HAW HAMBURG

Bachelorarbeit

Tobias Schulz

Merkmalslose Malware-Erkennung durch dynamische Faltungsnetze und Multi-Target-Learning

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

Tobias Schulz

Merkmalslose Malware-Erkennung durch dynamische Faltungsnetze und Multi-Target-Learning

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Angewandte Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Michael Neitzke
Zweitgutachter: Prof. Dr. Peer Stelldinger

Eingereicht am: 12. Februar 2024

Tobias Schulz

Thema der Arbeit

Merkmalslose Malware-Erkennung durch dynamische Faltungsnetze und Multi-Target-Learning

Stichworte

Faltungsnetz, dynamisches Faltungsschicht, Lernen mehrerer Ziele, Malware-Erkennung

Kurzzusammenfassung

Diese Arbeit untersucht die Eignung von dynamischen Faltungsschichten und das Multi-Target-Learning zur Steigerung der Sensitivität bei geringer Falsch-Positiv-Rate und einer größeren Erklärbarkeit oder Informationsgewinnung bei der merkmalslosen Malware-Erkennung. Als zweites Ziels, welches als Nebenziel angesehen wurde, wurde dabei exemplarisch eine Multi-Label-Klassifizierung von 126 Verhaltensbeschreibungen verwendet. Damit diese Techniken nicht eingeschränkt sind, stellt die Arbeit eine leicht modifizierte Art von der klassischen Mnemonic-Sequenz-Augmentation und eine neue Vorverarbeitung mit DLL-Funktionsauflösung vor. In den Untersuchungen mit Hilfe von 5 Basismodellen zeigt sich, dass beide Techniken in den meisten Fällen zu einer teils starken Steigerung der Sensitivität führen. Gerade das Multi-Target-Learning kann aufgrund der Tatsache, dass über diese Technik auch zeitgleich weitere Informationen zur Eingabe erlangt werden können, bei der Entwicklung von künstlicher Intelligenz gestützter Erkennung von zuvor unbekannter Malware helfen. Je nach zweites Ziel erzielt jedoch ein zweites Modell, ohne der Klassifizierung zwischen Malware und gutartiger Software jedoch gegebenenfalls bessere Ergebnisse für das zweite Ziel.

Tobias Schulz

Title of Thesis

Featureless malware detection using dynamic convolutional networks and multi-target learning

Keywords

CNN, dynamic convolution, Multi-Target-Learning, Malware detection

Abstract

This Thesis examines the suitability of dynamic convolutional layers and multi-target learning to increase sensitivity with a low false-positive-rate and to achieve greater explainability or information gain in featureless malware detection. As a secondary target, which is considered as a side objective, a multi-label classification of 126 behavioral descriptions was exemplarily used.

To ensure that these techniques are not restricted, the a slightly modified version of the classical mnemonic sequence augmentation and a new preprocessing technique involving DLL function resolution is introduced. In the investigations using five base models, it is observed that both techniques, in most cases, lead to a significant increase of the sensitivity of the models.

Especially multi-target learning, due to the fact that it allows simultaneous acquisition of additional information about the input, can assist in the development of artificial intelligence supported detection of previously unknown malware. However, depending on the secondary goal, a second model achieves potentially better results for the secondary goal without distinguishing between malware and benign software.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

API Programmierschnittstelle.

AUC Fläche unter der Kurve (eng: area under the curve).

CNN faltendes neuronales Netzwerk (eng: convolutional neural network).

DepConv tiefenweise Faltungsschicht (eng: depthwise convolution).

DLL Dynamic Link Library.

DynConv dynamische Faltungsschicht (eng: dynamic convolution).

E2E Ende-zu-Ende.

FNR Falsch-Negativ-Rate.

FPR Falsch-Positiv-Rate.

LSH Locality Sensitive Hashing.

MACs Multiplizieren-Akkumulieren Berechnungen (eng: multiply-accumulate computations).

ML maschinelles Lernen.

MTL Lernen mehrerer Ziele (eng: multi-target learning oder multi-task learning).

PE Portable Executable.

RNN Rekurrentes neuronales Netz.

SepConv tiefenweise Faltungsschicht mit anschließender punktueller Faltungsschicht (eng: depthwise seperable convolution).

TLSH Locality Sensitive Hash von TREND MICRO.

TPR Richtig-Positiv-Rate (eng: true positive rate).

1 Einleitung

1.1 Motivation

Im Jahr 2022 stieg die Anzahl der weltweiten bekannten Malwarebefälle um etwa 2%, und betrug damit um die 5,5 Milliarden [79, S. 22], nachdem sie 2021 um 4% auf 5,4 Milliarden gefallen war [78, S. 22]. Dass die Zahl immer noch so hoch ist, liegt unter anderem daran, dass es relativ schwierig ist, Malware zu entdecken. Dies ist insbesondere dann der Fall, wenn die Malware von bereits bekannter Schadsoftware stark abweicht [35, S. 1]. Ein klassischer Abgleich mit bekannten Signaturen reicht längst nicht mehr aus [35, S.1], auch da weltweit monatlich über 400.000 neue Malware entdeckt werden [78, S. 19]. Es gibt zwar spezielle Systeme wie Honeypots, welche extra dafür ausgelegt sind, von Hackern angegriffen und mit Malware befallen zu werden, um diese zu entdecken und zu analysieren [19], doch häufig sind normale Systeme das erste Opfer neuer Malware. Grund hierfür ist neben der verhältnismäßig geringen Anzahl an Honeypots im Vergleich zu anderen Internetnutzern auch, dass Malware nicht nur willkürlich verbreitet wird, sondern teils auch gezielt gegen bestimmte Ziele verwendet wird[vgl. z.B. 78, S. 32]. Um eben auch diese „never before seen attacks“[78, S. 52] zu erkennen, sind weitere Methoden notwendig, wobei gerade Ansätze des maschinellen Lernens (ML) hier vielversprechend scheinen [21]. Dies liegt vor allem daran, dass diese ohne klar definierte Regeln auskommen und so auch Unbekanntes klassifizieren können [73, S. 17, 78, S. 51f]. Fleshman, Raff u. a. konnten zudem experimentell darlegen, dass ML-Ansätze deutlich immuner als herkömmliche Virenscanner gegenüber Veränderungen im Code sind [21, S. 5]. Weitere Forschung auf diesem Gebiet kann also zu einer wirkungsvolleren Erkennung von Malware führen, sowohl, wenn diese für die Erkennung verwendet werden, als auch dadurch, dass durch sie neue Charakteristiken/Merkmale von Malware entdeckt werden können.

1.2 Zielsetzung

Es gibt sehr viele verschiedene Arten ML für die Erkennung von Malware einzusetzen. Dabei unterscheiden sich die Ansätze nicht nur darin, welche ML Methode angewendet wird, sondern auch darin, welche Daten mit welcher Vorverarbeitung verwendet werden. Einer der weniger gut erforschten Ansätze ist die Malware-Erkennung über ein merkmalsloses neuronales Netzwerk-Modell [3, S. 2410].Merkmalslos bedeutet hier, dass das ML Modell keine von Hand ausgewählten Merkmale, sondern direkt das zu bewertende Objekt als Eingabe erhält. Die Repräsentationsweise darf dabei aber abgewandelt werden. Konkret bedeutet dies hier, dass beispielsweise direkt die Bytes in der Form, wie sie in der ausführbaren Datei vorhanden sind, als Eingabe verwendet werden. Wird dabei das zu bewertende Objekt vollständig und (fast) ohne Vorverarbeitung als Eingabe verwendet, so wird dies auch als Ende-zu-Ende (E2E)-Lernen bezeichnet [3, S. 2410, 47, S. 1, 69]. Dies hat den Vorteil, dass die Modelle nicht durch möglicherweise Voreingenommenheit bei der Erstellung der Features/Merkmale beschränkt sind und so neben der Erkennung auch weitere Erkenntnisse für die Malware-Erkennung liefern können [67, S. 3, 49, S. 3].

Statt die Bytes direkt in das Netz zu geben, ist eine leicht abgewandelte Version, dass der Code vorher disassembliert wird und dann die Anweisungen mit oder ohne Operanden als Eingabe verwendet werden [siehe dazu z.B. 45, S. 3]. Eine Anweisung ohne Operanden wird dabei auch als Befehl oder Anweisung (eng: Mnemonic) und dessen numerische Darstellung als Opcode bezeichnet [35, S. 6]. Hier handelt es sich genau genommen nicht ganz um E2E-Lernen, da sowohl etwas Vorverarbeitung vorgenommen wurde, als auch nicht die gesamte Datei verarbeitet wird. Da jedoch keine Merkmale definiert werden und zudem merkmalslos oft mit E2E gleichgesetzt wird, wird auch dies häufig als E2E bezeichnet [47, 35].

Besonders was die Einbindung neuer beziehungsweise weiterer Techniken betrifft, gibt es bei der Malware-Erkennung über merkmalslose neuronale Netze noch einiges zu erforschen. Beispiele für diese Techniken sind das Lernen mehrerer Ziele (eng: multi-target learning oder multi-task learning) (MTL)[1] [10, 70] und neuartige Schichten/Modellstrukturen

[1] Ob das Lernen mehrerer Ziele mit der Definition von merkmalslos vereinbar ist, ist nicht ganz klar, da in gewissen Maßen die Merkmale über das Ziel dem Netz aufgezwungen werden. Spätestens jedoch, wenn wie in [70] die Malware- oder-gutartig-Software-Klassifizierung beim Lernen mehr gewichtet wird oder das zweite Ziel nicht korrelierend mit dem ersten Ziel ist, kann argumentiert werden, dass es sich immer noch um ein merkmalsloses Netz handelt. Die Eingabe des Netzes ist weiterhin nicht großartig vorverarbeitet und das Netz hat die Möglichkeit intern weitere Eigenschaften zu repräsentieren, da

wie dynamische Faltungsschichten (eng: dynamic convolutions) (DynConvs) [12]. Bei MTL ist die Hoffnung, dass dies dem Netz zu einer tiefer greifenden Erkennung zwingt, als über N-Gramm-Analysen möglich ist. Untersucht wird dies exemplarisch mit Label, die das Verhalten von Malware beschreiben, wie zum Beispiel, dass eine HTTP(s)-Verbindung aufgebaut wird, Dateien gelöscht werden oder das Vorhandensein eines Virenscanners geprüft wird. Im Folgenden werden diese Label Verhaltenslabel genannt. DynConv wiederum stellt eine interessante Alternative zu den Ansätzen mit Transformer wie beispielsweise I-MAD von Li, Fung u. a. [56] dar, welche zwar das Potenzial von Attention vermutlich besser ausnutzen, jedoch eine vordefinierte Längenbeschränkung aufweisen und deutlich mehr Rechenleistung benötigen als Faltungsschichten [66]. Eine alternative Lösung, bei der es ebenfalls keine vorher festgelegte Längenbegrenzung gibt, jedoch annähernd die volle Stärke von Transformer vorhanden bleibt, ist die Verwendung von Block-Recurrent Transformers [42]. Diese können zwar deutlich effizienter als herkömmliche Transformer sein, kommen jedoch dennoch bei Weitem nicht an die Effizienz von Faltungsnetzen heran [42, S. 7].

Das Ziel dieser Arbeit ist es daher, neue merkmalslose Modelle für die Erkennung von Malware im Portable Executable (PE)-Format mit den Techniken MTL und DynConvs zu entwerfen, zu untersuchen und mit aussagekräftigen Referenz-/Basismodellen ohne diese Techniken zu vergleichen. Dabei soll beantwortet werden, inwiefern die Verwendung von MTL und DynConvs in der merkmalslosen Malware-Erkennung über neuronale Netze geeignet ist, um zuverlässiger zwischen Malware und gutartiger Software zu unterscheiden und um zusätzliche Erkenntnisse über die dem Modell übergebene PEs zu erlangen. Über eine Literaturrecherche wird dafür eine geeignete merkmalslose Darstellung einer PE ermittelt, bei welcher auch die begrenzte Ressourcenverfügbarkeit von Relevanz ist, da die Eingaben selbst bei kleinen PEs relativ groß sind, wodurch für die Backpropagation viel Speicher benötigt wird. Zur Nachvollziehbarkeit der Entscheidungen bei der Herausarbeitung des Eingabeformates und eines geeigneten Basismodells wird zuerst in Abschnitt 3 auf die Grundlagen von PEs, Malware und auf die in dieser Arbeit wichtigsten Aspekte von faltendes neuronales Netzwerk (eng: convolutional neural network) (CNN) eingegangen. Nachdem in Abschnitt 4.1.1 das geeignete Eingabeformat begründet wird, wird in Abschnitt 4.1.2 ein geeignetes Referenzmodell beschrieben, auf welchem dann nach den

es anders als bei klassischen, auf Merkmalsvektoren basierenden Netzen nicht auf diese Merkmale begrenzt ist. Die größere Gewichtung des Hauptzieles wird in dieser Arbeit angewendet, doch wie in 4.2 beschrieben, korreliert nur ein kleiner Teil der Label des zweiten Zieles nicht direkt mit der Malware-Erkennung. Unter diesen Umständen werden in dieser Arbeit auch die Netze mit dem in dieser Arbeit angewendeten MTL als merkmalslos angesehen.

optimalen Bedingungen von DynConvs und weiteren Erkenntnissen weitere Basismodelle entwickelt werden. Im darauffolgenden Abschnitt werden dann die möglichst geeigneten und für die untersuchten Techniken nicht zu restriktiven Basismodelle beschrieben, bevor in Abschnitt 4.1.4 die Einbindungen der untersuchten Techniken in diesen Modellen dargestellt werden. Im Abschnitt 4.2 wird dann der für die Untersuchungen verwendete Datensatz und die Sicherstellung dessen Qualität beschrieben. Die Ergebnisse werden schlussendlich in Abschnitt 6 aufgezeigt, analysiert und diskutiert, wobei die dafür durchgeführten Untersuchungen und Modelltrainings in Abschnitt 5 beschrieben werden.

Betrachtet werden in dieser Arbeit nur PEs, bei denen die eigentliche Programmlogik als nativer Code ausgeführt wird und nicht in einer virtuellen Maschine beziehungsweise einem Laufzeitsystem laufen. Auch Programme, die erst zur Laufzeit kompiliert werden, werden in dieser Arbeit nicht betrachtet. Beispielsweise werden also Python-Code-Programme, Anwendungen, die in der JVM laufen, oder die Common Intermediate Language ausführen in dieser Arbeit nicht betrachtet. Des Weiteren wird nicht die gesamte Datei als Entscheidungsgrundlage verwendet, sondern nur der Teil, der den Programmcode enthält. Schlussendlich erweisen sich sowohl Header als auch der Maschinencode und andere Abschnitte der Datei als geeignet, doch wenn diese nicht separiert betrachtet werden, fokussieren sich viele Modelle vor allem auf den Header, da dieser häufig aussagekräftige Zeichenketten enthält, welche für das Modell einfach zu erkennen sind [35].Da der Programmcode schlussendlich aber das Entscheidende ist und zu diesem zudem weniger Forschung besteht, beschränkt sich diese Arbeit auf den enthaltenen kompilierten Programmcode.

In der Forschung (z.B. [55, 34]) wird eine Sequenz von Anweisungen häufig auch als Opcode-Sequenz bezeichnet, auch wenn beispielsweise alle Opcodes der Aufrufanweisungen `call` (*e8*, *ff* und *9a*) als das gleiche betrachtet werden [43, S. 735], also eigentlich eine Mnemonic-Sequenz und keine Opcode-Sequenz betrachtet wird. Wenn die einzelnen Opcodes einer Anweisung zusammengefasst werden, dann werden in dieser Arbeit die Begriffe Anweisungssequenz oder Mnemonic-Sequenz verwendet. Eine Opcode-Sequenz ist in dieser Arbeit tatsächlich die rohe Opcode-Sequenz, also die Bytefolge ohne unter anderem die Bytes, welche die Kodierung der Argumente oder die tatsächlichen Argumente darstellen.

2 Verwandte Arbeiten

In den letzten Jahren gab es zu dem Thema, wie mit Hilfe von neuronalen Netzen Malware erkannt oder klassifiziert werden kann, viel Forschung und neue Erkenntnisse. Bezogen auf die Verarbeitung von den reinen Bytes einer Datei zur Erkennung von Malware sind die wohl relevantesten Arbeiten die von Raff, Barker u. a. [67] und Krčál, Švec u. a. [49].

Raff, Barker u. a. zeigen in „Malware Detection by Eating a Whole EXE“, dass bereits ein relativ flaches Faltungsnetz in der Lage ist, mit einer relativ hohen Genauigkeit Malware zu erkennen. So weist deren MalConv Modell, das nur 134.632 trainierbare Parameter hat, nach einem Training mit 2 Millionen Dateien bei einer Testdatensatzgröße von 400.000 eine Genauigkeit von 90,9 % auf. Sowohl eine größere Anzahl an Faltungsschichten, als auch der Austausch des Max-Poolings oder der Faltungsschicht durch eine Rekurrentes neuronales Netz (RNN)-Schicht[1] führen zu einer schlechteren Genauigkeit. Das von ihnen entwickelte Modell mit mehr Faltungsschichten weist zudem erst bei einem deutlichen Overfitting eine hohe Trainingsgenauigkeit auf. Eine Strategie, die erstmalig von Nataraj, Karthikeyan u. a. [60] angewendet wurde, ist die Darstellung von Malware als Graustufenbilder. Dafür wird jeder Byte eines Programms in einen Graustufenpixel umgewandelt und durch eine „arbitrary ‚image width‘“[67, S. 9] die PE in ein zweidimensionales Bild umgewandelt. Dadurch entsteht jedoch ein neuer Hyperparameter, für welchen es wohl keinen passenden Wert gibt, da zwangsweise nicht vorhandene Lücken und räumliche Zusammenhänge entstehen. Die von Raff, Barker u. a. erstellten Modelle mit dieser Technik erzielen, wie von ihnen erwartet, schlechtere Ergebnisse. [67]

Krčál, Švec u. a. [49] schlagen in „Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only“ ein tieferes Modell mit 4 Faltungsschichten und anschließenden 4 vollständig verbundenen Schichten vor, mit welchem sie mit ihren 20 Millionen Trainingsdaten eine Genauigkeit von 96 % erreichten. Weitere relevante Erkenntnisse im Vergleich zu [67] sind vor allem, dass die Einbettungswerte nicht zu groß sein dürfen und dass die Schrittweite bei Faltungsschichten gerade oder am besten

[1] RNN ist hier als Klasse gemeint und meint nicht nur das von [71] mitentwickelte Standard RNN.

eine Zweierpotenz sein sollte[2]. Sie sind der Meinung, dass Deep Learning in diesem Bereich vor allem in Zukunft dafür benutzt werden kann, um zuvor nicht gesehene Muster aufzudecken. Trotz der Tatsache, dass sie alle verwendeten Modelle mit 20 Millionen ausführbaren Dateien getestet haben, weist deren Modell mit 96 % eine höhere Genauigkeit als das von ihnen trainierte MalConv Modell auf. Letzteres erzielt eine Genauigkeit von 94,6 %. [49, S. 2f]

In beiden dieser Arbeiten wird dabei festgehalten, dass die Netze ihre Entscheidung vor allem auf Informationen aus dem Header der ausführbaren Datei treffen, wobei beim MalConv Modell dieser Anteil auf 58 % bis 61 % beziffert wird [67, S. 6, 49, S. 3]. Beide stellten zudem fest, dass Batch-Normalisierung, also eine Normalisierung der Aktivierungsvektoren der verdeckten Schichten vor der Übergabe an die nächste Schicht, (in allen getesteten Modellen) zu einem schlechteren Ergebnis führt [67, S. 6f, 50]. Daher wird auch in dieser Arbeit auf Batch-Normalisierung verzichtet.

In vielen Arbeiten wie zum Beispiel [33, 74, 34, 45] wird jedoch gezeigt, dass ein künstliches neuronales Netz relativ gut eine Malware anhand nur des Opcodes—also ohne Header-Informationen—erkennen kann. So erreicht die Mnemonic-basierte Komponente aus [34] bei einer Malware-Familien-Klassifikation eine Genauigkeit von 99,17 %. Das CNN Modell von Sharma, Malacaria u. a. [74] erreicht bei der Malware-Erkennung eine Wahrscheinlichkeit von 99,2 %. In [34] wird dabei ein flaches Netz mit verschiedenen Filtergrößen verwendet und in [74] ein tiefes Netz aus 6 Schichten mit jeweils einer Filtergröße von 3 [74, S. 9, 34, S. 6]. Li, Fung u. a. [56] stellte zudem fest, dass dieser zumindest bei getrennten Teilnetzen auch in Kombination mit vielen weiteren Informationen wie Header-Informationen einen großen Beitrag für die Erkennung liefern kann.

Einen genaueren Überblick über Teilbereiche des aktuellen Forschungsstandes inklusive der in dieser Arbeit nicht betrachteten Ansätze kann unter anderem auch [35] entnommen werden.

Bezogen auf das MTL, was im Grunde eine Multi-Label-Klassifizierung ist, bei welcher die einzelnen Label einem zumindest etwas anderem Problem zugehörig sind [10], ist die hier am relevantesten Arbeit die von Rudd, Ducau u. a. [70]. Rudd, Ducau u. a. erkannten, dass auch bei der Malware-Erkennung MTL zu einer besseren Genauigkeit führt,

[2] Beschrieben wird ihre Erkenntnis mit „Power of two“[49, S. 2], doch die Vergleichswerte der Strides, die keine Zweierpotenz sind, sind zudem auch ungerade [49, S. 2]. Die Zweierpotenz wird nur darin begründet, dass alle Compiler die Abschnitte der ausführbaren Datei zu einem Vielfachen einer Zweierpotenz ausrichten[49, S. 2], was jedoch nicht wirklich für eine Schrittweite einer Zweierpotenz spricht.

beziehungsweise führen kann. Konkret wurde bei den Experimenten das beste Ergebnis erreicht, wenn alle vier Ziele gemeinsam gelernt wurden. Die einzelnen Ziele sind dabei: 1) Ist die PE eine Malware; 2) Anzahl der Antivirenhersteller, die die PE als Malware klassifiziert haben; 3) welche Hersteller die PE als Malware klassifiziert haben und 4) eine Multi-Label-Klassifizierung für die Art der Malware. Weiter konnte festgestellt werden, dass die Qualität des Modells hinsichtlich des Hauptzieles nur dann steigt, wenn das zweite Ziel hilfreiche Informationen für das Hauptziel darlegt. Dabei konnte gezeigt werden, dass MTL keine gegen Overfitting regulierende Wirkung hat—zumindest nicht, solange kein starkes Overfitting auftritt. Dadurch, dass die Malware-Erkennung in der Verlustfunktion eine höhere Gewichtung hat, nutzen sie zudem das Prinzip des Hauptzieles, welches in dieser Arbeit übernommen wird. [70]

Huang und Stokes [41] behandelten ebenfalls das MTL bei der Malware-Erkennung, und stellten ebenfalls eine Verbesserung durch diese Technik fest. Sie stellten fest, dass ein auf einen Merkmalsvektor basierendes Modell besser Malware erkennen kann, wenn es zusätzlich die Malwarefamilie bestimmen soll. Verwendet wurden dafür zwei getrennte Softmax-Schichten, wobei die erste Schicht eine binäre Klassifizierung zwischen Malware und nicht Malware vornimmt und die zweite Schicht zwischen 98 Malwarefamilien, den restlichen Familien und gutartiger Software unterscheidet. [41] Die binäre Klassifizierung ist damit prinzipiell redundant zur hundertklassigen Klassifizierung, was erklären könnte, warum die Verbesserung geringer als beim Modell von Rudd, Ducau u. a. ist [70, S. 310, 41, S. 12].

