Jahren zugefügt wurden, wird deutlich, dass neue Methoden zu deren Detektion benötigt werden.
- Dies gilt insbesondere für virtualisierte Systeme, welche in den letzten Jahren einen enormen Zuwachs an Nutzerzahlen erlebt haben. Diese Systeme haben besondere Sicherheitsanforderungen, bieten aber gleichzeitig auch besondere Möglichkeiten zur Bekämpfung von Trojanern.
- Die Verknüpfung dieser Möglichkeiten mit modernen Methoden aus dem Bereich Data Science bietet das Potential für eine enorme Erhöhung der Sicherheit heutiger IT-Systeme.
- Diese Arbeit knüpft an die bestehende Forschung auf diesem Gebiet an. Die Grundlage dazu bildet eine von Cohen und Nissim vorgestellte Methode zur Erkennung von Verschlüsselungstrojanern in Hauptspeicher-Abbildern virtualisierter Server mittels Maschinellen Lernens.
- Die Methode wird repliziert und auf Basis eines veränderten Hypervisors und Betriebssystem angewandt. Ihre Leistungsfähigkeit mit einem neuen und erweiterten Satz von jeweils zehn Verschlüsselungstrojanern und gutartigen Programmen in dieser veränderten Umgebung umzugehen, wird erprobt. Es wird versucht die Detektionsleistung mit der Einführung zweier weiterer Algorithmen des Maschinellen Lernens zu verbessern.
- Die Ergebnisse zeigen, dass die Methode den hier gemachten Veränderungen gewachsen ist.
- Die Detektionsleistung ist bei bekannten und sogar bei unbekannten Verschlüsselungstrojanern sehr gut.
- Die Analysegeschwindigkeit der Hauptspeicher-Abbilder mit Volatility kann darüber hinaus durch parallelisierte Bearbeitung wesentlich verbessert werden.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die hier vorliegende Arbeit selbstständig, ohne unerlaubte fremde Hilfe und nur unter Verwendung der in der Arbeit aufgeführten Hilfsmittel angefertigt habe.

_____

Ort, Datum Christian Haupt