Neu in dieser Arbeit bezogen auf das MTL ist damit, dass, wie in Abschnitt 4.2.3 beschrieben ist, ein größerer Labelraum beim zweiten Ziel verwendet wird, und dass MTL in dieser Arbeit auf eine merkmalslose Modelleingabe angewendet wird. Des Weiteren wird in dieser Arbeit weiter untersucht, ob die Modelle mit beziehungsweise ohne MTL etwas anderes gelernt haben. Smith, Johnson u. a. [77] untersuchten zudem bereits, ob merkmalslose Modelle in der Lage sind, Verhaltenslabel einer Malware zuzuordnen, jedoch verwendeten sie dafür nur die ersten 1024 Bytes der einzelnen Malware, welche sie in ein Graustufenbild konvertiert haben [77, S. 57f]. Dass dadurch kein zufriedenstellendes Ergebnis erzielt werden konnte, verwundert damit nicht wirklich, da der Header, welcher am Anfang des Programms steht, nicht genug Informationen über das Verhalten einer PE aussagt. Unterscheiden tut sich zu dieser Arbeit auch, dass die Verhaltenslabel als einziges Ziel und nicht im Rahmen eines MTL verwendet wurden.

3 Grundlagen

3.1 Portable Executable

Die in dieser Arbeit betrachteten Dateien sind PE—also unter Windows ausführbare Dateien. Das Format von PE wird in *PE-Format - Win32 apps* [46] beschrieben. Sie bestehen aus einem MS-DOS-Stub, einer Signatur, einem COFF-Dateiheader, einem optionalen Header, einer Abschnittstabelle und verschiedenen darauf folgenden Abschnitten [46]. Im COFF-Dateiheader wird an erster Stelle die Architektur angegeben, für welche die PE erstellt wurde. Von den dreißig von Microsoft aufgeführten Architekturen können auf einem Rechner mit der AMD64 Architektur unter Windows 10 nur PE für die AMD64 (x64/x86-64) und I386 (x86) Architektur ausgeführt werden. Die Abschnitte haben verschiedene Eigenschaften, die auch deren Verwendung beschreiben. So kann in einem Abschnitt beispielsweise Code, vom Programm verwendete Daten, Debuginformationen oder Verweise auf externe oder von dieser PE angebotene Methoden enthalten sein. Der in dieser Arbeit betrachtete Code ist üblicherweise in dem Abschnitt mit dem Namen *.text* enthalten, doch dieser Abschnitt kann auch anders heißen und auch andere Daten wie Debuginformationen enthalten. Zudem kann auch in mehreren Abschnitten ausführbarer Code enthalten sein. Ausschlaggebend sind schlussendlich die Eigenschaften *IMAGE_SCN_CNT_CODE* und *IMAGE_SCN_MEM_EXECUTE*. [46].

3.2 Malware

3.2.1 Definition von Malware

Das National Institute of Standards and Technology (NIST) definiert Malware als „software or firmware intended to perform an unauthorized process that will have adverse impacts on the confidentiality, integrity, or availability of a system. A virus, worm, Trojan horse, or other code-based entity that infects a host. Spyware and some forms of

adware are also examples of malicious code“ [23, S. 407][1]. Das Bundesamt für Sicherheit in der Informationstechnik definiert Malware noch etwas breiter als eine „Software, die mit dem Ziel entwickelt wurde, unerwünschte und meist schädliche Funktionen auf einem IT-System auszuführen“ [29]. Unter letzterem fallen beispielsweise auch alle Adware, wobei bei „unerwünscht“ viel Interpretationsraum bleibt.

Der Einfachheit halber wird in dieser Arbeit anders als beispielsweise in einer der ersten Definitionen eines Virus in Cohen [13] nicht unterschieden, ob der schädliche Code jemals ausgeführt werden kann oder nicht. Konkret bedeutet dies, dass in dieser Arbeit auch etwas als Malware angesehen wird, wenn der schadhafte Code nicht ausgeführt wird, weil sich das Programm beispielsweise vorher beendet, der vorherige Abschnitt des Programms möglicherweise endlos läuft oder die schadhafte Funktionalität innerhalb des Programmcodes nie adressiert wird. Für die späteren Experimente ist dies wichtig, da es hin und wieder vorkommt, dass Malware eine feste oder dynamische Ablaufzeit haben, wenn beispielsweise der Server, mit denen sie kommunizieren, nicht mehr erreichbar ist. So wurde beispielsweise bei den Trainingsdaten festgestellt, dass sich mehrere Programme nach dem Überprüfen der Zeit selbst beendet haben (siehe Seite 46)[2]. Diese Dateien sind damit in der Theorie unter bestimmten Bedingungen zwar noch immer aktiv, und mit dieser Änderung besteht hier keine Unklarheit, ob sie als Malware anzusehen sind.

Im Rahmen dieser Arbeit wird jedoch nicht selbst entschieden, ob es sich bei einer Datei um Malware handelt oder nicht. Stattdessen wird sich darauf verlassen, dass alle im verwendeten Malware-Datensatz enthaltenen Dateien Malware sind.

Im Folgenden werden die gängigsten Arten von Malware weiter beschrieben, wobei Malware häufig nicht nur zu einer Art gehört, sondern meist eine Mischform verschiedener Arten sind:

Virus: Viren sind Malware, die sich an andere Programme oder Systeme anhängen und beim Aufruf dieser sich replizieren können. Dadurch können sie sich innerhalb eines System und oder auf weitere Geräte ausbreiten. [32]

Wurm: Würmer unterscheiden sich zu Viren darin, dass sie eigene ausführbare Programme sind. Auch sie können sich auf einem Gerät oder über Geräte hinweg ausbreiten. [32]

[1] In [22, S. B-13] wird malicious code mit Malware gleichgesetzt. Andere gängige deutsche Bezeichnungen für Malware sind Schadprogramm und Schadsoftware.

[2] Zur weiteren Sicherstellung, dass diese jedoch schädlich waren, wurden in den Trainingsdaten nur die mit aufgenommen, bei welchen trotzdem Malware-Verhalten erkannt wurde.

Trojaner: Trojanische Pferde, welche auch Trojaner genannt werden, zeichnen sich dadurch aus, dass sie sich als ein nicht schädliches oder gar „nützliches“ [31] Programm ausgeben [31]. Je nachdem, was ein Trojaner tut, kann es sehr schwierig sein diese zu erkennen. Wenn sie beispielsweise als Dropper fungieren, also weitere Malware auf dem System ablegen, dann ist es nicht immer einfach, den Ursprung auf diesen Trojaner zurückzuführen. Da sie zudem zumindest relativ ähnlich mit gutartiger Software sind, können sie einfach bei vielen Heuristiken unentdeckt bleiben. Schwieriger oder gar unmöglich ist es dann zudem, wenn die gefährliche Aktivität nicht von einer normalen Aktivität unterscheidbar ist oder der eigentliche Schaden auf einem Backendserver geschieht. So stelle man sich einen Passwortmanager vor, welcher die Daten verschlüsselt auf einem Backendserver speichert, doch die Daten so verschlüsselt/kodiert, dass diese auf dem Backend von Hersteller oder Betreiber wieder entschlüsselt/dekodiert werden können. Ein anderes Beispiel ist hier ein als Chatclient getarnter Trojaner, der ebenfalls neben der eigentlichen Funktionalität das Passwort dem Hersteller zukommen lässt.

Spyware: Bei Spyware handelt es sich um Schadsoftware, die ohne Erlaubnis des Benutzers Daten von einem System, auf dem diese installiert ist, oder der Interaktion mit diesem System, sammelt. Hierunter fallen unter anderem Keylogger und Password-Stieler. [23, S. 419, 35, S. 4]

Ransomware: Ransomware—auch Erpressersoftware genannt —sind „Schadprogrammen, die den Zugriff auf Daten und Systeme einschränken oder unterbinden“ [30]. Dies wird meist durch das Verschlüsseln oder Löschen der Daten erreicht. Häufig senden Ransomware zudem die Daten vorher an einen anderen Server. Zur Wiederherstellung der Daten wird anschließend Lösegeld gefordert, indem eine Datei abgelegt wird, oder eine Nachricht angezeigt wird. [30]

Bot: Bots sind Programme, die ohne Einverständnis ferngesteuert werden. Bots richten auf dem infizierten Rechner weiteren Schaden an, indem sie beispielsweise weitere Malware installieren oder als Teil eines Botnetzes mit anderen Bots unter anderem Spam-Mails versenden oder durch einen Distributed Denial of Service versuchen einen anderen Rechner zu überlasten. [27]

Adware: Adware sind Programme, die auf dem infizierten Rechner dafür sorgen, dass Werbung angezeigt wird. Dabei zeigt die Malware entweder selbst Werbefenster an oder modifiziert dafür die Internetbrowser. Häufig ist Adware auch Spyware, sodass sie die Werbung effizienter an die Opfer ausrichten können. [25].

Weitere typische Malware-Klassifizierunge sind Dropper und Backdoor, wobei erstere Malware beschreibt, die weitere Malware versteckt installiert und zweitere, die auf einem System unbefugten Zugang für Dritte ermöglicht. [28, 26]

Festgehalten werden kann also, dass Malware eine Gruppe relativ verschiedener Arten von Programmen ist, wodurch es schwierig ist, Malware sicher zu erkennen. Sie eindeutig zu klassifizieren ist jedoch auch nicht einfach, da Malware sich oft mehreren Klassen zuzuordnen , auch da diese Klassen teils stark ineinander übergehen. Dazu kommt noch, dass selbst, wenn man den kompletten Sourcecode der Client-Anwendung kennt, man nicht weiß, was auf dem Backend-Server passiert.

3.2.2 Malware ist oft verschleiert

Es gibt zwar viele verschiedene Ansätze, um Malware zu identifizieren, doch keine davon bietet eine hundertprozentige Sicherheit [1, S. 10]. Grund hierfür ist neben der bereits genannten Tatsache, dass Malware sehr verschieden ist und teils ihr unerwünschtes Verhalten in nützlichen Programmen versteckt, auch Obfuskation/Verschleierung und Verschlüsselung. Verschleierung beschreibt dabei die absichtliche Veränderung des Programmcodes, sodass dieser schwerer nachvollziehbar ist, ohne dabei die Funktionalität zu ändern. Beispiele dafür sind der Austausch von Anweisungssequenzen, die das Gleiche bewirken, das Einfügen von nicht benutztem oder redundantem Code und das Ändern der Reihenfolge von Anweisungen und Funktionen. Aber auch das Verpacken in einer sich selbst entpackenden Datei über einen Packer ist eine häufige Methode von Malware, um vor Virenscannern unentdeckt zu bleiben [55, S. 18858]. Des Weiteren sind auch teils die enthaltenen Strings oder Sprung- und Aufrufadressen verschlüsselt. [1]

Im Folgenden wird aufgrund ihrer Ähnlichkeit zum verschleierten Code die hier erwähnte Art vom verschlüsselten Code auch als eine Art von verschleiertem Code angesehen.

Bei einer dynamischen Analyse oder der dynamischen Bestimmung der tatsächlichen Anweisungsreihenfolge über dafür entwickelte Programme haben fast alle Obfuskationsmethoden zwar keinen Effekt, doch das Ausführen der Malware ist häufig nicht praktikabel. Zur Analyse kann eine Anwendung in einer Sandbox wie CAPEv2 Sandbox[3] ausgeführt werden, doch beim normalen Betrieb ist dies zu resourcenaufwendig und dauert zudem zu lange. Zudem versuchen viele Malware zu erkennen, ob sie in einer Sandbox oder einer virtuellen Maschine ausgeführt werden, um in diesen Fällen sich wie eine gutartige

[3] Verfügbar unter https://github.com/kevoreilly/CAPEv2.

Software zu verhalten. Aus diesem Grund ist auch die statische Analyse, zu welcher auch der in dieser Arbeit betrachtete merkmalslose Ansatz gehört, von großer Relevanz. [1, S. 10]

In einer statischen Analyse lassen sich meist ebenfalls verpackte Dateien untersuchen, da die Dateien meist auch ohne sie auszuführen entpackt werden können [55, S. 18858, 20].

Generell ist bezüglich Verschleierung zu beachten, dass allein aufgrund vorhandener Obfuskation eine PE jedoch nicht als Malware deklariert werden kann, da auch seriöse Hersteller ihre Software beispielsweise zum Schutz vor Kopierungen oder vor Manipulationen bei Online-Spielen verschleiern [72, S. 6].

3.3 Neuronale Netze

3.3.1 Faltungsschicht

Faltungsschichten (engl.: convolution layers), die auf die Ideen von Fukushima, Waibel, Hanazawa u. a. und LeCun basieren, bilden die Grundlage aller in dieser Arbeit betrachteten Modelle.

Bei diesen wird sinnbildlich eine Faltungsmatrix über eine in der Regel deutlich größere mehrdimensionale Matrix bewegt, und dabei eine Matrixmultiplikation durchgeführt [48, S. 5]. Das Ergebnis wird dann als Ausgaberasters oder Feature-Map bezeichnet [48, S. 5, 52, S. 13]. Mathematisch lässt sich dies mit dem diskreten Konvolutions-/Faltungoperator oder der Kreuzkorrelation beschreiben, welche sich nur darin unterscheiden, dass bei der Konvolution die Faltungsmatrix gespiegelt verwendet wird [37, S. 328f]. In den meisten Implementierung, wie beispielsweise der von TensorFlow, wird dabei die Kreuzkorrelation verwendet, da diese einfacher zu verwenden ist [37, S. 329, 82]. Die n-dimensionale Kreuzkorrelation mit $n \in \mathbb{N} \geq 1$ auf der Eingabe I kann mit folgender Formel 3.1 berechnet werden, wobei bezogen auf diesen Anwendungsfall i und j Vektoren der Länge n sind, die den Offset auf der Feature-Map F beziehungsweise in der Faltungsmatrix K der einzelnen Dimensionen angeben:

$$F(i_1, \cdots, i_n) = (K * I)(i_1, \cdots, i_n) = \sum_{j_1} \cdots \sum_{j_n} I(i_1 + j_1, \cdots, i_n + j_n) K(j_1, \cdots, j_n) \quad (3.1)$$

Es ist zu erkennen, dass die einzelnen Werte der Feature-Maps meist nicht von allen Eingabewerten beeinflusst werden, sondern nur von dem Bereich, der von der Faltungsmatrix, welche in diesem Kontext auch als Filter bezeichnet wird, abgedeckt wird. Die Maße des Feldes, durch welches eine Ausgabe beeinflusst wird, wird auch als lokales rezeptives Feld bezeichnet [52, S. 13, 54, S. 399]. Werden dabei auch die vorherigen Schichten berücksichtigt, so ist dies dann das indirekte oder globale rezeptive Feld [48, S. 5]. Aufgrund dieser spärlichen Konnektivität eines einzelnen Neurons geht zwar Information verloren, doch hat dies den Vorteil, dass dadurch eine Robustheit gegenüber Verschiebungen der Merkmale innerhalb der Eingabe entsteht [52, S. 14, 53, S. 2283]. Bezogen auf die Erkennung von schadhaftem Code bedeutet dies, dass ein Faltungsnetz eine schadhafte Aktion in einer PE wiedererkennen kann, auch wenn diese an einer ganz anderen Position als in den Trainingsdaten ist. Zudem reduziert sich so auch die Anzahl der benötigten Rechenoperationen im Vergleich zu einer vollständig verbundenen Schicht.

Die Werte/Parameter innerhalb eines Filters werden auch als Gewichte bezeichnet und sind die einzigen Werte, die neben einem einheitlichen Bias-Wert, welcher auf das Ergebnis der Kreuzkorrelation addiert wird, während der Backpropagation angepasst werden. Aufgrund der geringeren Anzahl dieser Gewichte sinkt zudem das Risiko für Overfitting, also dass ein Modell nicht aufgrund allgemeingültiger, sondern eingabenspezifischer Eigenschaften zu einer bestimmten Entscheidung kommt [81, S. 2]. Dadurch würde das Modell bei den Trainingsdaten eine hohe Genauigkeit haben, ohne bei den Validierungs- oder Testdaten eine annähernd gleich gute Genauigkeit vorzuweisen.

In der Praxis werden zudem meist mehrere Filter verwendet, damit das Modell deutlich mehr Merkmale extrahieren und berücksichtigen kann. Die einzelnen Feature-Maps werden anschließend übereinandergelegt, wodurch die nächste Schicht mehr Kanäle bekommt. Dadurch ist das Netz in der Lage zuerst kleinere Merkmale und dann abstraktere Merkmale zu erkennen. Da abstraktere Merkmale meist weiter auseinanderliegen, ist es häufig vorteilhaft, wenn die Filter in späteren Schichten etwas größer als die der ersten Schichten sind. [81, S. 3, 48, S. 5]

Dies beides führt jedoch dazu, dass ab der zweiten Schicht der Rechenaufwand deutlich ansteigt, weshalb es sinnvoll sein kann, die Schrittweite (eng: stride) der Filter zu erhöhen. Dadurch können zwar möglicherweise kleinere Merkmale nicht erkannt werden, jedoch kann bereits eine Schrittgröße von 2 den benötigten Rechenaufwand annähernd halbieren: Für die Größe der Ausgabe gilt in einer 1D-Faltungsschicht: $\frac{i-f+p+s}{s}$, wobei i die Länge der Eingabe, f die Filtergröße und p die Größe des Paddings ist, wel-

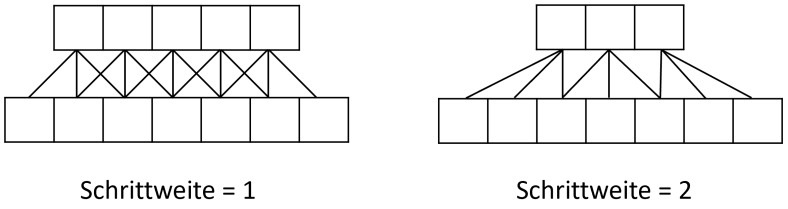

Abbildung 3.1: 1D-CNN mit verschiedenen Schrittweiten. Angelehnt an [48, S. 6]

che hier als x zusammengefasst werden können. Schlussendlich ist so zu erkennen, dass $\lim_{x\to\infty} \frac{i-f+p+s}{s} = \frac{1}{s}$ gilt. Wie in Abbildung 3.1 zu erkennen ist, rücken dabei jedoch auch weiter entfernte Bereiche zusammen, wodurch weiter verteilte Eigenschaften erkannt werden können. Jedoch besteht die Wahrscheinlichkeit, dass kleinere Muster teilweise nicht mehr erkannt werden, da Muster nun abhängig von ihrer Position von anderen Parametern berücksichtigt werden. [48, S. 5f]

Inklusive einer Aktivierungsfunktion A für die Nichtlinearität ergibt sich dann insgesamt folgende erweiterte Formel 3.2, wobei k den Kanal der Ausgabe, b_k den Bias-Wert des Kanals k und s die Schrittweite der Filter angibt:

$$F(i_1, \cdots, i_n, k) = A(\sum_{j_1} \cdots \sum_{j_n} I(i_1 \cdot s + j_1, \cdots, i_n \cdot s + j_n) K(j_1, \cdots, j_n, k) + b_k) \quad (3.2)$$

3.3.2 Pooling

Eine weitere Möglichkeit zur Reduzierung des Rechenaufwandes und gleichzeitiger Vergrößerung des indirekten rezeptiven Felds ist die Verwendung von Pooling-Schichten. Diese Art des Subsamplings wurde bereits bei den Anfängen von CNN verwendet und erwies sich schon häufig als sehr nützlich [54, S. 400, 16, S. 3481]. Über einfache —meist komplett statische —Funktionen werden dabei mehrere Eingaben eines Kanals zu einem neuen Wert zusammengefasst, wodurch mit sehr wenig Rechenaufwand die Schärfe der Eingabe reduziert wird [54, S. 400, 16, S. 3481]. Im Vergleich zu einer Erhöhung der Schrittweite einer vorhandenen Faltungsschicht ist der Rechenaufwand zwar etwas höher, doch dafür wird das Netz zuverlässig weniger anfällig für Verschiebungen und Verzerrungen [54, S. 400], wobei Verzerrungen bezogen auf PE mit zusätzlich eingefügten Anweisungen gleichgesetzt werden kann. Prinzipiell ist auch bei der Pooling-Schicht eine Schrittweite von 1 möglich, doch verliert man dadurch den Vorteil, dass die Größe der Schicht stark verringert wird, wodurch auch der Rechenaufwand weniger reduziert und

das rezeptive Feld weniger vergrößert wird. Den eigentlichen Vorteil erhält man also erst ab einer Schrittweite von 2 in Kombination mit einer Filtergröße von 3, da anders als bei einer Faltungsschicht hier die Operation bei allen Verbindungen gleich ist. Trotz der Schrittweite von 2 werden bei einer Filtergröße von 3 also alle zwei benachbarten Zellen zusammengefasst, und eine Verschiebung der Werte in der Eingabe um 1 verschiebt nur die Werte in der Ausgabe um 1 und bleibt ansonsten unverändert —zumindest wenn man die Ränder ignoriert. Idealerweise reduziert eine Pooling-Schicht dabei jedoch nicht nur die Schärfe der gesamten Eingabe, sondern hebt die relevanten Informationen von den unwichtigen Informationen oder gar Unreinheiten hervor [2, S. 879].

Aufgrund ihrer Einfachheit in der Umsetzung und Berechnung sind die zwei häufigsten Pooling-Arten Max-Pooling und Average-Pooling [16, S. 3481], doch es gibt noch viele andere weniger häufig genutzte Pooling-Arten [2, 16]

Was Pooling-Schichten für diese Arbeit besonders wichtig macht, ist, dass eine globale Pooling-Schicht einem Klassifizierungs-CNN die Fähigkeit verleiht, mit unterschiedlich großen Eingaben arbeiten zu können, ohne dass die Eingabe skaliert werden muss. Dafür wird nicht mehr pro Filterposition eine neue Ausgabe erzeugt, sondern die Pooling-Funktion wird jeweils auf den gesamten Kanal angewendet, wodurch für jeden Eingabekanal eine Ausgabe entsteht [2, S. 883]. Damit ist auch bei Sequenzen beliebiger Länge zumindest in der Theorie möglich, dass verschiedene erkannte Muster miteinander verarbeitet werden können, unabhängig davon, wo die Muster erkannt werden. Globales Pooling macht das Modell also invariant gegenüber der Verschiebung von Mustern innerhalb der Eingabe.

3.3.3 Verlustfunktion

Ein für diese Arbeit wichtiges zentrales Element für das Lernen jedes neuronalen Netzwerks ist die Verlustfunktion (eng: Loss-function). Durch diese wird die Abweichung zwischen der erwarteten und tatsächlichen Ausgabe des Modells berechnet, welche dann auch als Verlust (eng: loss) bezeichnet wird. Dabei ist diese Verlustfunktion jedoch häufig nicht linear, sondern enthält eine polynome, exponentielle oder logistische Funktion. Grund dafür ist, dass der Gradient der Funktion über den Optimierer den Ausgangspunkt für die Berechnung der Änderungen der Parameter bestimmt und eine nicht lineare Funktion in vielen Fällen das Lernverhalten verbessert. So kann beispielsweise eine große Abweichung

bei einer Ausgabe zu einem größeren Gradienten als eine kleinere Abweichung führen. [65, 51]

3.3.4 Dynamische Faltungsschicht

Um wiederum die Kapazität des Modells, also die Fähigkeit, das Muster beziehungsweise die Funktion für die Klassifizierung der Daten erkennen und abbilden zu können, zu steigern, wird bei CNN üblicherweise die Anzahl der Filter und/oder die Filtergröße erhöht [86, S. 1]. Yang, Bender u. a. und Chen, Dai u. a. haben jeweils relativ unabhängig eine alternative Lösung entwickelt und untersucht, bei welcher zwar der Parameterraum deutlich vergößert wird, ohne dabei den Rechenaufwand stark zu erhöhen [12, 86].Wie links in Abbildung 3.2 dargestellt, wird dies durch eine über Attention —in der Abbildung als ROUTE FN bezeichnet—gesteuerte Kombination mehrerer paralleler Filter zu von der Eingabe abhängigen Filtern erreicht. Dadurch vergrößert sich die Anzahl der Parameter und damit die Anzahl der möglichen Muster, die das Netz erkennen kann. Die Aufsummierung der einzelnen Gewichtsmatrizen $\tilde{W}_k$ und Biaswerte $\tilde{b}_k$, welche jeweils durch die berechneten Attention-Faktoren $\pi(X)_k$ reguliert werden, findet dabei pro Schicht nur einmal für die gesamte Eingabe statt. Dieser Ansatz ist damit, wie in Abbildung 3.2 und Formel 3.3 zu erkennen ist, mathematisch äquivalent mit dem Mischung von Experten (eng: Mixture of Experts) Verfahren, doch benötigt deutlich weniger Rechenschritte [86, S. 3]. Aufgrund dieser Gegebenheit werden im Folgenden die einzelnen Gewichtsmatrizen auch als Experten bezeichnet.

$$(\pi(x)_1(W_1) + \cdots + \pi(x)_n W_1) * x = \pi(x)_1(W_1 * x) + \cdots + \pi(x)_n(W_n * x) \tag{3.3}$$

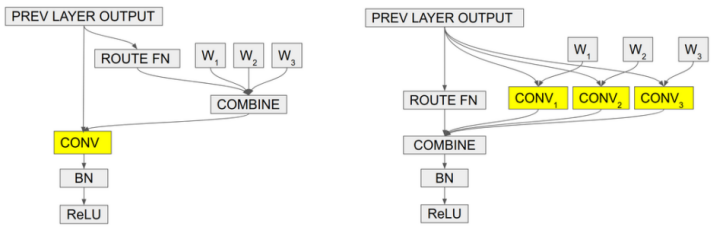

Abbildung 3.2: Dynamische Faltungsnetze und das Mischung von Experten Verfahren sind mathematisch äquivalent, unterscheiden sich jedoch stark in der benötigten Rechenleistung. Von [86, S. 737, Ausschnitt]

Unterscheiden tun sich die in [86] und [12] vorgestellten Verfahren vor allem darin, dass beim Letzteren statt einer Sigmoid-Aktivierung für die Attention eine Softmax-Aktivierung vorgeschlagen wird [12, S. 4f]. Dadurch verringert sich wieder deutlich die Anzahl der möglichen Parameterausprägungen für die schlussendliche Faltung. Da diese Verringerung jedoch nur durch eine geringere Kombinierbarkeit der Gewichte zustande kommt und die Anzahl der Parameter gleich bleibt, erleichtert dies vor allem das Lernen der Faltungs- und Attention-Gewichte. Trotz möglicherweise geringerer Repräsentationskraft konnte daher experimentell gezeigt werden, dass die Variante mit Softmax der mit einer Sigmoid-Aktivierung bessere Ergebnisse liefert, selbst wenn die Anzahl der parallelen Filter / Experten reduziert wird [12, S. 5]. Da jedoch die von der Sigmoid-Aktivierung bei der Initialisierung gegebene annähernd gleichmäßige Attention sich als vorteilhaft für das Lernen erweist, wird bei der Softmax-Aktivierung die Verwendung einer über das Training hinweg sinkenden Temperatur τ vorgeschlagen, wodurch die einzelnen Experten zu Beginn gleichmäßig und gleichzeitig angelernt werden können [12, S. 4]:

$$\pi_k = \frac{e^{\frac{z_k}{\tau}}}{\sum_j e^{\frac{z_j}{\tau}}} \tag{3.4}$$

Chen, Dai u. a. [12, S. 5] schlagen dabei für τ einen Startwert von 30 vor, welcher linear innerhalb der ersten 10 Epochen auf 1 reduziert wird. Für die Attention wird darüber hinaus das in [40] vorgestellte Squeeze-and-Excitation-Verfahren vorgeschlagen, bei welchem Verfahren die einzelnen Kanäle zuerst über ein globales Durchschnitts-Pooling zu einzelnen Werten zusammengefasst werden, und dann anhand zwei vollständig vernetzten Schichten die Skalierung der einzelnen Kanäle bestimmt wird. Bei der ersten vollständig verbundenen Schicht wird dabei die Anzahl der Neuronen weiter reduziert,

Statt jedoch wie bei Squeeze-and-Excitation die eigentlichen Kanäle zu skalieren, werden anschließend bei den DynConvs die einzelnen Faltungsgewichte skaliert. [40, S. 3, 12, S. 3]

4 Entwurf und Beschreibung der Experimente

4.1 Definition der Modelle

4.1.1 Format der Eingabedaten

Bei der Wahl des Eingabeformates der Daten für das Netz gilt es sowohl die Größe, den Informationsgehalt und die Einfachheit der Informationsentnahme für das neuronale Netz zu berücksichtigen. Dabei ist zu beachten, dass ein einfaches Modell, das nicht (mehrere) Milliarden Parameter hat, nach dem aktuellen Forschungsstand nicht in der Lage ist, aus einem Eingabeformat alle relevanten Informationen zu entnehmen. Aufgrund des sehr hohen Rechenaufwandes gibt es jedoch bei den großen Modellen keine wirkliche Forschung auf diesem Gebiet. Zumindest bei den kleineren Modellen (nicht mehr als wenige Millionen Parameter), schneiden jedoch die Modelle, welche ein Programm aus verschiedenen Sichten als Eingabe haben, in Experimenten bedeutsam besser ab [34, S. 13].

Auch daher sind die im Folgenden miteinander verglichenen Eingabeformate nur die in dieser Arbeit betrachteten merkmalslosen Formate, die den Programmcode als Teil des Programms auf einem niedrigen Level, d.h. nur wenig vorverarbeitet, darstellen. Diese sind Byte-Sequenzen, Anweisungs-/Mnemonics-Sequenzen mit oder ohne Operanden und Opcode-Sequenzen, inklusive der Darstellung all dieser als Graustufenbilder. In der praktischen Nutzung sollten diese dann mit Modellen mit anderen Sichten, wie zum Beispiel einer reinen API-Aufruf-Sequenz oder Metadaten wie Sequenzlänge, Informationen aus dem Header, Entropie der Bytes und anderen Eigenschaften kombiniert werden. Welche Merkmale genau gewählt werden sollten, wird in dieser Arbeit nicht betrachtet und auch die vorherige Aufzählung ist nur exemplarisch ohne konkrete Nachweise ihrer Effektivität.

Bei dieser Art der Sicht stellt zumindest in der Theorie eine Sequenz der Bytes, wie sie in der Datei enthalten sind, die meisten Informationen bereit. Dies lässt zumindest die Tatsache anmuten, dass bei diesem Ansatz dem Netz annähernd dieselben Informationen zur Verfügung stehen, wie der CPU, auf welcher der Code ausgeführt wird. Nicht berücksichtigt werden kann dabei jedoch unter anderem die Auflösung der Aufrufe externer Methoden, d.h. das Netz kann, wenn überhaupt, erkennen, dass etwas aufgerufen wird, nicht aber was aufgerufen wird. Erkennt das neuronale Netz die Byte-Folge *ff 15 10 21 44 00*, so kann es in der Theorie zumindest lernen, dass es sich hier um etwas wie eine *CALL*-Anweisung handelt, jedoch ist es für das Netz praktisch unmöglich, zu erschließen, welche Funktion aufgerufen wurde. Bei der i386-Architektur (x86) ist *10 21 44 00* in diesem Beispiel die in Little Endian dargestellte absolute Adresse, an welcher die eigentliche Adresse der Funktion steht [43, S. 249]. Da die erste Adresse eine Adresse ist, die meist im .rdata-Abschnitt steht, müsste dieser Abschnitt mit übergeben werden, doch selbst wenn der vollständige Inhalt der PE dem Netz vorliegt, ist es sehr unwahrscheinlich, dass ein praktikable umsetzbares Modell die zweifache Adressauflösung durchführen könnte. Dazu kommt jedoch noch, dass Adressen auch oft relativ angegeben werden, und Dynamic Link Librarys (DLLs) zudem noch dynamisch nachgeladen werden können [43]. Letzteres führt dazu, dass nicht mal die letzte Version (2.2) von Dependency Walker, alle Abhängigkeiten statisch erfassen kann, sondern diese nur während der Laufzeit des Programms ermitteln kann [84]. Theoretisch ist es jedoch denkbar, dass das Netz lernt, über den Kontext zu erkennen, welche Art von Methode aufgerufen wird. Dass dies jedoch zuverlässig funktioniert, ohne dass das Netz explizit darauf trainiert wird, ist jedoch anzuzweifeln.

Da eine aufgerufene externe Methode jedoch großen Einfluss auf den Kontrollfluss hat, ist ein hier vorgestellter Ansatz, mit diesem Problem umzugehen, die Auflösung der Zieladresse in der Vorverarbeitung. Dabei ist zu beachten, dass es einerseits mehrere Möglichkeiten gibt, um eine Funktion aufzurufen, und zudem die meisten Funktionsaufrufe keine externe, sondern interne Funktionen aufrufen. Wenn die aufgerufene Funktion nun eine externe Methode ist, so kann die Adresse durch eine einheitliche Adresse, welche über alle Dateien gleich ist und möglich als reale Adresse vorkommt, ersetzt werden. Da es jedoch sehr viele verschiedene DLL-Funktionen beziehungsweise API-Aufrufe gibt, sollte hier eine Einbettung vorgenommen werden, welche ähnliche API-Aufrufe zusammenfasst [4, S. 4]. Eignen tut sich hier beispielsweise die von Mikolov, Chen u. a. [59] vorgestellte Word2Vec Methode [4, S. 4]. Adressen, die auf interne Funktionen verweisen oder nicht aufgelöst werden können, sollten vermutlich ebenfalls vereinheitlicht werden, da diese

sonst weiterhin Werte sind, die dem Netz keine nachvollziehbaren Informationen geben können und das Netz sonst zusätzlich lernen müsste, diese zu ignorieren. Eine Ausnahme könnten hier die relativen near call- oder near jump-Adressen sein, wo untersucht werden müsste, ob diese ebenfalls vereinheitlicht werden sollten oder nicht. Wenn die Fenstergröße eines CNN groß genug ist oder ein mitgetragener Zustand wie in einem LSTM vorhanden ist, dann könnte zumindest in der Theorie das Modell lernen, dass dies ein Verweis auf die spätere oder frühere Anweisungsfolge ist. Ob dies für die Erkennung jedoch einen Mehrwert hat, ist ungewiss. In [68] wird ein relativ ähnlicher Ansatz namens „Instructions with Function Resolution“ [68, S. 4] vorgestellt, jedoch wird bei diesem die Ersetzung im Assemblercode durchgeführt und die Adressen für interne Funktionen werden nicht vereinheitlicht [68, S. 4]. Auch wird die erzeugte Sequenz mit einer N-Gram-Analyse über Regression und nicht mit einem neuronalen Netzwerk verarbeitet [68].

Nach dem aktuellen Forschungsstand liefern jedoch Modelle, die auf den Anweisungen basieren, eine deutlich größerer Genauigkeit [34, S. 13]. Bei dieser Aussage ist jedoch zu beachten, dass es aktuell keine (bedeutende) Forschung gibt, wo nur die rohen Bytes der Codeabschnitte als Eingabe verwendet werden, ohne diese vorher in ein Graustufenbild zu transformieren. So werden auch im Vergleich in [34, S. 13] Anweisungssequenz-Modelle mit Modellen, die alle Bytes der Datei in ihrer rohen Form bekommen, verglichen. Letztere haben zwar noch mehr Informationen, doch kommt hier das Problem hinzu, dass Sequenzen an Bytes je nach Abschnitt innerhalb der PE wie .text und .rdata eine andere Bedeutung haben [46, 49, S.1]. Bereits, wenn nur die Bytes der Programmanweisungen als Eingabe verwendet werden, wird das Lernen möglicherweise dadurch erschwert, dass Bytes je nach Position und Kontext etwas anderes bedeuten [43, 35, S. 14]. Dadurch, dass bei den Bytes der Programmanweisungen keine einheitliche Ausrichtung beziehungsweise Anweisungslänge gibt [43], kann auch bei Faltungsschichten dies nicht durch die Schrittweite gelöst werden. Helfen könnte aber eine zusätzliche Eingabe, welche die aktuelle Position des Bytes übergibt. Auch gibt es verschiedene Kodierungen für den gleichen Befehl, welche in der Anweisungssequenz direkt zusammengefasst werden [43]. In der Byte-Darstellung ist es dagegen aus den zuvor geschriebenen Gründen schwierig umzusetzen, dass diese durch eine Einbettung ähnlich an das neuronale Netz übergeben werden können. Zudem verringert die Abwesenheit der Parameter die Anzahl der Möglichkeiten, wie eine Funktionalität in den Eingabedaten dargestellt werden kann. Es gibt sowieso schon sehr viele Möglichkeiten, wie etwas umgesetzt werden kann, wodurch es für ein Modell nicht einfacher wird, wenn beispielsweise Register ausgetauscht werden können und das Modell deren Gleichheit lernen muss —siehe dazu auch [68, S. 4]. Schlussendlich

ist dies aber nur eine Vermutung, da das Weglassen der Operanden auch ein Informationsverlust bedeutet. Beispielsweise muss es sich bei der Sequenz `mov mov cmp je` nicht zwingend um eine Verschleierung einer `jmp` Anweisung handeln, doch bei `mov rsi,7 mov rbx,7 cmp rsi,rbx je 140002EC1` ist es eindeutig: da die zwei `mov` in die Register rsi und rbx jeweils den Wert 7 laden, setzt die Vergleichsanweisung immer den Statusflag, dass die Werte der Register gleich sind, weshalb die jump short if equals Anweisung `je` immer ausgeführt wird und so diese Sequenz der bedingungslosen Sprunganweisung `jmp` gleichgesetzt werden kann [43]. Das genaue Zusammenspiel der Anweisungen kann also genauso wenig wie unnötige Anweisungswiederholungen erkannt werden. Das beste von beiden ist es vermutlich daher, wenn der gesamte Assemblercode als Eingabe verwendet wird. Eine entsprechende Möglichkeit für die Einbettung kann aus Asm2Vec [17] abgeleitet werden. Bei einer vollständigen Sequenz der Anweisungen mit Operanden besteht jedoch wieder das Problem der vielen Möglichkeiten in der Umsetzung, die das Lernen erschweren können. Über Asm2Vec bekommen ähnliche Operanden beim Trainieren zwar eine ähnliche Einbettung, jedoch wird nicht beschrieben, wie mit Konstanten umgegangen wird [17]. So müssten auch Operanden wie `[rbp+18h]` und `[rbp+4h]` zusammengefasst werden, damit es nicht vorkommen kann, dass es für neue Operanden keine Vektordarstellung gibt. Dies führt zwar erneut zum Informationsverlust, doch die zuvor genannte Verschleierung kann noch mit einer höheren Sicherheit erkannt oder ausgeschlossen werden. Es kann zwar nicht mehr bestimmt werden, ob der Sprung immer oder nie ausgeführt wird, doch da immer noch die Information verfügbar ist, dass die Werte für die Auswertung konstant sind, kann es sich jedoch nicht um eine typische dynamische Auswertung handeln.

Ein großer Nachteil, welcher gerade aufgrund der langen Sequenzen an Anweisungen hier besonders relevant ist, ist jedoch die Größe der Einbettung. Asm2Vec weist zwar mindestens zwischen einer Vektorgröße von hundert bis vierhundert eine relativ stabile Güte auf, doch ist bereits zwischen 200 und 100 ein geringer Abfall zu erkennen [17, S. 482]. Deshalb ist es fraglich, ob die Größe des Vektors weiter auf um die 100 reduziert werden kann, beziehungsweise die Vektorgröße bei um die 200 bleiben kann, wenn durch die aufgelösten Funktionen die Anzahl der verschiedenen Token stark steigt. Bei größeren Vektorgrößen wird das Trainieren andererseits sehr speicherintensiv. Op2Vec erwies sich dagegen bereits bei Dalvik Executable, welche 255 verschiedene Opcodes hat, bei einer geringen Vektorgröße von 2 [47]. In der AMD64 Architektur gibt es zwar deutlich mehr verschiedene Anweisungen, doch konnte bereits mit einer sehr ähnlichen Einbettung mit einer Vektorgröße von vier ein über eine Mnemonic-Sequenz Malware mit einer hohen

Genauigkeit erkannt werden [34, S. 16, 28ff]. Hilfreich für die Qualität des Modells scheint dabei zu sein, dass Mnemonics, die nur in sehr wenigen Trainingsdaten vorkommen, durch ein neues *UNKNOWN* Token zu ersetzen [34, S. 28, 15, S. 3]. Eine alternative Lösung um den benötigten Speicher zu reduzieren, ist, dass lange Sequenzen einfach abgeschnitten werden oder eine Sequenz maximaler Länge zufällig aus der Sequenz entnommen wird. Dies hat jedoch den Nachteil, dass besonders die Verhaltenslabel nicht auf diesen Teil zutreffen müssen und zudem es auch sein kann, dass die ausgewählte Teilsequenz nicht schädlich ist. So enthalten unter anderem vor allem Trojaner auch besonders viel Code, der allein nicht schädlich ist. Dadurch ist dieser Ansatz nicht wirklich zielführend.

Ein weiteres Format, welches in Nataraj, Karthikeyan u. a. [60] vorgestellt wird, ist die Darstellung der Malware als ein 2D-Graustufenbild. Dieser Ansatz hat jedoch mehrere Nachteile. Die zweite Dimension führt zu eigentlich nicht im Programmcode vorhandenen Beziehungen von Abschnitten, die nach der Transformation beieinanderliegen [35, S. 9, 67, S. 9]. Diese Art der Eingabe ist zudem besonders anfällig für Verschleierung durch Codeblockverschiebungen und der Einbindung zusätzlichen Codes [60, S. 5], auch da bereits kleine lokale Änderungen durch die Verschiebung komplett verschiedene Muster in der zweiten Dimension verursachen. In Vergleichen schneiden Modelle mit diesem Eingabeformat daher meist deutlich schlechter ab—vgl. z.B. [56, S. 11].

Bei manchen Modellen wie HYDRA von Gibert, Mateu u. a. werden auch Byte- und Anweisungssequenzen als verschiedene Sichten behandelt [34]. Ob die Verwendung beider Sichten wirklich Vorteile hat oder ob die Ersetzung beider Sichten durch eine beispielsweise über Asm2Vec kodierte Sequenz der gesamten Anweisungen gegebenenfalls vorteilhafter ist, muss noch weiter (in zukünftigen Arbeiten) untersucht werden.

Eine separate Kodierung aller unterschiedlichen Opcodes einer Anweisung könnte beispielsweise mehr Informationen über die Kompilierung in die Analyse hinzufügen, doch entstehen dadurch vermutlich vor allem nur mehr verschiedene Token. Die Einbettungsgöße kann dabei aber wahrscheinlich gleich bleiben, da die Einbettung aller Opcodes einer Anweisung vermutlich sehr ähnlich sein dürften, wenn nicht gerade die Verwendung untypischer Opcodes ein entscheidendes Merkmal ist—was möglich ist. Es ist jedoch auch zu beachten, dass durch die Unterscheidung zwischen den konkreten Opcodes das Modell möglicherweise stärker vom Concept Drift[1]. betroffen ist. Grund dafür ist unter

[1] Die Art und Weise wie programmiert wird, die Compiler (Versionen) und damit gegebenenfalls auch der Assemblercode und die verwendete Programmierschnittstelles (APIs) ändern sich. Daher müssen Modelle fortlaufend trainiert werden, sodass die Genauigkeit nicht mit der Zeit abnimmt. [11, 64]

anderem, dass diese Unterscheidung auch dazu führt, dass zwischen den Opcodes unterschieden wird, die häufiger oder ausschließlich in 32-Bit anstelle von 64-Bit Programme verwendet werden, wobei mit einem Rückgang von 32-Bit Anwendungen in der Zukunft zu erwarten ist. Da in der Literatur bis dato üblicherweise nicht zwischen den einzelnen Opcodes einer Anweisung unterschieden wird, wurde auch in dieser Arbeit diese Vorgehensweise verwendet. Welchen Einfluss dies auf die erreichte Genauigkeit und Anfälligkeit für ein Concept Drift hat, muss noch in späteren Forschungen untersucht werden.

Schlussendlich wurde in dieser Arbeit daher eine Sequenz an Anweisungen ohne Operanden verwendet, die im Folgenden auch kurz als Mnemonics-Sequenz oder Anweisungssequenz bezeichnet wird. Die zuvor für die Bytedarstellung vorgestellte Funktionsauflösung der Zieladressen in der Vorverarbeitung lässt sich dabei auch mit dieser Mnemonics-Sequenz kombinieren. Die aufgelösten Funktionsnamen werden dann als seien sie eigenständige Funktionen hinter der `call`-Anweisung eingefügt. Alternativ könnte der aufgelöste Funktionsname auch die *Call*-Anweisung ersetzen, doch hätte dies den Nachteil, dass eventuell strukturelle Informationen wie eine hohe Anzahl an Funktionsaufrufen verloren geht, je nachdem, wie die Funktionsnamen schlussendlich eingebettet sind. Damit eine `call`-Anweisung nicht direkt mit einem Aufruf einer internen Funktion gleichgesetzt wird, wird bei Aufrufen interner Methoden statt dem Funktionsnamen ein *INTERN*-Token angehängt. Wenn die Zieladresse über den Inhalt der Register bestimmt wird und dieser nicht einfach zu bestimmen ist, dann wird stattdessen ein *REGISTER*-Token eingefügt, welches sowohl für eine externe als auch interne Methode stehen kann. Kann jedoch der Inhalt des Registers bestimmt werden, doch die Adresse zeigt nicht auf eine statisch gebundene -Funktion, sondern liegt außerhalb des Codebereiches des PE, dann wird ein *UNKNOWN*-Token verwendet. Liegt sie innerhalb, dann wird ein *INTERN*-Token verwendet. Das *UNKNOWN*-Token steht damit unter anderem für alle dynamisch angebundenen DLL-Funktionen, neben den Funktionen. Zeigt eine Sprunganweisung oder eine Anweisungskette an aufeinander zeigende Sprunganweisungen oder Aufrufanweisungen ebenfalls auf eine DLL-Funktion, dann wird auch hinter der Sprunganweisung der Token für die gefundene Funktion ergänzt. Ist die Kette länger als ein festgelegter Grenzwert, dann wird ebenfalls ein *UNKNOWN*-Token eingefügt, falls die Kette mit einer *Call*-Anweisung beginnt. Darüber hinaus wurden alle Funktionsnamen, die insgesamt seltener als 50-mal oder in weniger als 40 Dateien vorkommen, ebenfalls durch das *UNKNOWN*-Token ersetzt. Anweisungsoperatoren, die in weniger als in 5 verschiedene Dateien vorkommen, wurden ebenfalls durch ein neues *UNK*-Token ersetzt. Da `int` bereits häufig von den Compiler für das Trennen von Funktionen verwendet

wird, wurde dieses als Padding benutzt, um innerhalb eines Batches die Daten auf eine einheitliche Länge zu bekommen. Um die Verwendung dieses Paddings einzuschränken, wurden zuvor die Trainingsdaten nach ihrer Größe sortiert, sodass die Daten innerhalb eines Batches bereits eine möglichst ähnliche Größe haben.

4.1.2 Herleitung des Referenz- und Basismodells

Damit die Eignung und der Mehrwert des MTL und der dynamischen Faltungsschicht für die merkmalslose Erkennung von Malware bewertet werden können, benötigt es ein Referenzmodell, zu welchem die Ergebnisse verglichen werden können. Um eine hohe Aussagekraft über die neuen Techniken zu haben, dient das Referenzmodell auch als Basismodell, in welchem die untersuchten Techniken integriert werden. Daher sollte das Referenzmodell bereits ein relativ gutes Ergebnis liefern und eine gewisse Flexibilität für den Lernprozess bieten, sodass davon ausgegangen werden kann, dass die untersuchten Techniken nicht durch die Struktur des Basismodells begrenzt wurden. Ein bereits gut funktionierendes Modell, welches Mnemonic-Sequenzen beliebiger Längen als Eingaben akzeptiert, ist die Mnemonics basierte Komponente des multimodalen Deep-Learning-Frameworks HYDRA von Gibert, Mateu u. a. [34][2]. Dieses ist in Abbildung 4.1 dargestellt.

Das Modell besteht aus einer Einbettungsschicht und drei parallelen Faltungsschichten, die jeweils hundert Filter haben. Für die Filtergrößen gilt $h \cdot k$ mit $h \in \{3, 4, 5\}$ und $k :=$ Größe der Einbettung. Die Dimension wird anschließend jeweils über ein globales Max-Pooling reduziert, bevor die einzelnen Ausgaben konkateniert und dann an die vollständig verbundene Ausgabeschicht mit einer Softmax-Aktivierung übergeben werden. Die unterschiedlichen Filtergrößen sollen dem Netz erlauben, Sequenzen unterschiedlicher Länge zu erkennen. [34, S. 171f]

Da dieses Modell ausschließlich als Ersatz der vorher häufig verwendeten aber sich bei großen Daten als nicht praktikabel erwiesenen n-Gramm-Analyse konzeptioniert wurde

[2] Die Modelle aus [45] verwenden zusätzlich noch einen Autoencoder zur Komprimierung der Daten, was die Ergebnisse für die in dieser Arbeit untersuchten Techniken verfälschen kann. Das hierarchische Modell in [33] erfordert zudem eine größere Vorverarbeitung der Daten, da einzelne Funktionen zusammengelegt werden müssen, was professionelle Software erfordert und zudem fehleranfällig ist, da durch Verschleierung und Optimierung des Assemblercodes es keine Garantie gibt, dass diese zuverlässig erkannt werden können. Wie später zudem begründet, ist zudem die Verwendung des globalen Durchschnitts-Pooling nach der letzten Faltungsschicht dieses Modells nachteilig. Das in [74] vorgestellte Modell wiederum erzwingt durch Padding für alle Eingaben die gleiche Sequenzlänge.

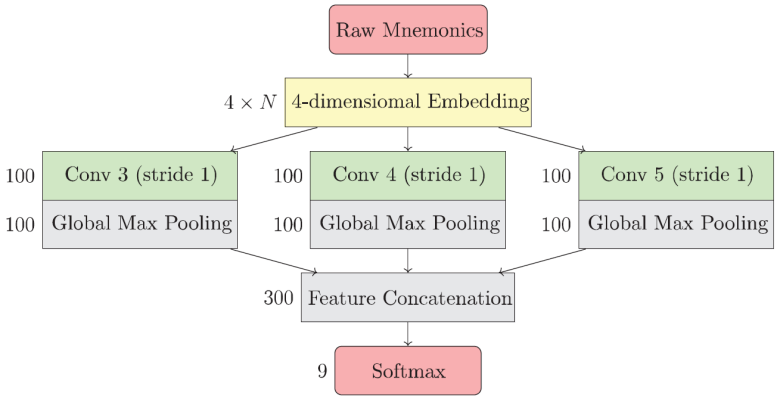

Abbildung 4.1: Aufbau des Referenzmodells: Mnemonics-basierte HYDRA-Komponente. Von [34, S. 6]

[34, S. 171], wurde dieses Modell nicht als alleiniges Referenz-/Basismodell verwendet. Stattdessen dient es als Ausgangspunkt für die Entwicklung eines Referenz- und Basismodells, in welchem auch die DynConvs besser integriert werden können. Da zudem davon auszugehen ist, dass relevante Codeabschnitte meist aus deutlich mehr als 5 Anweisungen bestehen, sollte zudem das rezeptive Feld vergrößert werden. Erreicht kann diese Vergrößerung prinzipiell über vier verschiedene Ansätze: 1. Filter vergrößern, 2. weitere Schichten hinzufügen, 3. Schrittweite erhöhen und 4. Pooling, wobei auf die Vor- und Nachteile letzter beiden bereits in Abschnitt 3.3.1 und 3.3.2 eingegangen wurde.

Eine größere Filtergröße hat den Vorteil, dass nicht nur das rezeptive Feld vergrößert wird, sondern es für das Netz in der Theorie zudem im Vergleich zu den anderen Methoden einfacher sein sollte, größere Muster zu erkennen, da diese vorher nicht komprimiert werden. Aufgrund einer annähernd gleichbleibenden Schichtengröße bleibt zudem eine große Repräsentationsmöglichkeit vorhanden und eine größere Anzahl an Parametern kann zudem die Erkennungskraft des Modells erhöhen. Die größere Anzahl an Parametern und die geringere Notwendigkeit zu Abstraktion kann jedoch auch stark das Risiko für Overfitting erhöhen. Gerade wenn sowieso bereits viel Rechenleistung und Speicher erforderlich ist, stellt dieser Ansatz jedoch sowieso keine nutzbare Möglichkeit dar, da große Filtergrößen den Rechenaufwand und den Speicherbedarf stark erhöhen. Weitere Schichten haben zwar ebenfalls ähnliche Vor- und Nachteile, doch fördern diese im Gegensatz zu großen Filtern zudem die Fähigkeit, abstraktere Eigenschaften zu erkennen [48, S. 5].

Zur Untersuchung, auf welcher Weise das Referenzmodell verbessert werden sollte, wurde in einem eigenen Experiment die Erkennungskraft von verschiedenen einfachen CNNs verglichen. Dafür wurden 12.000 zufällige Sequenzen der Länge 30 aus Zahlen zwischen 0 und 1 mit zwei Kanälen erstellt, aus welchen anschließend drei verschiedene Merkmale beziehungsweise Muster erkannt werden sollten. Das erste der drei Label steht für das Vorhandensein des Wertpaares $(0,6|0,2)$. Das zweite Label steht für das Vorhandensein des Musters $(x_1 > 0,6|x_2 < 0,4)$, $(x_1 > 0,6|x_2 > 0,6)$ $(x_1 > 0,6|x_2 < 0,4)$. Das dritte Label gibt an, dass im ersten Kanal mindestens einmal der Wert 0,9 und im zweiten Kanal der Wert 0,6 vorkommt. Bei der Erzeugung der Sequenzen wurden dabei als Hilfestellung für das Modell alle Werte, die eine kleinere Abweichung als 0,05 von den gewünschten Werten für das erste und letzte Label hatten, zum gewünschten Wert verändert. Es wurden verschiedene einfache CNN miteinander verglichen, die alle bis zu zwei Faltungsschichten und teilweise zwischen diesen Schichten eine Pooling-Schicht haben. Alle Pooling-Schichten haben eine Filtergröße von 3 und einer Schrittweite von 2. Auf die letzte Faltungsschicht folgt jeweils ein Globales-Max-Pooling.

Neben den klassischen Pooling-Schichten wurden auch verschiedene Faltungsschichten mit einer Schrittweite von 2 und einer Filtergröße von 3 als lernende Pooling-Schichten ausprobiert —vgl. dazu auch [80, S. 3].Da eine normale Faltungsschicht—die in den Ergebnissen dieses Experimentes als Conv bezeichnet wird—einige neue Parameter einführt und zudem einen deutlichen Mehraufwand in der Berechnung gegenüber echten Pooling-Schichten hat, wurden auch tiefenweise Faltungsschichten (eng: depthwise convolutions) (DepConvs) und tiefenweise Faltungsschichten mit jeweils einer anschließenden punktuellen Faltungsschicht (eng: depthwise seperable convolutions) (SepConvs)[76, S. 108ff] betrachtet. Bei DepConvs betrachten dabei alle Filter nur einen Kanal, und punktuelle Faltungsschichten sind einfache Faltungsschichten mit Filtern der Größe $1 \cdot k$ mit $k :=$ Anzahl der Kanäle, die die neuen Kanäle kombinieren und auf neue Kanäle abbilden [76, S. 108f]. Je nach Filtergröße kann der Berechnungsaufwand und die Anzahl der Parameter dadurch erheblich reduziert werden, und teils kann dadurch auch die Anzahl der benötigten Trainingsdaten reduziert werden [76, S. 114]. Da beim eigentlichen Malware-Erkennungsproblem aufgrund beispielsweise Trojaner (siehe Seite 9) in einer großen Sequenz einzelne Muster erkannt werden sollen, die sich vom Rest abheben und nicht durch ein Durchschnitt-Pooling neutralisiert werden sollten, wird wie auch in [67, S. 5] vorgeschlagen kein reines Durchschnitt-Pooling verwendet und daher hier auch nicht getestet. Da, wie in Abbildung 4.2 zu erkennen ist, jedoch sowohl Durchschnitt-Pooling als auch Max-Pooling in manchen Situationen nicht zum erwünschten Ergebnis

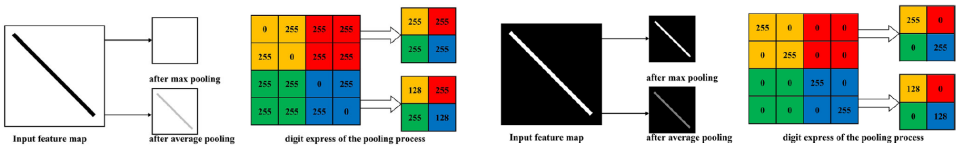

Abbildung 4.2: Sowohl Durchschnitt- als auch Max-Pooling versagen in manchen Situationen. Von [2, S. 881]

führen [2, S. 880f], wurde als Pooling-Option neben Max-Pooling auch Durchschnitt- und Max-Pooling parallel geschaltet in die Untersuchung mit aufgenommen. Diese wird im Folgenden als AvgMax-Pooling bezeichnet.

Da dadurch jedoch doppelt so viele Kanäle entstehen, wird anschließend die Anzahl der Filter wieder mit einer punktuellen Faltungsschicht auf die vorherige Anzahl gebracht.

Alle Modelle wurden mit zehntausend Sequenzen und einer Batch-Größe von 512 250 Epochen trainiert und anschließend mit zweitausend neuen Sequenzen evaluiert.

Anhand des Experiments konnte—wie den Tabellen 4.1 und 4.2 zu entnehmen ist—festgestellt werden, dass CNNs mit zwei Schichten CNNs mit nur einer Schicht stark überlegend zu sein scheinen. Auch eine Pooling-Schicht erwies sich in diesem Experiment als anderen Schichten unterlegend. Die verschiedenen Faltungsschichten als gelerntes Pooling erzielten genauso wie die Variante der erhöhten Schrittweite teils bessere Ergebnisse, jedoch nur bei bestimmten Konfigurationen. Diese Ergebnisse decken sich mit der Feststellung von Chen, Dai u. a. [12], welche festgestellt haben, dass die Verwendung von DynConvs einen größeren positiven Effekt hat, wenn diese in späteren Schichten anstelle von ausschließlich den in frühen Schichten verwendet werden [12, S. 6]. Dieses Ergebnis kann schlussendlich nicht nur so gedeutet werden, dass DynConvs für einen wirklichen Mehrwert Schichten vorher benötigen, sondern deutet eben auch auf die entscheidende Rolle weiterer Faltungsschichten hin. Bei der Erkennung von Wertpaaren kann zudem ausschlaggebend sein, dass bei zwei Schichten auch in der Theorie deutlich einfacher bestimmte Werte erkannt werden können. So könnte beispielsweise die erste Schicht erkennen, ob es Werte gibt, die mindestens so groß sind wie der Schwellwert, und die zweite Schicht könnte dann erkennen, ob die Werte auch nicht zu groß sind. Wie die trainierten Modelle die Wertpaare aber schlussendlich erkannt haben, wurde nicht betrachtet und wird vermutlich anders sein als das vorherige Beispiel. Prinzipiell können einzelne erkannte Merkmale auch über die anschließende vollständig verbundene Schicht miteinander kombiniert werden, jedoch hat eine weitere Faltungsschicht auch noch den Vorteil, dass sie zu weiterer Abstraktion

führen und auch die Abstände und Reihenfolge der Merkmale berücksichtigen kann [48, S. 5]. Aufgrund möglicher Sprünge könnte die Reihenfolge der Merkmale jedoch auch unwichtig oder gar irreführend sein.

Fügt man des Weiteren eine punktuelle Faltungsschicht direkt nach der ersten Schicht hinzu, so lässt sich eine Verbesserung fast aller Modelle feststellen (siehe Spalte *Genauigkeit mit extra* $1 \cdot k$ in Tabelle 4.1 und 4.2). Dabei lässt sich erkennen, dass die Modelle mit einer klassischen Pooling-Schicht am zuverlässigsten bessere Ergebnisse erzielten. Mit dieser zusätzlichen $1 \cdot 1$-Faltungsschicht erzielen jedoch auch Modelle mit einer erhöhten Schrittweite ab der zweiten Schicht akzeptable Ergebnisse. Reduziert man jedoch in diesen gelernten Pooling-Schichten die Konnektivität (SepConv und DepConv), so sind die Ergebnisse weniger gut. Wie im Vergleich zwischen Pooling = Nein mit S_2= 2 und Pooling = Conv zu erkennen ist, ist eine zu große Anzahl an Parametern ebenfalls bei diesem Experiment nicht förderlich. Ebenso erzielten auch die Modelle mit einer erhöhten Schrittweite in der ersten Schicht unterdurchschnittliche Ergebnisse. Bei diesem Experiment erreichten zudem generell Modelle mit einer kleinen Filtergröße in der ersten Schicht deutlich bessere Ergebnisse als Modelle mit größeren Filtergrößen. Dies wird hier aber auch an den nur kleinen zu erkennenden Mustern liegen, weshalb diese Erkenntnis nicht direkt auf das eigentliche Experiment übertragen werden kann. Da bei der Erkennung von Malware beispielsweise möglichst nur etwas größere Muster erkannt werden sollen, und nicht das Vorhandensein einer bestimmten Anweisung oder DLL-Funktion ausschlaggebend sein sollte, wird beim Basismodell auf die Verwendung der Filtergröße 2 verzichtet. Was jedoch festgehalten werden kann, ist, dass bereits kleine Änderungen größere Sprünge in der Genauigkeit bringen können, und dabei keine echte Korrelation festgestellt werden kann. Einzig die Tatsache, dass größere Netze bei diesem Experiment meist schlechtere Ergebnisse liefern, kann als Hinweis auf die Notwendigkeit einer höheren Abstraktion interpretiert werden, da es bei größeren Modellen schneller zu Overfitting kommt und diese auch generell teils schlechter die eigentlichen Merkmale aufnehmen. Hinsichtlich der echten Pooling-Schichten kann zudem erkannt werden, dass Max-Pooling teils bessere Ergebnisse liefert als AvgMax-Pooling, im Durchschnitt letzteres jedoch unterschiedliche Netzgrößen besser ausgleicht und so im Durchschnitt besser als Max-Pooling ist. Im Gegensatz zu den Erkenntnissen von [80] führte die Erhöhung der Schrittweite auf 2 in einer späteren Schicht bei diesem Experiment nicht zu besseren Ergebnissen als eine klassische Pooling-Schicht, doch die erreichte Genauigkeit und Stetigkeit ist zumindest auch hier sehr ähnlich. Das in Tabelle 4.2 mehr schlechtere Ergebnisse zu erkennen sind liegt daran, dass bei den Modellen mit einer erhöhten

A_1	G_1	S_1	Pooling	A_2	G_2	S_2	Acc	Mit extra $1 \cdot k$		
								Acc	Params	MACs
32	4	1	Nein	-			0,469	0,534	867	2,14e-5
32	2	1					0,408	0,603	739	1,93e-5
32	6	1					0,445	0,513	995	2,30e-5
64	2	1					0,434	**0,796**	1411	3,83e-5
32	2	1	Nein	32	2	1	**0,709**	**0,786**	1843	4,84e-5
32	4	1		32	2	1	0,550	0,529	1971	4,85e-5
32	2	1		32	4	1	0,664	**0,771**	2867	7,30e-5
32	6	1		32	2	1	0,503	0,496	2099	4,80e-5
32	2	1		18	6	1	0,626	0,679	2491	6,09e-5
32	2	1	Max	32	2	1	0,531	**0,790**	1843	3,32e-5
32	4	1		32	2	1	0,501	0,513	1971	3,42e-5
32	2	1		32	4	1	0,513	**0,717**	2867	4,23e-5
40	2	1		40	2	1	0,537	**0,713**	2299	4,13e-5
40	2	1		32	3	1	0,496	**0,766**	2523	4,30e-5
32	2	1	DepConv	32	2	1	0,508	0,550	1907	3,36e-5
32	4	1		32	2	1	0,490	0,530	2035	3,46e-5
32	2	1		32	4	1	0,518	0,545	2931	4,28e-5
48	2	1		32	2	1	0,499	0,551	2243	4,31e-5
40	2	1		32	3	1	0,5	**0,721**	2587	4,35e-5
32	2	2	Nein	32	2	1	0,552	0,602	1843	2,46e-5
32	3			32	2	1	0,537	0,551	1907	2,38e-5
32	3			32	4	1	0,505	0,530	2931	3,29e-5
32	4			32	2	1	0,519	0,498	1971	2,47e-5
32	2			32	4	1	0,502	0,567	2867	3,48e-5
56	2			48	2	1	0,574	0,613	2923	3,93e-5
40	2			40	4	1	0,503	0,576	3579	4,35e-5

Tabelle 4.1: Experiment zur Erkennungskraft einfacher CNNs. $A_X :=$ Anzahl der Filter in Hauptschicht X, $G_X :=$ Filtergröße in Hauptschicht X. Die entscheidenden Gruppen sind durch mehrzeilige Spalten gekennzeichnet. Die absolut 20% höchsten erreichten Genauigkeiten sind fettgedruckt und die jeweils 20% höchsten Genauigkeiten der zwei Spalten sind grau hinterlegt. (Teil 1 von 2)

A_1	G_1	S_1	Pooling	A_2	G_2	S_2	Acc	Mit extra $1 \cdot k$		
								Acc	Params	MACs
32	2	1	Nein	32	2	2	0,558	**0,751**	1843	3,39e-5
32	3	1		32	2		0,542	0,532	1907	3,50e-5
32	3	1		32	4		0,515	0,516	2931	4,73e-5
32	4	1		32	2		0,502	0,523	1971	3,50e-5
32	2	1		32	4		0,616	**0,699**	2867	4,61e-5
40	2	1		48	2		0,687	**0,771**	2587	4,59e-5
40	2	1		32	3		0,619	**0,751**	2523	4,58e-5
32	2	1	Conv	32	4	1	0,681	0,567	6483	8.63e-5
32	2	1		32	3	1	0,558	**0,802**	5459	7,81e-5
32	2	1		16	6	1	0,520	0,6670	5395	6.87e-5
32	2	1	SepConv	32	2	1	0,560	0,617	3459	5,42e-5
32	4	1		32	2	1	0,527	0,636	3587	5,37e-5
32	2	1		32	4	1	0,520	0,542	5507	7,26e-5
32	2	1	AvgMax	32	2	1	0,515	**0,754**	2371	4,08e-5
32	4	1		32	2	1	0,492	0,692	2499	4,13e-5
32	2	1		32	4	1	0,516	**0,755**	3395	5,00e-5

Tabelle 4.2: Experiment zur Erkennungskraft einfacher CNNs. $A_X :=$ Anzahl der Filter in Hauptschicht X, $G_X :=$ Filtergröße in Hauptschicht X. Die entscheidenden Gruppen sind durch mehrzeilige Spalten gekennzeichnet. Die absolut 20% höchsten erreichten Genauigkeiten sind fettgedruckt und die jeweils 25% höchsten Genauigkeiten der zwei Spalten sind grau hinterlegt. (Teil 2 von 2)

Schrittweite weitere Varianten mit größeren Filtergrößen in der ersten Schicht untersucht wurden.

Für das Basismodell kann also neben dem Vorteil einer weiteren Schicht auch die Wichtigkeit von verschieden großen Filtern festgehalten werden, da man nicht weiß, welche Größe am besten geeignet ist, wobei am Anfang die Filtergrößen —wie auch in [81, S. 3] vorgeschlagen wird —eher kleiner sein sollten.

Auch relevante Anweisungssequenzen zur Erkennung von Malware können ganz unterschiedlich lang sein, und allein um Sequenzen mehrerer Längen besser erkennen zu können, sollte nicht nur eine Filtergröße verwendet werden. Kleinere Filtergrößen am Anfang können dabei beispielsweise zudem erstmals ähnliche Muster zusammenfassen. Anzumerken ist hier jedoch, dass dennoch beispielsweise Krčál, Švec u. a. [49] und Raff, Barker u. a. [67] mit nur einer einzigen recht großen Filtergröße von 32 beziehungsweise 512 relativ gute Ergebnisse erzielen konnten. Weiter kann für das Basismodell die Wichtigkeit des Vorhandenseins einer punktuellen Faltungsschicht und des Vorhandenseins von mindestens zwei nicht punktuellen Faltungsschichten festgehalten werden. Kurz: das Netz sollte über verschieden große Filter die Möglichkeit haben, viele Informationen verarbeiten zu können und dabei ohne Informationsverlust gezwungen werden, Mustern verallgemeinern zu müssen. Letzteres ist dabei auch vorteilhaft für die Umsetzung, da so weniger Rechenleistung und Speicher benötigt wird.

Zu berücksichtigen bleibt jedoch, dass die Erkenntnisse von einem Kontext und Modell nicht direkt allgemeingültig sind.

4.1.3 Das Referenz- und Basismodell

Um diese Ziele im Basismodell umsetzen zu können, wurde die Idee des Inception-Moduls mit Dimensionsreduktion von Szegedy, Liu u. a. [81] (siehe Abbildung 4.3) aufgegriffen. Die einzelnen Faltungsschichten aus der Mnemonics basierten HYDRA-Komponente werden ohne globales Pooling konkateniert, sodass alle folgenden Schichten auf alle Informationen vorheriger Schichten zugreifen können. Da davon auszugehen ist, dass an vielen Positionen mehrere verschiedene „cluster“ an relevanten Daten sich überlagern, scheint eine punktuelle Faltung auch neben verbesserter Performance sinnvoll zu sein [81, S. 3f]. Die Performance wurde dadurch verbessert, dass über punktuelle Faltungen die Anzahl der Kanäle reduziert wird. Größere Filter haben dadurch deutlich weniger Parameter, wodurch weniger Berechnungen durchgeführt werden müssen. Um dabei weiterhin Muster

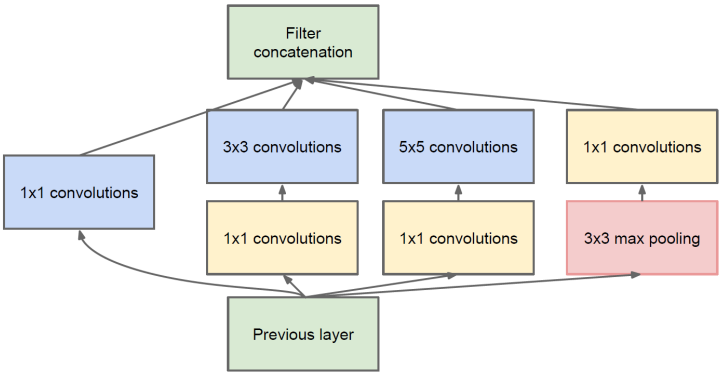

Abbildung 4.3: Inception-Modul mit Dimensionsreduktion. Von [81]

verschiedener Größen effektiv erkennen zu können, wird anschließend wieder mit etwas größeren Filtern gearbeitet. Für das Basismodell wurden die in Abbildung 4.3 zu erkennenden Schichten des Inception-Moduls aus [81] verwendet. Zur weiteren Vergrößerung des rezeptiven Feldes wurde aufgrund des Erfolges anderer Netze zur Erkennung von Malware mit größeren Filter wie [67, 49, 33] ein weiterer Pfad mit einer Faltungsschicht mit einer Filtergröße von 27 hinzugefügt.Um dabei jedoch nicht die Anzahl der Parameter und den Rechenaufwand stark zu erhöhen, wurde dafür anstelle einer normalen Faltungsschicht eine SepConv verwendet, die in Abschnitt 4.1.2 beschrieben ist. Zudem wurde vor dieser SepConv-Schicht ein schichtenübergreifendes Max-Pooling hinzugefügt, welches an jeder Stelle das Maximum aus je 2 Schichten bestimmt. Auf Grundlage des zuvor beschriebenen Experiments wurden zudem bei allen Pfaden, die auch nicht punktuelle Faltungsschichten haben, nach den punktuellen Faltungsschichten jeweils eine der zuvor drei besten Subsampling-Möglichkeiten hinzugefügt. Um weiterhin mit unterschiedlichen Größen an PEs arbeiten zu können und um die Invarianz gegenüber der Positionen der Muster beizubehalten, wird jeder Pfad wie auch zuvor im Ausgangsmodell mit einer globalen Max-Pooling-Schicht beendet, bevor die einzelnen Pfade wieder konkateniert werden. Um mehr Möglichkeiten für die Kombination der Daten zu ermöglichen, wurde wie bei den Modellen von [74] und [33] neben der Ausgabeschicht noch eine weitere vollständig verbundene Schicht hinzugefügt. Um Overfitting entgegenzuwirken, wurde zudem nach der ersten Konkatenation eine Spatial-Dropout-Schicht [83] mit einer Dropoutrate von 0,2 und nach der Konkatenation der globalen Pooling-Schichten eine Dropoutschicht mit einer Dropoutrate von 0,5 hinzugefügt. Um weiter die Rechenleistung nicht stark zu erhöhen, wurde die Anzahl der Filter vom Referenzmodell von jeweils 100 auf jeweils 72 beziehungsweise 64 bei der Schicht, die eine Filtergröße von 4 hat, reduziert.

Da eine Maxpooling-Schicht vor einer globalen Max-Pooling-Schicht keinen Unterschied außer einen kleinen Informationsverlust und eine Effizienzsteigerung im Vergleich zum Pfad, welcher ausschließlich aus einer punktuellen Faltungsschicht besteht, hat, wurde zur weiteren Reduzierung des Rechenaufwandes letzterer Pfad entfernt.

Insgesamt hat dieses Modell durch diese in Abbildung 4.4 dargestellte Netzstruktur die Möglichkeit, sowohl sich nicht überlagernde Informationscluster aus der ersten Schicht zu kombinieren, als auch direkt die erkannten Merkmale aus der ersten Schicht in möglicherweise angepasster Form direkt an die vollständig verbundene Schicht durchzureichen. Mit einem rezeptiven Feld vor dem globalen Pooling von 3 bis 59 kann das Netz dabei viele verschieden große Merkmale erkennen und verarbeiten[3]. Die punktuelle Faltungsschichten erzwingen durch eine Verringerung der Kanäle für eine bessere Performance aufgrund einer Informationsverdichtung auch weitere Abstraktion. Dabei hat jeder Pfad eine eigene punktuelle Faltungsschicht, damit jeder Pfad potenziell unterschiedliche Informationen aus den vorherigen konkatenierten Daten verarbeiten kann, um einen Informationsverlust durch die Verdichtung zu vermeiden.

Da dieses Modell jedoch in der Praxis nach 15 Epochen —abgesehen von bei hohen Grenzwerten für eine geringe Falsch-Positiv-Rate (FPR)—keine besseren Ergebnisse als das Referenzmodell erzielte und um die untersuchten Techniken an mehreren Modellarten zu testen, wurden für die schlussendlichen Untersuchungen sowohl das ursprüngliche Referenzmodell, das beschriebene erweiterte Basismodell als auch vereinfachte Varianten dieses Basismodells als Modellgrundlagen verwendet. Vereinfacht bedeutet konkret, dass bei diesem Modell auf den extra Pfad mit der SepConv-Schicht mit der Filtergröße 27 verzichtet wurde und dafür die Anzahl der Filter ab der zweiten Schicht leicht erhöht wurde. Dieses wird im Folgenden als *SimpBase* bezeichnet. Die im vorherigen und diesem Abschnitt beschriebene Modellgrundlage wiederum werden entsprechend als *Ref* und *Base* bezeichnet. Letztere beiden wurden sowohl mit als auch ohne der extra vollständig verbundenen Schicht vor der Ausgabeschicht verwendet, wobei in den Base-Modellen die zusätzliche vollständig verbundene Schicht aus 256 Neuronen und in den SimpBase-Modellen aus 128 Neuronen besteht. Die Modellbezeichnungen, in denen diese extra Schicht verwendet wurde, enthalten den Zusatz *Dense*. Der genaue Aufbau des SimpBaseDense-Basismodells ist in Abbildung 4.5 dargestellt. Insgesamt wurden also fünf Verschiedene Basismodelle verwendet, in welche jeweils DynConvs und MTL integriert wurden.

[3] Die Größe des rezeptiven Feldes r auf der Schicht l kann über die Formel $r_{l-1} = s_l * r_l + (k_l - s_l)$ anhand der Schrittweiten s und der Filtergröße k berechnet werden[vgl. 5].

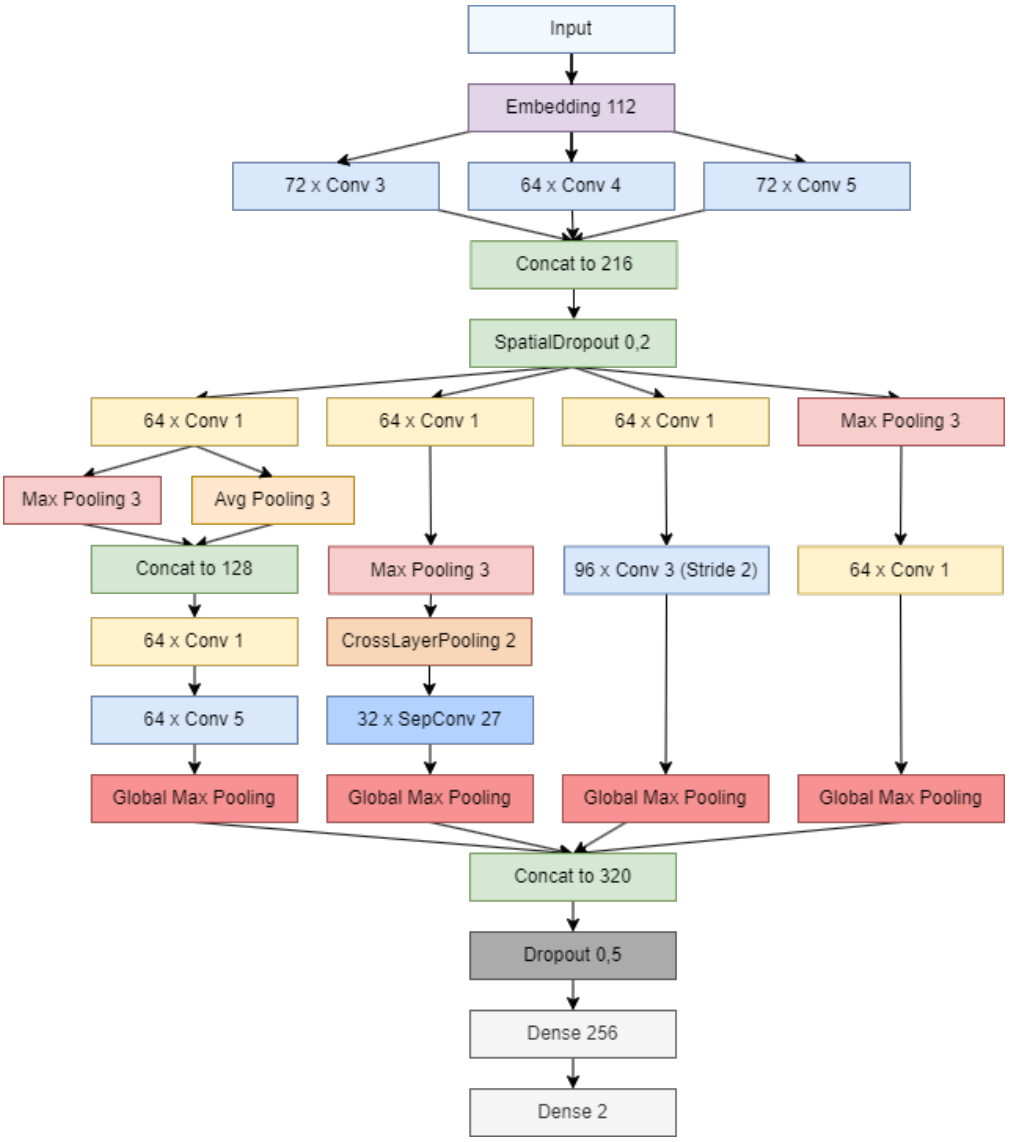

Abbildung 4.4: Struktur des vollständigen Basismodells *BaseDense*. Eigene Darstellung mit Inspiration für das Farbschema von [81]

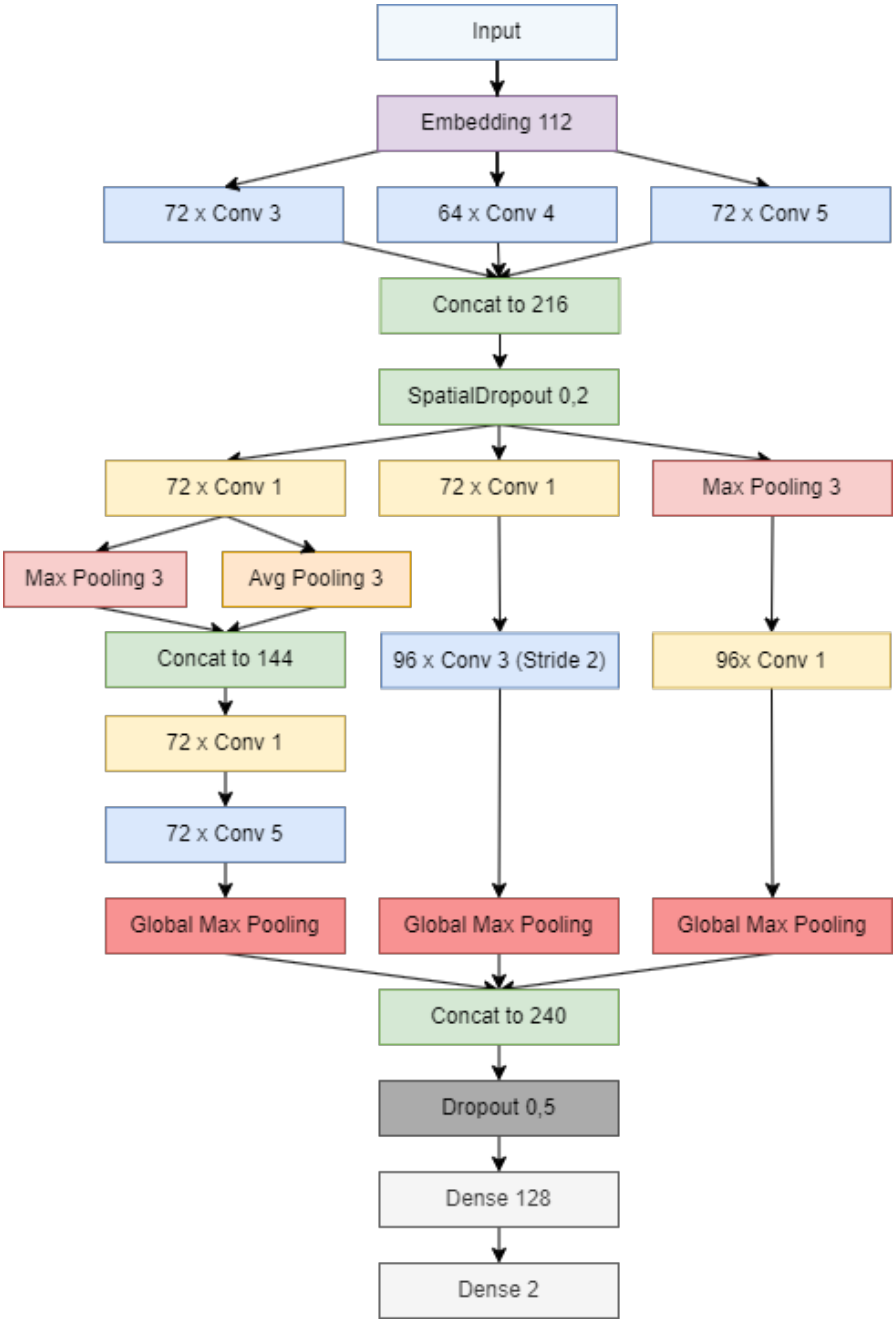

Abbildung 4.5: Struktur des vollständigen vereinfachten Basismodells *SimpBaseDense*. Eigene Darstellung mit Inspiration für das Farbschema von [81]

4.1.4 Experimentelle Modelle

Dynamisches Faltungsnetz

Der erste Ansatz zur Verbesserung der merkmalslosen Erkennung von Malware ist die Verwendung der in Abschnitt 3.3.4 beschriebenen DynConvs von Chen, Dai u. a. [12]. Da es bis dato jedoch keine offizielle oder andere zufriedenstellende Implementierung dieses Netzes gibt, wurde eine eigene Implementierung verwendet. Dessen korrekte Funktionsweise wurde dafür über Tests mit festen Gewichten und über einem Vergleich zu normalen Faltungsschichten und der klassischen Mischung von Experten mit dem MNIST-Trainingsatz[4] sichergestellt. Dabei wurde festgestellt, dass vor allem in den ersten Schichten trotz der Verwendung der Temperatur τ (siehe Formel 3.4) das Modell dazu neigt, einen Experten mehr als die anderen Experten zu nutzen, wie in Abbildung 4.6 zu sehen ist. Dieses Verhalten wurde bereits unter anderem von Eigen, Ranzato u. a. für das Mischung von Experten Verfahren festgestellt [18, S. 2]. Erklärt werden kann dies damit, dass der Experte, der für die ersten Eingaben am meisten für ein richtiges Ergebnis beigetragen hat, aufgrund der Backpropagation anschließend von der Attention beziehungsweise der Gewichtungsfunktion bevorzugt wird [18, S. s]. Dadurch, dass diese Schicht nun häufiger von der Gewichtungsfunktion bevorzugt wird, wird diese Schicht anschließend stärker trainiert, wodurch diese Schicht aufgrund folgender besserer Ergebnisse wiederum erneut von der Attentionschicht stärker bevorzugt wird [18, S. s]. Wie auf der rechten Seite in der Abbildung 4.6 zu sehen ist, kam es bei diesem Experiment jedoch nicht zu dieser Endlosschleife, da von Anfang an immer auch die anderen Experten etwas mitverwendet wurden. Bei einem Test ohne der Temperatur konnte dabei keine markante Änderung dieses Verhaltens festgestellt werden. Mögliche Gründe, dass zwar wie in der Theorie ein Experte mehr genutzt wurde als die anderen, eine endlose Steigerung dieses Verhaltens jedoch ausblieb, ist, dass das Modell schon recht schnell eine hohe Genauigkeit aufwies und so der beschriebene Effekt etwas abgeschwächt ist.

Um eine stärkere Balancierung der Nutzung und gleichzeitig eine wirkliche Spezialisierung zu erhalten, können sowohl weiche als auch harte Bedingungen angewendet werden [75, S. 5]. Da bei der Untersuchung der Funktionsweise bereits ab der zweiten Schicht eine Spezialisierung der Experten festgestellt werden kann, und auch von späteren Arbeiten wie [75] und [Bengio et al. (2015) zitiert von [75]] weiche Bedingungen verwendet wurden,

[4] MNIST ist ein Datensatz aus 60.000 Trainings- und 10.000 Test-Bildern von handgeschriebenen Ziffern in der Größe 28x28. Verfügbar unter http://yann.lecun.com/exdb/mnist/

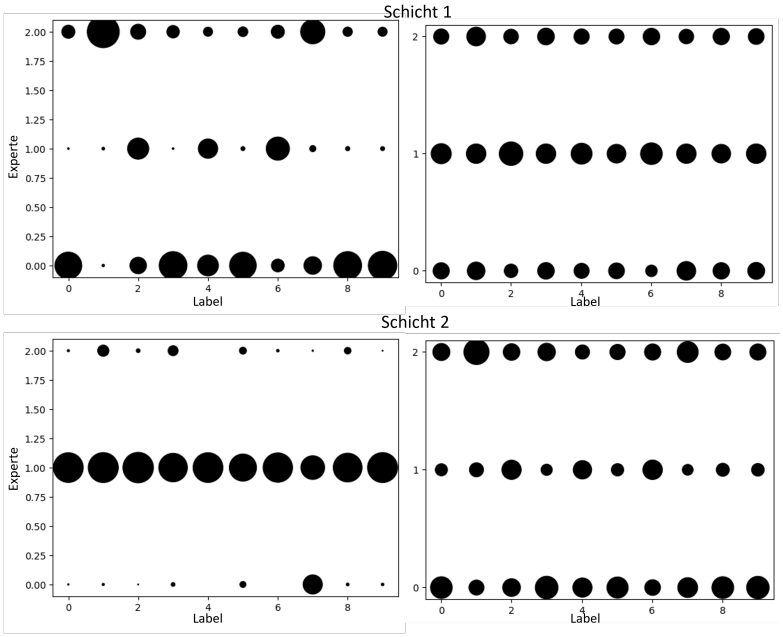

Abbildung 4.6: Ungleiche Nutzung der Gewichtsmatrizen. Die linken Grafiken zeigen, wie häufig die einzelnen Gewichtsmatrix die höchste Gewichtung bekommen haben, und die richten Grafiken stellen die absoluten Summen der einzelnen Attention-Faktoren $\pi(X)_k$ dar.

wurde auch in dieser Arbeit eine weiche Bedingung verwendet. Diese hat den Vorteil, dass dabei die Stärke der Bedingung besser eingestellt werden kann und das Erreichen einer hohen Genauigkeit weiterhin das höherwertige Ziel bleibt. Umgesetzt wird diese weiche Bedingung durch eine oder mehrere zusätzliche Verlustfunktionen, die auf die bereits vorhandene Verlustfunktion aufaddiert werden [75, S. 5]. Konkret werden zwei zusätzliche Verlustfunktionen verwendet, die in den Formeln 4.1 und 4.3 dargestellt sind: $L_{Balance}$, um das Netz dazu zu bringen, alle Experten zu nutzen, und $L_{Spezialisierung}$, damit das Netz durch die vorherige Funktion nicht alle Experten auf einmal benutzt. $L_{Balance}$, welche in [75] vorgestellt wurde, berechnet den Verlust anhand des Quadrats des Variationskoeffizienten (VC= Standardabweichung durch Mittelwert) der Summen für die einzelnen Ausgaben eines Batchs:

$$L_{balance} = VC(\sum_{x \in X} \hat{Y}_x)^2 \tag{4.1}$$

mit $X :=$ alle Eingaben des Batchs X und $\hat{Y}_x :=$ Ausgabevektor für Eingabe x. Über $L_{Spezialisierung}$, welche L_{Load} aus [75] ersetzt, soll das gleiche Ergebnis wie von L_{Load}

erzielt werden, nämlich, dass es unwahrscheinlicher wird, dass manche Experten selten viel und andere Experten häufig etwas verwendet werden [75, S. 6], doch über einen anderen Ansatz. Während in [75] über ein komplexeres Verfahren mit Rauschen, Zufall und zwischengespeicherten Matrizen versucht wird, dass alle Experten annähernd gleich oft verwendet werden, wird über $L_{Spezialisierung}$ explizit angestrebt, dass bei jeder Eingabe fast nur ein Experte verwendet wird. Der Hauptgrund für die umständliche Definition von L_{Load} in [75] ist eine optimale Ausnutzung verteilter Hardware [75, S. 6], was aufgrund Kombinierung der Gewichte anstelle der Ergebnisse mehrerer Experten für DynConv kein Effekt und keine Relevanz mehr hat. Der explizite Zwang zur Spezialisierung könnte zwar möglicherweise die Repräsentierbarkeit des Modells geringfügig reduzieren, erhöht gegebenenfalls aber die Erklärbarkeit des Modells, da die Wahl des Experten möglicherweise Hinweise liefern könnte. Der Grundgedanke hinter der Definition von $L_{Spezialisierung}$ ist, dass bei der Softmax-Aktivierung die Spezialisierung am stärksten ist, je weiter die einzelnen Aktivierungen vom Mittelpunkt —also $\frac{1}{K}$ mit $K :=$ Anzahl der Experten—entfernt sind. Damit der Verlustwert bei größerer Abweichung abnimmt, muss die Summe der ermittelten Abweichungen nun vom maximalen Wert für die gegebene Anzahl an Experten abgezogen werden. Die maximale Abweichung ist immer dann gegeben, wenn ein Wert 1 und alle anderen Ausgabewerte 0 sind, was bedeutet, dass $K-1$ mal eine Abweichung von genau dem Durchschnitt vorhanden ist und einmal eine Abweichung von $1-1/K$. Zusammen ergibt dies $2-\frac{2}{K}$:

$$(K-1)\cdot\frac{1}{K}+1-\frac{1}{K}=1-\frac{1}{K}+1-\frac{1}{K}=2-\frac{2}{K} \tag{4.2}$$

Damit diese Verlustfunktion ebenfalls unabhängig von der Batchgröße ist, wird noch über alle Werte eines Batches der Mittelwert gebildet, bevor auch hier der Wert für ein besseres Lernverhalten quadriert wird. Zusammen ergibt dies:

$$L_{Spezialisierung}=(\sum_{x\in X}\frac{2-\frac{2}{K}-\sum_{\hat{y}\in\hat{Y}_x}\|\hat{y}-\frac{1}{K}\|}{\|X\|})^2 \tag{4.3}$$

Um keinen Konflikt zur genutzten Temperatur für die gleichmäßige Nutzung der Experten in den ersten Epochen zu haben, wurden beide Verluste anschließend noch durch die aktuelle Temperatur geteilt. Zusätzlich wurde experimentell ein Gewichtungsfaktor von 0,2 für beide Verlustfunktionen ermittelt, durch welche diese Verluste nicht das eigentliche Lernziel überdecken und dennoch ihr gewünschtes Ziel hervorbringen. Bei gleicher Skalierung hat $L_{Balance}$ immer prinzipiell einen höheren Maximalwert, doch beim MNIST

Datensatz und 3 Experten führt bereits ein Unterschied zwischen beiden Skalierungen von $0,01$ zu einem schlechteren Resultat bezogen auf die Nutzung der Experten. Bezogen auf die Genauigkeit wiesen alle 4 untersuchten Modelle (ohne DynConvs, mit und ohne extra Verlustfunktionen und mit klassischem Mischung von Experten Verfahren) eine Genauigkeit von $0,985 \pm 0,004\%$ auf, ohne dass ein Modell nach mehreren Trainings signifikant häufiger besser als die anderen Modelle war. Bei den Untersuchungen wies zwar am häufigsten das Modell ohne die Verwendung von Experten die geringste Genauigkeit auf, schlussendlich lässt sich aber dennoch festhalten, dass auch das Lernen bei der eigenen Implementierung vom dynamischen Faltungsnetz nach [12] funktioniert. Die durch die Verwendung der zusätzlichen Verlustfunktionen erreichte Spezialisierung ist in Abbildung 4.7 dargestellt.

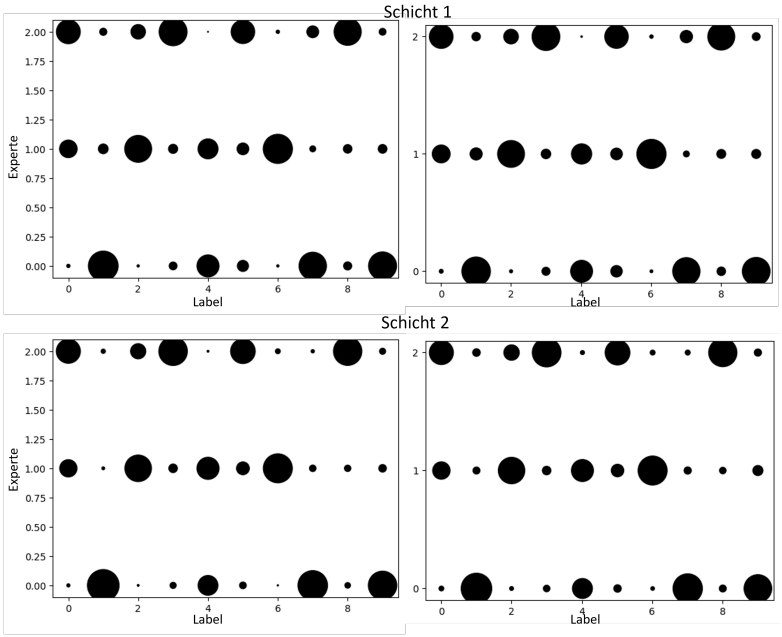

Abbildung 4.7: Durch zusätzliche Verlustfunktionen erreichte Spezialisierung der Experten. Die linken Grafiken zeigen, wie häufig die einzelnen Gewichtsmatrix die höchste Gewichtung bekommen haben, und die richten Grafiken stellen die absoluten Summen der einzelnen Attention-Faktoren $\pi(X)_k$ dar.

Da diese DynConvs direkt normale Faltungsschichten ersetzen können, unterscheidet sich das Modell für diese Schicht nur darin, dass Faltungsschichten durch ein DynConv mit den zuvor definierten Verlustfunktionen ersetzt wurden. Da sich die Genauigkeit nicht erkennbar weiter steigen solle, sobald mehr als vier Experten beziehungsweise kernel verwendet werden [12, S. 6], wurde auch hier die Anzahl der Experten auf nicht über vier

festgelegt. Um jedoch die Anfälligkeit für ein Overfitting zu reduzieren, wurde für diese Arbeit die Anzahl auf drei festgelegt.

MTL-Modell

Die zwei Varianten mit dem MTL unterscheiden sich vom Basismodell beziehungsweise vom Modell mit der dynamischen Faltungsschicht nur in der Anzahl der Ausgabeneuronen und der Verlustfunktion.[5] Damit das Modell weiterhin den Fokus auf die Malware-Erkennung legt und nicht zu stark von den weiteren Zielen beeinflusst wird, werden die hinzugefügten Ausgabeneuronen in der vollständigen Verlustfunktion weniger gewichtet.

Da in dieser Arbeit für das MTL nur Verhaltenslabel für Malware, nicht aber für gutartige PEs vorhanden sind (siehe Abschnitt 4.2.3), wird für dieses Modell eine Verlustfunktion benötigt, die mit fehlenden Label umgehen kann. Da durch den verwendeten Datensatz es weitere konkrete Rahmenbedingungen für diese Verlustfunktion gibt, wird im Folgenden erst der Datensatz beschrieben.

4.2 Verwendeter Datensatz

4.2.1 Gutartige Software des Datensatzes

Aus Urheberrechtsgründen gibt es fast keine öffentlichen Datensätze, in denen eine breite Menge an gutartiger Software enthalten ist. Aus diesem Grund stammen die gutartigen PEs vom eigenen Rechner und zusätzlich über den Windows-Paketmanager winget heruntergeladene Software. Da das nachträgliche Analysieren der einzelnen Anwendungen zu lange dauern würde und zudem bei interaktiven Applikationen sowieso eine automatische vollständig Untersuchung in der Praxis nur sehr schwierig bis unmöglich durchzuführen wäre, gibt es für die gutartigen Anwendungen des Datensatzes keine Verhaltenslabel. Für das Sammeln der PEs vom eigenen Rechner und der virtuellen Maschine, auf welcher 1200 Pakete über winget heruntergeladen wurden, wurde Everything von Voidtools[6]

[5] In Keras-Tensorflow lässt sich das Modell prinzipiell auch so umsetzen, dass die zusätzlichen Ausgabeneuronen nach dem Training entfernt werden können, doch da sowohl Haupt-, als auch Nebenziel anschließend ausgewertet werden und es im Rahmen dieser Arbeit keinen Vorteil für diesen modularen Aufbau gibt, wurde darauf in dieser Arbeit verzichtet.

[6] Verfügbar unter https://www.voidtools.com

verwendet. Über dieses Tool wurden alle auf dem System vorhandenen .exe, .dll, .sys und .drv Dateien gesammelt. Damit das in Raff, Barker u. a. [67, S. 3] beschriebene Szenario, dass das Modell schlussendlich nur Software von Microsoft von Software, die nicht von Microsoft ist, unterscheidet, nicht auftritt, wurden keine .exe oder .dll unter C:\Windows oder einem dessen Unterverzeichnissen wie System32 verwendet. Um den geringen Anteil an .sys und .drv Dateien nicht noch weiter zu reduzieren, wurden diese trotzdem auch aus C:\Windows mit in den Datensatz aufgenommen.

Winget eignet sich für das Beschaffen weiterer Trainingsdaten, da die dort verfügbare Software von mehreren Virenscannern überprüft wurde [44]. Um eine große Spanne verschiedener Anwendungen zu sammeln, wurde die Webseite https://winstall.app verwendet, indem sowohl 900 Identifikatoren aus den von Nutzern erstellten Sammlungen als auch 300 Identifikatoren von der alphabetischen Auszählung für die Installation von Paketen gesammelt wurden.

4.2.2 Erkennung ähnlicher Daten

Da viele PEs, insbesondere DLLs sehr häufig auf einem System vorkommen, wird ein Verfahren angewendet, um PEs nicht mehrmals in den Datensatz aufzunehmen. Allein den Namen zu vergleichen wäre jedoch zu restriktiv, da einerseits ganz verschiedene Dateien denselben Namen haben können und andererseits es auch innerhalb gleichen Programms aufgrund eines anderen Compilers oder eines großen Versionssprungs größere Unterschiede geben kann. Haq und Caballero [39] bietet einen Überblick über mögliche Methoden, wobei jedoch N-Gramm, Funktionsgraph oder semantische Ansätze zu zeitintensiv sind. Auch Ansätze, die auf ML basieren und nicht trainiert bereitgestellt werden, werden aus zeitlichen Gründen nicht berücksichtigt. Da potenziell viele Dateien verglichen werden und zudem der Vergleich über mehrere Läufe auf unterschiedlichen Geräten funktionieren sollte[7], ist eine kleine speicherbare Repräsentation ebenfalls von Vorteil. Aus diesem Grund sind auch Ähnlichkeitsmaße wie der Jaccard-Koeffizient oder der (Damerau-)Levenshtein-Abstand ebenfalls nicht nutzbar, auch da diese viel zu langsam sind. Geeignet sind daher auf Merkmale basierte Ansätze und Locality Sensitive Hashing (LSH). Letzteres sind Hashing-Verfahren, bei denen kleine Änderung im Gegenstand, über welchen der Hashwert gebildet wird, möglichst nur zu kleinen oder keinen Änderungen im Hashwert führen [63, S. 1]. Anhand der Abweichung zweier Hashwerte

[7] Es wurden sowohl PE vom eigenen Rechner, als auch weitere auf einer VM über winget installierte PE gesammelt.

kann dann ungefähr bestimmt werden, wie stark sich zwei Objekte ähneln, wobei dieser Wert nicht komplett zuverlässig ist und es zu Falsch-Positiven kommen kann. Im Falle dieser Arbeit ist es jedoch nicht schlimm, wenn ein paar Dateien zu viel aussortiert werden.

Einfach zu extrahierende und für diesen Anwendungsfall mit den zuvor festgelegtem Eingabeformat relevanten Merkmale sind die benutzten DLL-Funktionen und die Dateigröße beziehungsweise die Größe des Codes. Ob die Größe des Codes jedoch tatsächlich berücksichtigt werden sollte, ist fragwürdig, da eine Erweiterung eines Programms zwar größer sein kann, doch der Code aus dem kleineren Programm kann dennoch bereits fast vollständig vom größeren abgedeckt sein. Dadurch entsteht eine ungewollte Dopplung in den Trainingsdaten, was das Risiko für Overfitting steigern kann, wenn diese häufiger bei der gleichen Basisdatei vorkommt. Von den verschiedenen LSH-Verfahren eignet sich der Locality Sensitive Hash von TREND MICRO (TLSH) am besten, da dieser experimentell eine sehr hohe Erkennungsrate aufweist und es zudem frei verfügbare Implementierungen gibt [63, S. 4f]. Des Weiteren wird dieser bereits im Sicherheits- und Malware-Erkennungs-Bereich eingesetzt und ist zudem verhältnismäßig resistent gegenüber Codeblock-Verschiebungen [61, 62].

Da bereits andere Compiler-Einstellungen zu größeren Unterschiede im Assemblercode führen können, sind nur die Merkmale vermutlich nicht so aussagekräftig. Um Klarheit zu schaffen, wurden beide Methoden miteinander verglichen. Verwendet wurden dafür verschiedene Java-Implementierungen derselben und unterschiedlichen Versionen, verschiedene 7-Zip- und Git-Implementierungen und weitere zufällig ausgewählte Dateien ähnlicher Größe. Bei dieser Analyse wurde festgestellt, dass auch mehrere git.exe und 7z.exe sehr verschiedene Mnemonic-Sequenzen haben. Über einen manuellen Vergleich mehrerer Dateien mit Hilfe von ExamDiff Pro[8] wurde sichergestellt, dass tatsächlich nur sehr wenige übereinstimmende Sequenzen einer Länge von über 10 Mnemonics vorhanden sind.

Den TLSH anhand des Bytes der Abschnitte, die Code enthalten, zu berechnen, erfüllt zwar einigermaßen den Zweck, doch werden hier ähnliche Dateien schneller als unterschiedlich erkannt, als bei den anderen Ansätzen. Unterschiedliche Dateien wiederum wurden teils als ähnlich eingestuft. Bei einer Berechnung des Hashwertes anhand der später verwendeten Mnemonic-Sequenzen mit Funktionsauflösung werden zwar alle ähnliche Dateien als ähnlich erkannt, doch noch mehr PEs, die praktisch nichts gemeinsam

[8] Verfügbar unter https://www.prestosoft.com/edp_examdiffpro.asp

haben sollten, wurden als ähnlich eingestuft. Wird für die Hashwertberechnung nur die reine Opcode-Sequenz ohne die Funktionsnamen verwendet, ist die Genauigkeit des Filters deutlich höher. Es werden zwar mehr unterschiedliche PEs als ähnlich eingestuft, diese sind aber überwiegend PEs des gleichen Programms, die in einer manuellen Analyse große Unterschiede aufweisen. Im aktuellen Anwendungsfall ist dies jedoch besser, da diese Dateien schlussendlich auch wirklich zumindest in ihrer Semantik sehr ähnlich sind. Da die Funktionsauflösung zudem länger dauert, als die reine Extraktion der Opcode-Sequenzen, hat diese Lösung im Vergleich zum vorherigen Ansatz auch noch einen Effizienzvorteil, da nicht viel Zeit für eine Datei aufgewendet werden muss, die anschließend ignoriert wird.

Wenn man nur auf das erwartete Ergebnis schaut, dann scheidet der Vergleich der Funktionen über die folgende Formel am besten ab: $\frac{|A \triangle B|}{\max(|A|,|B|)}$, wobei A die Menge der in der einen PE referenzierten DLL-Funktionen ist, und B die Menge der referenzierten DLLs-Funktionen der anderen PE. Fragwürdig ist jedoch, wie zuverlässig dies schlussendlich in der Praxis ist, da der komplette Code der PE unbeachtet bleibt und so unter anderem Compiler-Änderungen und große Refaktorierungen teils nicht erkannt werden können. So wurde auch bei den durchgeführten Untersuchungen festgestellt, dass Abweichungen in der Opcode-Sequenz teils von dieser Metrik nicht erkannt wurden, auch wenn in den Untersuchungen dadurch bei einem etwas geringeren Schwellwert keine falsche Entscheidung getroffen wurde.

Für eine höhere Beständigkeit wurde schlussendlich daher der Durchschnitt aus dem Unterschied der TLSHs der Mnemonic-Sequenzen ohne Funktionsauflösung und der zuvor beschriebenen Formel zum Vergleich der Funktionen verwendet. Diese Kombination klassifizierte ebenfalls nur einmal eine Datei als ähnlich zu einer anderen, obwohl dies nicht der Fall sein sollte, doch lassen sich mehr kleine Unterschiede erkennen. Als geeigneter Mittelwert für die beste Balance zwischen hoher Erkennungsrate und niedriger FPR wurde 46 ermittelt, doch für eine möglichst hohe Erkennungsrate wurde schlussendlich 60 als Schwellwert gewählt, bevor beide PEs verwendet wurden.

Da einzelne Hashwerte aufgrund ihrer kleinen Größe im Gegensatz zu den teils recht großen Funktionslisten im Speicher gehalten werden können, wurde auch versucht, den Vergleich über einen TLSH der sortierten Listen durchzuführen, doch führt dieser Ansatz zu einem markant schlechteren Ergebnis. Es gibt deutlich mehr Falschklassifizierungen, wobei kleine absolute Änderungen teils zu größeren angeblichen Abweichungen führen, als eine große absolute Anzahl an Abweichungen.

Die genauen Ergebnisse der Vergleiche sind im Anhang A.1 dargestellt.

Den Hashwert von der gesamten Datei zu berechnen, wurde nicht untersucht, da beispielsweise Header und Ressourcen nicht Bestandteil der Eingaben sind und eine Änderung dessen nicht zwangsweise eine Änderung der Anweisungs-/Opcode-Sequenz zur Folge hat.

4.2.3 Malware des Datensatzes

Aktuell gibt es zwar mehrere Datensätze an Malware, wobei aufgrund des MTL mit Verhaltenslabel keiner dieser Datensätze direkt verwendet werden kann. Als Quelle der Malware und Verhaltenslabel wurde daher `capesandbox.com` ausgewählt, da diese im Internet erreichbare Instanz von der CAPEv2 Sandbox eine große Menge an Malware mit vielen verschiedenen dynamisch ermittelten Verhaltenssignaturen/-label versehen hat und diese im Zeitraum der Datensammlung auch kostenlos automatisiert abgefragt werden konnten[9]. Als Label verwendet wurden die Warnings (in gelb) und Dangers (in rot) Infos (in blau), die das Verhaltens beschreiben. Auf `capesandbox.com` werden diese als *Signatures* bezeichnet. Da die Analyse jedoch nicht immer fehlerfrei läuft, da beispielsweise manche Programme aufgrund fehlender Abhängigkeiten nicht starten, wurden nur Malware in den Datensatz aufgenommen, bei denen mindestens 3 Signaturen, die nicht vom Typ Info sind, angegeben sind.

Ähnliche Signaturen wurden nach der Extraktion zusammengefasst und Signaturen, die nichts mit dem Code der Malware zu tun haben, wie zum Beispiel, dass über dem Dateinamen vorgetäuscht wird, dass das Programm eine PDF-Datei sei, wurden entfernt. Zudem wurden auch Label, die kein spezifisches Verhalten, sondern die Erkennung einer Malwarefamilie oder das generelle Verhalten einer konkreten Malwarefamilie angeben, entfernt. Wie auch bei den Token wurden auch Verhaltenslabel, die nur selten im gesamten Datensatz vorhanden sind, entfernt, da es praktisch ausgeschlossen ist, dass das Modell bei sehr wenigen Trainingsdaten mit diesem Label das Label richtig lernen kann. Als Schwellwert wurde hier 20 gewählt. Schlussendlich wurden 126 verschiedene Verhaltenslabel verwendet.

[9] Die Malware konnte über einen API-Zugriff heruntergeladen werden, zu welchem der Zugang über Twitter angefragt wurde [9]. Die Signaturen wurden über einen Web Scraper von der Webseite extrahiert.

4.2.4 Filterung des Datensatzes

Über Nauz-File-Detector[10] wurde sichergestellt, dass es sich bei allen im Datensatz enthaltenen PEs um native, nicht in einem Laufzeitsystem laufende und nicht verpackte Programme handelt. Zur weiteren Sicherheit, dass die Anwendungen wirklich nicht verpackt sind, wurde, wenn kein Packer erkannt wurde, die Datei noch mit Detect-It-Easy[11] gescannt. Die Wahl dieser Tools wurde von Lee, Kim u. a. [55, S. 18858] übernommen.

Das Entpacken verpackter Dateien ist zwar meist möglich, doch da mit verschiedenen verfügbaren Tools keine Möglichkeit gefunden wurde, vollständig automatisch offline Malware zu entpacken, werden in dieser Arbeit nur nicht verpackte Malware verwendet. *Universal Extractor 2*[12] beispielsweise unterstützt zwar sehr viele verschiedene Packer, doch kann dieses Tool nicht vollständig automatisch ausgeführt werden, sondern erfordert häufige Nutzerinteraktion. Beispielsweise muss teils eine Entpackmethode ausgewählt werden, wenn es mehrere gibt, oder die Warnung, dass Malware gefunden wurde, weggeklickt werden. Dieses Tool unterstützt zwar einen Modus, bei welchem das Tool ohne Nutzerinteraktionen laufen solle, doch da es auch Fragen gibt, ob eine Software zum Entpacken ausgeführt werden darf, konnte dieser Modus auch nicht verwendet werden. Des Weiteren kommt die Malware-Warnung nicht von diesem Tool direkt, sondern von einem von Universal Extractor 2 verwendeten Programm. Des Weiteren müssen für das MTL alle aus einer PE entpackten PEs zusammen mit dem unverpackten Teil der verpackten PE zusammengehängt werden, da sonst die Verhaltenslabel nicht mehr stimmen. Ein zuverlässiges Ergebnis erhält man dann aber nur, wenn man nur die entpackten Anwendungen zusammenfügt, bei denen keine DotNet-Anwendungen oder Shell-Skripte wie .bat Dateien enthalten sind. Diese müssten bei der Erkennung von Malware getrennt behandelt werden, doch können mit für das zuvor erkannte Verhalten verantwortlich sein. Letzteres erschwert jedoch nur das Training, macht MTL jedoch für die Nutzung nicht weniger geeignet. Insgesamt wäre für diese Arbeit der Aufwand für das Entpacken schlussendlich also höher als der Nutzen.

Da bei Installationsprogrammen das eigentliche Verhalten ebenfalls erst bei den PEs vorhanden ist, die installiert wurden, und dann gegebenenfalls automatisch ausgeführt werden, wurden erkannte klassische Installationsprogramme ebenfalls nicht im Datensatz mit aufgenommen. Im echten Szenario müsste der Virenscanner dafür sorgen, dass

[10] Verfügbar unter https://github.com/horsicq/Nauz-File-Detector/releases
[11] Verfügbar über https://github.com/horsicq/DIE-engine/releases
[12] Verfügbar unter https://github.com/Bioruebe/UniExtract2

auch nachträglich erstellte Anwendungen erst ausgeführt werden können, nachdem sie gescannt wurden. Zur Sicherheit wurden im Rahmen dieser Arbeit jedoch keine Anwendungen, und damit auch keine Installationsprogramme ausgeführt, obwohl Malware bereits ausschließlich in einer abgesicherten und vom Internet getrennten virtuellen Maschine verarbeitet wurde. Da der Code von erkannten Installationsprogrammen trotz unterschiedlichem Namen zudem relativ ähnlich sein sollte, wurden auch im gutartigen Trainingsdatensatz keine als Installationsprogramme erkannten PE mit aufgenommen. Von CAPEv2 wurden zudem mehreren Malware die Information *Possible date expiration check, exits too soon after checking local time* zugeordnet. Da diese dadurch nicht wirklich analysiert werden konnten, wurden alle Malware, die dieser Signatur zugeordnet ist, entfernt, wenn dieser weniger als fünf weitere Warn-, Danger- oder als Verhaltenslabel verwendete Info-Signatures zugeordnet wurden.

Da teils auch Programme für eine ARM-Architektur gefunden wurden, wurde beim erneuten Sammeln schlussendlich auch sichergestellt, dass nur PEs für die Architektur AMD64 und I386 in den Datensatz aufgenommen werden.

Der Performance wegen wurden zudem nur PEs in den Datensatz aufgenommen, die verarbeitet weniger als eine Millionen Token haben.

4.2.5 Vorverarbeitung

Die PEs wurden in das in Abschnitt 4.1.1 beschriebene Format gebracht. Als Disassembler wurde dumpbin verwendet, welches in Microsoft Visual Studio[13] enthalten ist. Für die Funktionsauflösung wurde dabei eine Verfolgung von Sprunganweisungen zur Entdeckung versteckter Calls mit einer maximalen Tiefe von 5 durchgeführt. Zudem wurde anhand der letzten 110 oder bis zur vorherigen Sprunganweisung versucht, den Inhalt des Registers zu ermitteln. Wenn in der Ausgabe von dumpbin bereits die Zieladresse der Funktion angegeben ist, wurde diese direkt verwendet. Durch das Entfernen aller Symboldateien wurde dabei sichergestellt, dass in den Ausgaben von dumpbin nur Adressen und keine Funktionsnamen stehen. Eine Auflistung aller relevanter in der Vorverarbeitung verwendeter Tools und Bibliotheken ist in Anhang A.4.

[13] Verfügbar unter https://visualstudio.microsoft.com/de/downloads/

4.2.6 Augmentation

Da über die Hälfte der verarbeiteten Malware trotz eines unterschiedlichen SHA256-Hashwertes nach der Verarbeitung nach dem TLSH ein Duplikat einer anderen Malware waren, wurde zusätzlich mit Augmentation gearbeitet. Malware PEs mit doppeltem TLSH wurden aussortiert und für die übrig gebliebenen Dateien wurde pro Datensatz für die jeweils ungefähr 75% kleinsten Malware und gutartigen PEs eine augmentierte Version hinzugefügt. In der Literatur üblicherweise verwendete Augmentation für Anweisungssequenzen sind entweder das Löschen von zufälligen Anweisungen, das Abändern zu ähnlichen Anweisungen oder das Hinzufügen von zufälligen Anweisungen [7, S. 215, 58, S. 3]. Bae und Lee [7] und McLaughlin und del Rincon [58] schlagen darüber hinaus auch noch die Verwendung von verschiedenen ML-Methoden vor, über welche weitere Trainingsdaten erzeugt werden können, stellen jedoch fest, dass diese Ansätze deutlich aufwändiger sind und dafür nur für eine minimal größere oder gar zu einer geringeren Performancesteigerung führen, als die einfachen Ansätze [7, 58]. Abgesehen vom Abändern zu ähnlichen Anweisungen haben die genannten einfachen Ansätze jedoch das Problem, dass die erzeugten Daten sehr wahrscheinlich eine andere—möglicherweise zum Absturz führende—Programmlogik darstellen. Gerade bezüglich des MTL zur Erkennung des Verhaltens könnte dies nachteilhaft sein, da manche Funktionalitäten möglicherweise nur durch kleine Sequenzen zustande kommen, welche durch die Änderungen stark verändert werden könnten. Ob dies wirklich ein Problem darstellt oder doch durch die dadurch erhöhte Variabilität sogar Vorteile bringt, müsste in späteren Arbeiten überprüft werden. Um die Richtigkeit der Label beizubehalten, werden in dieser Arbeit nur einzelne Anweisungen oder Anweisungssequenzen eingefügt, bei denen es bei spezifischen Operanden immer möglich ist, dass sich der Programmfluss nicht ändert. Ein Beispiel hierfür ist `mov`, da `mov eax, eax` den Wert eines Registers auf den Wert setzt, der bereits in diesem steht. Auch beliebige Sprunganweisungen, die auf die nächste Anweisung zeigen, es also egal ist, ob der Sprung ausgeführt wird oder nicht, können problemlos der Sequenz hinzugefügt werden. Die verwendeten Sequenzen können dem Anhang A.2 entnommen werden.

An welcher Stelle welche Sequenz eingefügt wurde, wurde genauso wie der genaue Abstand der einzelnen eingefügten Sequenzen zueinander zufällig entschieden. Der Mindestabstand wurde dabei jedoch auf 5 und der maximale Abstand auf 30 festgelegt. Der tatsächliche durchschnittliche Abstand betrug 9,95, wodurch die durch die Augmenta-

Datensatz	Malware	Gutartig
Training	18.379	18.392
Validierung	1.059	1.081
Test	23.040	21.255

Tabelle 4.3: Größen der verwendeten Datensätze

tion entstandenen Daten im Durchschnitt 14,86 % mehr Token enthalten, als ihr nicht augmentiertes Gegenstück.

4.2.7 Trennung der Daten in Trainings-, Validierungs- und Testdaten

Für eine bessere Vergleichbarkeit der Modelle ist die Unterteilung des Datensatzes in Trainings-, Validierungs- und Testdaten für alle getesteten Modelle gleich. Ausschlaggebend für die Unterteilung war daher das MTL-Experiment, da im Trainings- und Validierungsdatensatz alle Label vorkommen sollten, damit das Modell diese Label lernen kann und das Gelernte analysiert werden kann. Um mehr Malware-Daten für das Training zu haben, als über Capesandbox.com in dieser Zeit heruntergeladen werden konnten[14], wurden weitere Malware von Malware Bazaar (https://bazaar.abuse.ch) ohne Verhaltenslabel für den Testdatensatz verwendet. Wie gut das Verhalten beim MTL gelabelt wird, wurde dadurch zwar nur mit dem kleinen Validierungsdatensatz untersucht, doch konnten dadurch fast alle der 11.757 Malware mit Verhaltenslabel dem Trainingsset zugeordnet werden. Der dadurch resultierende große Testdatensatz (im Verhältnis der gesamten Datenmenge) hat dadurch zudem den Vorteil, dass einzelne Ergebnisse weniger Einfluss auf die gesamte Bewertung des Modells haben und so die Ergebnisse eine größere Aussagekraft haben. Die Größen der einzelnen Datensätze sind in Tabelle 4.3 dargestellt, wobei jeweils gerundet 42 % durch die zuvor beschriebene Augmentation erzeugt wurde. Die Trainingsdaten wurden in 855 unterschiedlich große Batches mit je insgesamt maximal 4.200.000 Token aufgeteilt. Die augmentierte Version ist dabei immer im gleichen Datensatz, wie die Repräsentation der PE, aus welcher sie erzeugt wurde.

[14] Es können nur 5 PE pro Minute heruntergeladen werden, von welchen mehrere entweder DotNet-Anwendungen oder verpackt sind oder identischen Code, aber anderen Header oder Ressourcen wie bereits heruntergeladene PEs haben, was jedoch erst nach dem Herunterladen festgestellt werden kann

4.2.8 Aus dem Datensatz folgende Verlustfunktion für die MTL Modelle

Bei der Verlustfunktion muss also berücksichtigt werden, dass nur für Malware Verhaltenslabel vorhanden sind und auch diese nicht immer vollständig sind. Da ein Verhalten nur erkannt, nicht aber bei einer unvollständigen Observation als nicht vorhanden erkannt werden kann, können positive Label als wahr anerkannt werden, doch zumindest bei gutartigen PE oder bei Malware mit dem Label *Possible date expiration check, exits too soon after checking local time* alle nicht erkannten Label als unbeobachtet angesehen werden. In diesem Fall gibt es dann also keine negativen Label. Ignoriert man nun vorerst die Malware ohne dieses Label, bei denen die Label also als (annähernd) vollständig beobachtet angenommen werden, dann lässt sich feststellen, dass ein einfaches Ignorieren der nicht observierten Label dazu führt, dass schlussendlich alle Label als immer vorhanden gelernt werden [14, S. 936]. Dies kommt dadurch zustande, dass es dann nur Gradienten für den Ausgabewert in Richtung des Wertes 1, nicht aber in Richtung 0 gibt. Bei dem verwendeten Datensatz gibt es zwar viele Malware ohne dieses Label, doch ist hier der Anteil noch immer zu gering, da es noch immer mit einem starken Klassenungleichgewicht gleichzusetzen ist. Dies ist insbesondere der Fall, da berücksichtigt werden muss, dass Verhaltenslabel wie beispielsweise *Performs some HTTP requests* und *A process sent information about the computer to a remote location* —beispielsweise für Analysezwecke—auch auf gutartigen Anwendungen zutreffen können, doch bei diesen nicht angegeben sind. Das Problem der unvollständigen Label besteht also sowohl zwischen Malware und nicht Malware, als auch innerhalb der Malware-Samples.

In [14] werden verschiedene Ansätze, um mit diesem Problem umzugehen, miteinander verglichen, wobei in diesem Fall die Ansätze „assume negative [...] with label smoothing“[14, S. 937], bezeichnet als L_{AN-LS}, und „weak assume negative“[14, S. 936], bezeichnet als L_{WAN}, am besten geeignet zu scheinen sind. Der „regularized online label estimation“[14, S. 937] Ansatz erzielte in den von Cole, Mac Aodha u. a. [14] durchgeführten Untersuchungen zwar noch bessere Ergebnisse, doch entsteht durch ihn ein größerer Overhead —insbesondere auf den benötigten Arbeitsspeicher [14, S. 940]. Da dieser bereits der limitierende Faktor beim Training ist, wurde dieser Ansatz nicht weiter betrachtet. Da es zudem auch keinen Erwartungswert an vorhandene Label pro Datei gibt, ist auch „Expected Positive Regularization“[14, S. 937] nicht für diesen Anwendungsfall geeignet.

Nach [14] erzielt L_{AN-LS} bessere Ergebnisse als L_{WAN}, doch Labelsmoothing allein führte bei der Identifizierung der Verhaltenslabel zu keiner markanten Verbesserung und das Modell gab weiterhin für fast alle Verhaltenslabel fast immer einen Wert nahe 0 zurück. L_{AN-LS} ist dabei definiert als:

$$L_{AN-LS}(f_n, z_n) = -\frac{1}{L}\sum_{i=1}^{L}(\mathbb{1}^{\frac{\epsilon}{2}}_{[z_{n_i}=1]}\log(f_{n_i}) + \mathbb{1}^{\frac{\epsilon}{2}}_{[z_{n_i}\neq 1]}\log(1-f_{n_i})) \tag{4.4}$$

und L_{WAN} als:

$$L_{WAN}(f_n, z_n) = -\frac{1}{L}\sum_{i=1}^{L}(\mathbb{1}_{[z_{n_i}=1]}\log(f_{n_i}) + \mathbb{1}_{[z_{n_i}\neq 1]}\gamma\log(1-f_{n_i})) \tag{4.5}$$

mit $\mathbb{1}_{[z_{n_i}=1]} := 1$, wenn $z_{n_i} = 1$, sonst 0, $L :=$ Batchgröße und $\mathbb{1}^{\alpha}_{[\text{Bedingung}]} = (1-\alpha)\mathbb{1}_{[\text{Bedingung}]} + \alpha\mathbb{1}_{[\neg\text{Bedingung}]}$ [14, S. 936-937]. Beim MTL wurden also die Verluste aller negativen Verhaltenslabel anstelle nur der Verluste bei unbeobachteten Labeln reduziert, um dem Ungleichgewicht der Label entgegenzuwirken, welches auch dann besteht, wenn nur die als vollständig beobachtet angesehene Malware PEs betrachtet werden.

Experimentell erwies sich dabei die Kombination von $\epsilon = 0,1$ und $\gamma = 0,3$ als gut geeignet.[15]Da durch einen niedrigen γ-Wert auch der gesamte Verlust aller Verhaltenslabel geringer ist, wurde als Ausgleich der Faktor für den gesamten Verlust der Verhaltenslabelzuordnung von 0,5 auf 0,7 erhöht. Der Faktor für die Entscheidung zwischen Malware und gutartig wurde bei 1 belassen.

[15]Bei ϵ wurden nur Werte im Abstand von 0,05 und bei γ im Abstand von 0,1 untersucht.

5 Training und Untersuchungen

Der erste Anhaltspunkt über die Eignung der Techniken für die Malware-Erkennung liefert dabei ein Vergleich der Verlust-, Genauigkeits-, Fläche unter der Kurve (eng: area under the curve) (AUC)-Werte. AUC beschreibt dabei in diesem Fall die Fläche unter der Kurve, die die Sensitivität/Richtig-Positiv-Rate bei den jeweiligen FPR angibt, welche sich durch die Anpassung des Grenzwertes verändert. Da beide untersuchten Techniken in vorherigen Arbeiten eine Verbesserung des Modells hervorbrachten, war die Hypothese auch hier, dass die Modelle mit den untersuchten Techniken eine höhere Genauigkeit als das Basismodell aufweisen. Interessant ist aber darüber hinaus noch, wie groß die Unterschiede bei den zuvor genannten Werten sind und ob die verschiedenen Modelle gar verschiedene Malware besser erkennen als andere oder sie Malware anhand anderer Sequenzen oder implizit aus den Daten extrahierbaren Eigenschaften erkennen.

Zur Betrachtung, ob die einzelnen Varianten die Malware an unterschiedlichen Eigenschaften erkennen, wurden für 2 Malware Heatmaps erstellt, die beschreiben, um wie viel sich die Entscheidung ändert, wenn ein bestimmter Bereich nicht mehr vorhanden ist. Betrachtet wurde dabei zur Erzeugung je die 30 wichtigsten Bereiche.

Interessant ist auch, inwiefern diese beiden Techniken die Erklärbarkeit der Malware-Erkennung anhand neuronaler Netze verbessern können. Auch hier könnten die zuvor beschriebenen Untersuchungen Hinweise liefern. Für die Durchführung dieser Untersuchungen wurden alle Modelle für 15 Epochen trainiert. 15 wurde gewählt, sodass das Trainieren nicht zu lange dauert, jedoch die Modelle mit den dynamischen Faltungsschichten mehrere Epochen mit einer Temperatur von 1 trainiert werden. Wies ein Modell bei einer früheren als der letzten Epoche den geringsten Verlust beim Validierungsdatensatz auf, so wurde das Modell auch mit der Epoche des geringsten Verlustes getestet. Diese sind im nächsten Abschnitt mit dem Suffix _*epoch*<*Epoche*> gekennzeichnet. Um Schwankungen beim Trainieren aufgrund ungünstiger Startparameter etwas auszugleichen, wurden bei den Modellen mit den Basismodellen *Ref*, *SimpBase* und *SimpBaseDense*, welche die drei am wenigsten performanceaufwendigsten Modelle sind, alle Varianten außer der

DynMTL Variante zweimal trainiert. Bei allen Modellen wurde dieselbe über das CBOW-Verfahren[59] mit einer Fenstergröße von 4 vortrainierte Einbettung verwendet. Die Einbettung wurde 30 Epochen mit 1.110.070 zufällig gewählten Kontext→Ziel-Paaren vortrainiert und konnte während des Trainings der eigentlichen Modelle weiter geändert werden.

Trainiert wurde überwiegend mit je zwei Nvidia A100 40-GB Grafikkarten über einer durchschnittlichen Trainingsdauer von gerundet 1,5 Stunden bei den Modellen mit der Ref-Basis und 3,5 Stunden bei den restlichen Modellen. 80 GB waren jedoch nicht zwingend notwendig, doch bei maximalen Anzahl an Token von 4,2 Millionen waren je nach Modell 48 oder etwas mehr VRAM notwendig. Vor allem die Modelle mit DynConvs beanspruchten mehr als die 48 GB Speicher.

6 Modellvergleich und Diskussion

Wie der Abbildung 6.1 und den Tabellen 6.1 bis 6.3 zu erkennen ist, weisen die durchschnittlichen Ergebnisse je nach angewandter Technik und Modellgrundlage keine sehr großen Abstände auf. Zumindest die Genauigkeit konnte durch die Verwendung von MTL und/oder DynConvs bei den meisten Basismodellen jedoch etwas verbessert werden.

Bei niedriger FPR jedoch erzielen im Durchschnitt die Modelle mit DynConvs und die mit MTL eine höhere Empfindlichkeit von mehreren Prozentpunkten als die Basisvarianten bei gleicher FPR. Mit steigendem Grenzwert und damit steigender FPR verkleinert sich die Qualitätssteigerung der Modelle und wird gar negativ. Gerade wenn statt der durchschnittlichen Sensitivität der durchschnittliche relative Falsch-Negativ-Rate (FNR) jeweils im Verhältnis zum Basismodell gleicher Modellgrundlage (siehe Abbildung 6.2) betrachtet wird, lässt sich gut erkennen, dass sowohl bei der Verwendung von MTL als auch DynConv es zu einer starken Zunahme der FNR führen kann. Generell lässt sich zudem erkennen, dass die Qualität teils stärker vom verwendeten Basismodell als von den in dieser Arbeit zentral untersuchten Techniken abhängt—siehe dazu auch Abbildung 6.3. So ist ab einer FPR von 0,25% in Richtung einer FPR von 100% die durchschnittliche Richtig-Positiv-Rate (eng: true positive rate) (TPR) der Modelle mit der Referenz-Modellgrundlage fast immer höher als die der anderen Modellgrundlagen, und dies um oft mehr Prozentpunkte als bei der Unterscheidung nach den eingesetzten Techniken. Bereits das Vorhandensein der weiteren vollständig verbundenen Schicht zur theoretischen Erhöhung der Kombinierbarkeit kann das Modell markant verbessern. Eine Tabelle mit allen TPRs und FPRs bei den Grenzwerten $\{0,1, 0,2, \ldots, 0,9\}$ ist in Anhang A.3.

Bezogen auf die Erkennung von Malware gibt es zwei verschiedene Sichtweisen. Zum einen sollte die Erkennungsrate möglichst hoch sein, da eine einzelne unerkannte Schadsoftware bereits drastische Folgen haben kann. Andererseits kann auch eine zu hohe FPR die Akzeptanz des Scanners reduzieren und dazu führen, dass eine tatsächliche Erkennung als ein erneuter Fehlalarm angenommen wird, da es einfach zu viele Fehlalarme gibt, um sie alle zu berücksichtigen [6]. Bereits eine FPR von 1 % würde bei einem System von

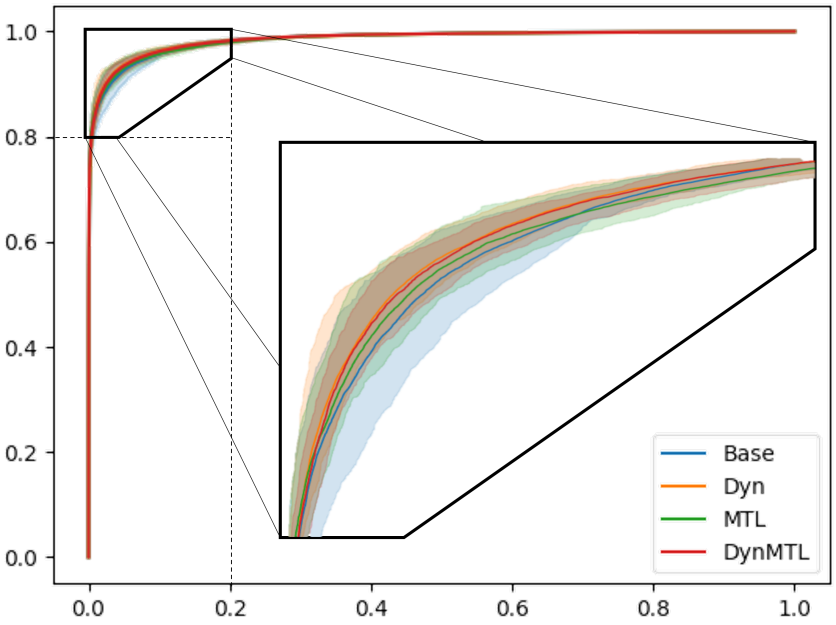

Abbildung 6.1: ROC-Kurven nach Technik

	Base	Dyn	MTL	DynMTL
SimpBase	0.98141	0.97954	**0.98312**	0.98288
SimpBaseDense	0.98397	0.98375	0.98356	**0.9859**
Ref	0.98566	0.98389	**0.98608**	0.98491
Base	0.98119	0.98171	**0.98179**	0.98102
BaseDense	0.98418	0.98173	0.98071	**0.98581**

Tabelle 6.1: Beste AUC-Werte aller Modelle bei Grenzwert = 0,5

Variante	Base	Dyn	MTL	DynMTL
Ref	0,94852	0,94449	**0,94878**	0,94754
Base	0,93004	**0,93997**	0,9388	0,93515
BaseDense	0,93681	0,94173	0,93353	**0,94574**
SimpBase	0,93119	**0,9396**	0,93604	0,9391
SimpBaseDense	**0,94414**	0,93992	0,94046	0,93878

Tabelle 6.2: Genauigkeit aller Modelle mit den besten AUC-Werten bei Grenzwert = 0,5

Variante	Base	Dyn	MTL	DynMTL
SimpBase	0,18852	0,22952	**0,17196**	0,18197
SimpBaseDense	0,17964	0,17083	0,1636	**0,14832**
Ref	0,15774	0,18215	**0,1562**	0,16233
Base	0,2013	0,21014	**0,18966**	0,19446
BaseDense	0,16262	0,20092	0,19969	**0,1539**

Tabelle 6.3: Verlust aller Modelle mit besten AUC-Werten bei Grenzwert = 0,5

	Base	Dyn	MTL	DynMTL
SimpBase	0,93116	**0,93958**	0,93603	0,9391
SimpBaseDense	**0,94413**	0,93992	0,94046	0,93874
Ref	0,94852	0,94448	**0,94878**	0,94754
Base	0,93	**0,93996**	0,9388	0,93514
BaseDense	0,93675	0,9417	0,93352	**0,94572**

Tabelle 6.4: F1-Wert aller Modelle mit den höchsten AUC-Werten bei Grenzwert=0,5

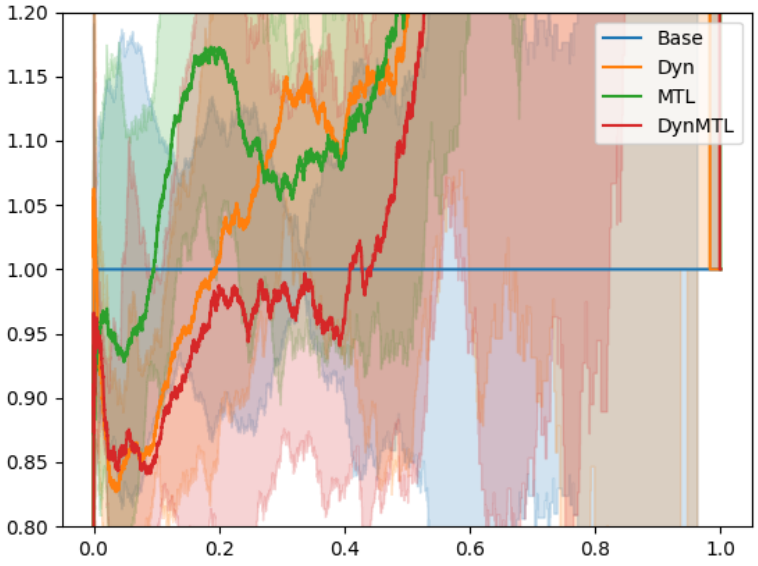

Abbildung 6.2: Relative FNR unter Verwendung von DynConv und MTL relativ zum jeweiligen Basismodell gleicher Modellgrundlage

über 200.000 PEs bedeuten, dass 2.000 PEs fehlerhaft als Malware eingestuft wurden, was im Produktivsystem nicht gerade akzeptabel ist—siehe dazu die ausführliche Erklärung

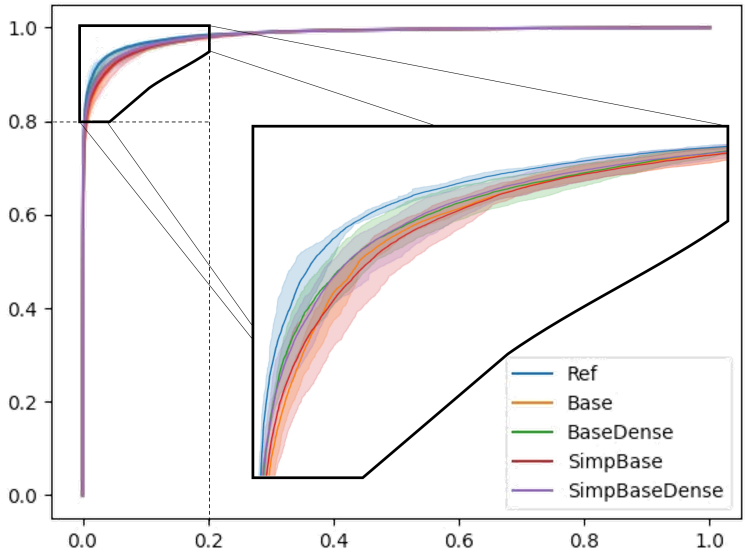

Abbildung 6.3: ROC-Kurven nach Modellgrundlage

in [6]. Aufgrund von oft sicherheitsrelevanten Updates ist zudem auch eine Whitelist keine Option, da auch Updates kompromittiert sein können, wenn beispielsweise der Hersteller angegriffen wurde. Kommerzielle Malware-Erkennungsprogramme haben daher eine sehr niedrige FPR von möglichst weit unter 0,001 % bei einer Sensitivität von dato bis zu 99,7 % bei einer rein statischen Analyse [8, 57]. In Kombination von dynamischen Analysen ist die Sensitivität jedoch noch markant besser [57], was die Wichtigkeit von der Kombination mehrerer Erkennungswege hervorhebt. Die Kombination mehrerer Techniken wiederum erhöht erneut die Notwendigkeit für eine geringe FPR der einzelnen Methoden. Für die Findung von Merkmalen zur Nutzung in der Malware-Erkennung ohne neuronale Netze gilt das Gleiche, da Merkmale, die auch auf viele gutartige Software zutreffen, nur beschränkt hilfreich sind.

Da jedoch bei so einer kleinen FPR kein einziges Modell eine annähernd akzeptable Sensibilität erreichen konnte, wurde der betrachtete Bereich auf 0,05% (0,0005) erweitert. In diesem Bereich sind, wie den Abbildungen 6.7 bis 6.10 zu erkennen ist, im Durchschnitt vor allem die MTL-Modelle besser als die Basismodelle, und dies bei fast allen getesteten Modellgrundlagen und Trainingsdurchläufen. Durch die Verwendung von DynConv konnte ebenfalls in mehreren Fällen die Sensitivität bei gleicher FPR erhöht werden,

doch gibt es hier mehr Kombinationen, in denen DynConvs zu einer Verschlechterung des Modells führt. Da die Sensitivität teils durch die Verwendung von DynConvs sehr stark abnehmen kann, ist auch, wie in Abbildung 6.10 zu erkennen ist, die durchschnittliche zum Basismodell der jeweiligen verwendeten Modellgrundlage relative FNR bei der Verwendung von DynConvs bei fast allen niedrigen FPRs über 1. Die FNR nahm im Durchschnitt also zu und nicht ab, wobei es zwischen den Modellgrundlagen größere Schwankungen gibt und teils die FNR um über 40% reduziert werden konnte. Betrachtet man dagegen die Sensitivität (Abbildung 6.9), so bedeutet diese Reduzierung der FNR teilweise eine Verdopplung der Sensitivität. Auffällig ist zudem, dass, wie in den Abbildungen 6.11 und 6.12 zu erkennen ist, bei niedriger FPR die Modelle mit der zusätzlichen vollständig verbundenen Schicht vor der Ausgabeschicht im Durchschnitt markant besser abschneiden. Genau bei diesen Modellen jedoch führen DynConvs und teils auch das MTL zu einer Verschlechterung des Modells anstelle zu einer Verbesserung —siehe dazu die Tabellen 6.8 und 6.10. Interessanterweise gilt dies, wie in der Tabelle 6.11 zu erkennen ist, jedoch nicht immer, da beispielsweise ohne den aufgelösten Funktionsnamen bei der SimpBaseDense-Modellgrundlage auch die Dyn-Variante bessere Ergebnisse, als die Base-Variante. Generell bestätigen die Ergebnisse in Tabelle 6.11 im Vergleich zu den Ergebnissen in Tabelle 6.10 jedoch den Mehrwert der Funktionsauflösung. Durch diese stieg bei geringer FPR die Sensitivität meist um über 20%.

Ob die Verwendung von mehreren Faltungsschichten hintereinander für die Malware-Erkennung vorteilhaft ist, lässt sich aus den Ergebnissen zumindest ohne Untersuchung des Referenzmodells mit einer zusätzlichen vollständig verbundenen Schicht nicht beantworten. Diese Arbeit bestätigt jedoch, dass die Feststellung in [12], dass DynConvs in späteren Schichten effektiver sind, auch für die Erkennung von Malware gilt.

Zu erkennen ist in Tabelle 6.10 auch, dass jedoch selbst beim gleichen Modell das schlussendliche Ergebnis stark variieren kann, je nachdem, wie unter anderem gerade die Anfangsausprägungen der Parameter waren. Die Menge der Modellgrundlagen, bei denen die Techniken für einen Mehrwert führen, ändert sich im Vergleich zwischen den besten Basismodellen und den jeweils schlechtesten erweiterten Modellen (siehe Tabelle 6.9) jedoch nur in einem von den 6 Fällen, bei denen mehrere Modelle trainiert wurden. Der Anstieg war bei diesem auch bereits im Vergleich mit ausschließlich den besten Modellen kleiner als der Durchschnitt. Bei der Adoption der Techniken in neuen Modellen kann daher zwar eine Evaluierung mit mehreren Trainingsdurchläufen hilfreich sein, scheint jedoch nicht zwingend notwendig zu sein. Allgemein ist die Erfolgsquote bei der Anwendung von MTL höher als bei DynConvs und bringt im Durchschnitt auch mehr. Bei den

MTL- und DynConv-Modellen ist der erforderliche Grenzwert für eine niedrige FPR zwar meist minimal geringer, doch selbst für eine FPR von unter 0,0005 ist der erforderliche Grenzwert noch immer bei fast allen Modellen über 0,99.

Bei MTL kommt zudem noch dazu, dass die zweite Ausgabe gegebenenfalls noch zusätzliche erste Einblicke liefern kann, bezogen beispielsweise darauf, warum dies als Malware erkannt wurde oder um welchen Typ von Malware es sich handelt. Wie der Abbildung 6.4 und den Tabellen 6.5 und 6.6 (ohne die *OnlySig*-Modelle) entnommen werden kann, erreichte die Signatur-Erkennung als zweites Ziel jedoch keine sehr hohe Genauigkeit. Viele Verhalten wurden nicht gut erkannt. Öfters wurde zudem ein Verhalten erkannt, auch wenn es nicht vorhanden ist. Vergleicht man in diesen Tabellen diese Ergebnisse zu dem Fall, in welchem die Signatur als getrenntes Modell trainiert wurde, dann fällt jedoch auf, dass im zweiten Fall nur eine gering höhere Genauigkeit erreicht wurde, was daraufhin deutet, dass die Verhaltenslabel allgemein schwierig zu lernen sind. Dass viele Verhalten nur sehr schlecht erkannt wurden und auch mehrere Verhalten erkannt, auch wenn es nicht vorhanden sind, ist konnte auch bei den einzeln trainierten Modelle festgestellt werden. Beispielhafte Rog-Kurven für diese Modelle sind in den Abbildungen 6.5 und 6.6 dargestellt. Da das OnlySigBaseDense-Modell jedoch sowohl die höchste Genauigkeit, als auch den geringsten Verlust erzielte, scheint es zumindest bei diesem Ziel noch sinnvoll zu sein, dass für das zweite Ziel ein eigenes Modell trainiert wird und nicht (ausschließlich) die Ergebnisse aus dem Malware-Erkennungs-Modell verwendet werden. Im Malware-Erkennungs-Modell dient das zweite Ziel dann gegebenenfalls also nur zur Steigerung der Genauigkeit. Hier besteht aber möglicherweise auch noch die Möglichkeit, dass beispielsweise über weitere Anpassungen der Verlustskalierungen und Weak-Assume-Negative-Faktoren eine bessere Modellqualität bezogen auf das zweite Ziel erreicht werden kann, ohne stark die Qualität bezogen auf das Hauptziel zu verringern.

Nach der Anzahl der Multiplizieren-Akkumulieren Berechnungen (eng: multiply-accumulate computations) (MACs) ist das MTL zudem weniger rechenaufwendig als die Verwendung von DepConvs. Bei den Untersuchungen dauerte die Klassifizierung aller Daten des Testdatensatzes bei den Dyn-Modellen im Durchschnitt 37,36 % länger, als bei den Baseline-Modellen. Bei den MTL-Modelle dauerte die Klassifizierung dagegen im Durchschnitt nur 0,07 % länger, als bei den Baseline-Modellen. Falls der Speicherverbrauch und die Dauer für die Klassifizierung relevant ist, dann sollte gegebenenfalls auf DynConvs verzichtet werden. Die genauen Zeiten für die Klassifierung des Testdatesatzes mit je zwei Nvidia Quadro P6000 (je 24 GB) Grafikkarten ist in der Tabelle 6.7 aufgelistet.

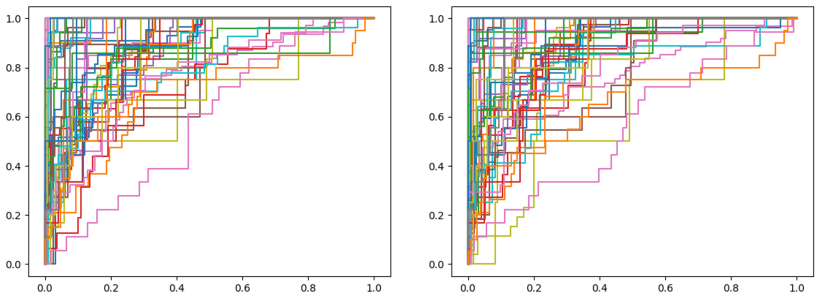

Abbildung 6.4: ROC-Kurven der Verhaltenlabel der zwei besten Malware-Erkennungs-Modelle: BaseDenseMTL (links) und BaseDynMTL (rechts).

	MTL	DynMTL
BaseDense	0.9323	0.9347
SimpBaseDense	0.9352	0.9325
Base	0.9326	**0.9367**
SimpBase	0.9358	0.9363
Ref	0.9347	**0.9368**
OnlySigBaseDense	0.9385	0.934
OnlySigSimpBaseDense	**0.9402**	
OnlySigBaseDenseMal	0.9269	0.9214
OnlySigSimpBaseDenseMal	0.9376	

Tabelle 6.5: Binäre-Genauigkeit für die Verhaltenslabel aller Modelle bei Grenzwert = 0,5

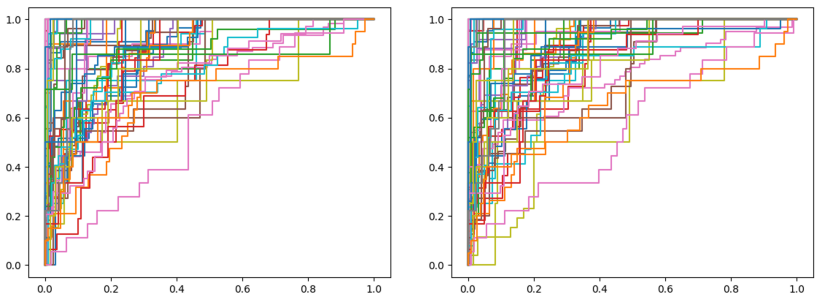

Abbildung 6.5: ROC-Kurven aller Verhaltenlabels der Modelle OnlySigSimpBaseDense (Links) und OnlySigSimpBaseDenseMal (Rechts).

Beide Techniken können aber auch miteinander kombiniert werden, wodurch die durchschnittlich höchste Sensitivität bei fast allen niedrigen FPRs erreicht wird. Dies gilt auch

	MTL	DynMTL
BaseDense	**0.1274**	0.1309
SimpBaseDense	0.134	**0.1279**
Base	0.1354	0.1322
SimpBase	0.1347	0.1348
Ref	0.1306	0.1303
OnlySigBaseDense	**0.1232**	0.1294
OnlySigSimpBaseDense	0.1253	
OnlySigBaseDenseMal	0.1309	0.135
OnlySigSimpBaseDenseMal	0.1262	

Tabelle 6.6: Verlust für die Verhaltenslabel aller Modelle

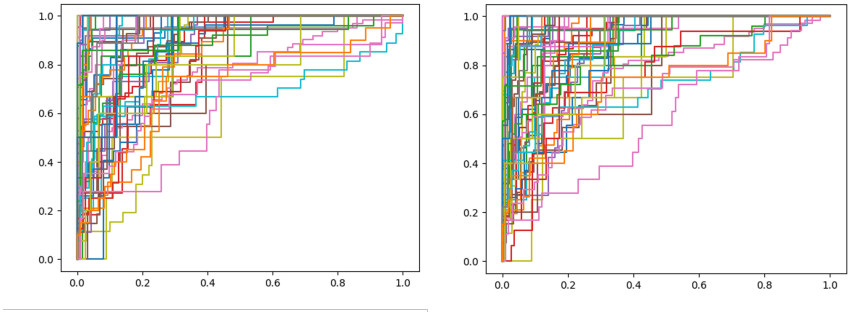

Abbildung 6.6: ROC-Kurven aller Verhaltenlabels der Modelle OnlySigBaseDense (Links) und OnlySigBaseDenseDyn (Rechts).

für fast alle FPRs von bis zu 0,0025, wie in Abbildung 6.8 zu erkennen ist. Wie der Tabelle 6.10 und der Abbildung 6.7 entnommen werden kann, wurde jedoch nicht immer beim Vergleich bei fixer FPR die höchste Sensitivität bei den DynMTL-Modellen erreicht, sondern teils auch bei den reinen MTL-Modellen ohne DepConvs und bei manchen FPRs erreichte auch das Basismodell die höchste Richtig-Positiv-Rate. Auch wenn sowohl MTL als auch DepConvs zu einer Verbesserung des Modells führen, führt eine Kombination beider nicht zwingend zu einer weiteren Verbesserung, sondern kann teils auch schlechter sein, als eine der beiden Techniken allein. Fast alle der Top 10 % Richtig-Positiv-Raten bei den einzelnen FPRs wurden dabei von Modellen der SimpBaseDense Modellgrundlage erreicht.

	Base	Dyn	MTL	DynMTL
BaseDense	0h 23m 54s	0h 33m 17s	0h 23m 48s	0h 33m 13s
SimpBaseDense	0h 23m 51s	0h 33m 32s	0h 23m 53s	0h 33m 31s
Base	0h 23m 57s	0h 33m 19s	0h 23m 55s	0h 33m 7s
SimpBase	0h 24m 8s	0h 33m 53s	0h 24m 7s	0h 33m 54s
Ref	0h 15m 56s	0h 19m 27s	0h 16m 5s	0h 19m 32s

Tabelle 6.7: Dauer der vollständigen Klassifizierung des Testdatensatzes

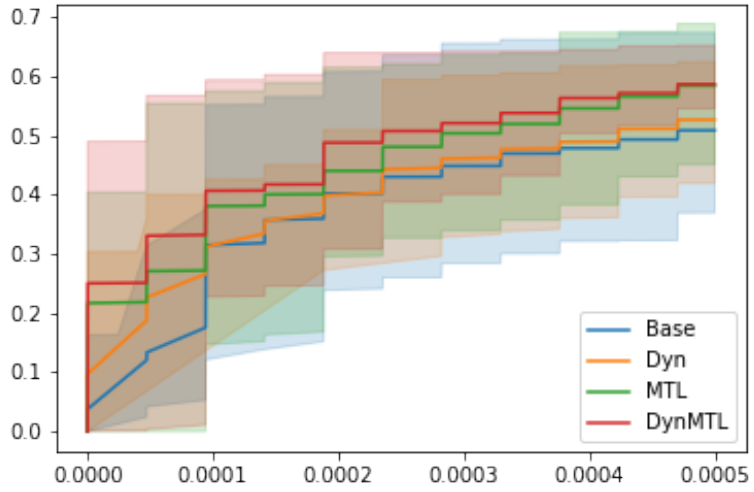

Abbildung 6.7: ROC-Kurven nach Technik bei niedriger FPR

	Base	Dyn	MTL	DynMTL
Ref	0,5112	0,5287	0,5574	**0,5679**
Base	0,3911	**0,6245**	0,4510	0,5455
BaseDense	**0,6692**	0,4898	0,5501	0,5542
SimpBase	0,4039	0,5949	0,6514	**0,6523**
SimpBaseDense	0,6739	0,5766	**0,6890**	0,6134

Tabelle 6.8: Beste TPR je nach verwendeter Technik und Modellgrundlage bei einer FPR von 0.0005

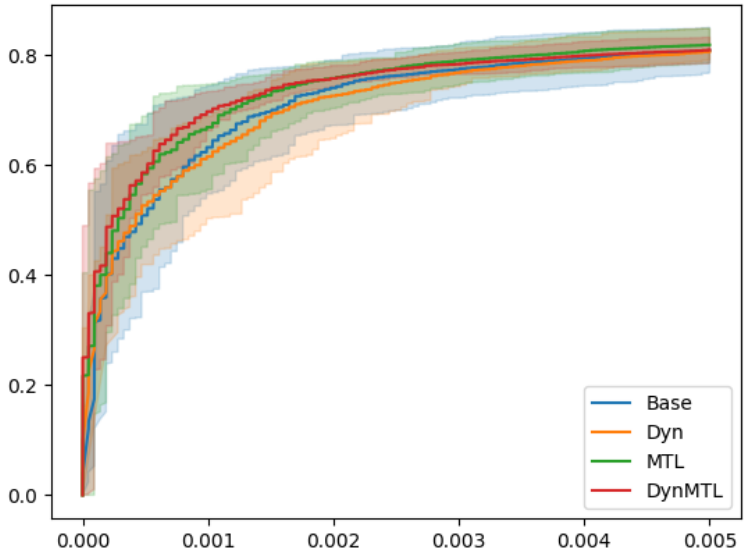

Abbildung 6.8: ROC-Kurven nach Technik bis zur FPR von 0,005

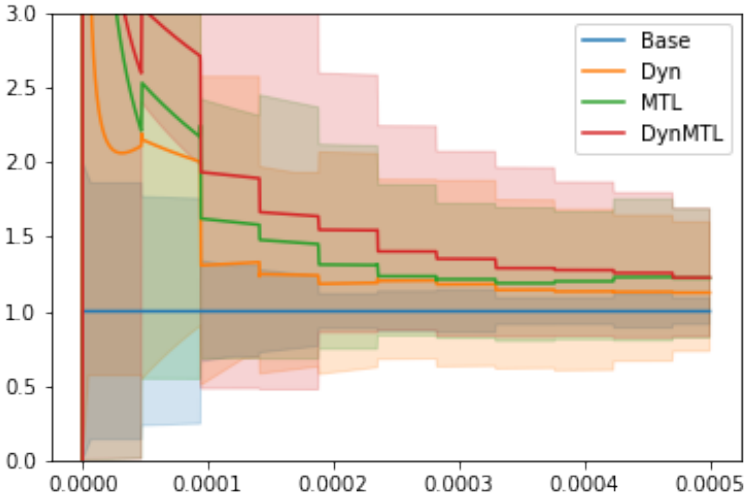

Abbildung 6.9: Relative TPR nach Technik relativ zum jeweiligen Basismodell bei niedriger FPR

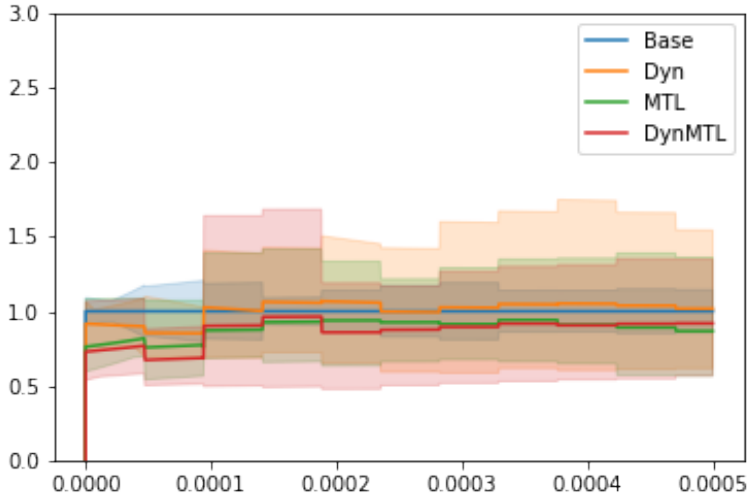

Abbildung 6.10: Relative FNR bei Verwendung von DynConv und MTL relativ zum jeweiligen Basismodell gleicher Modellgrundlage bei niedriger FPR

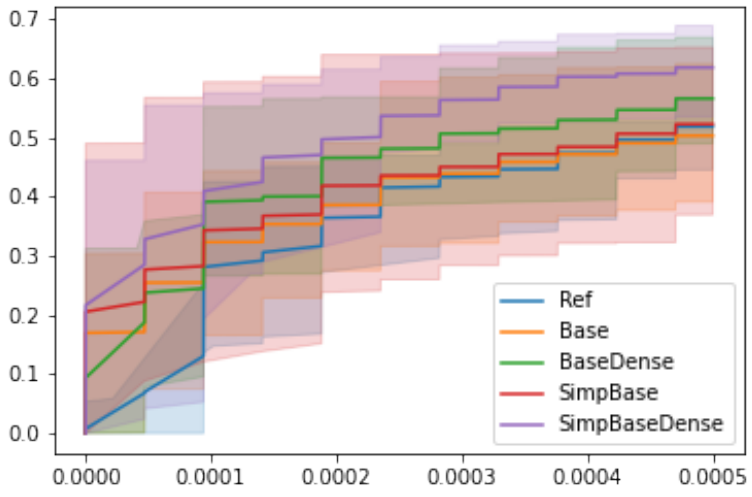

Abbildung 6.11: ROC-Kurven nach Modellgrundlage bei niedriger FPR

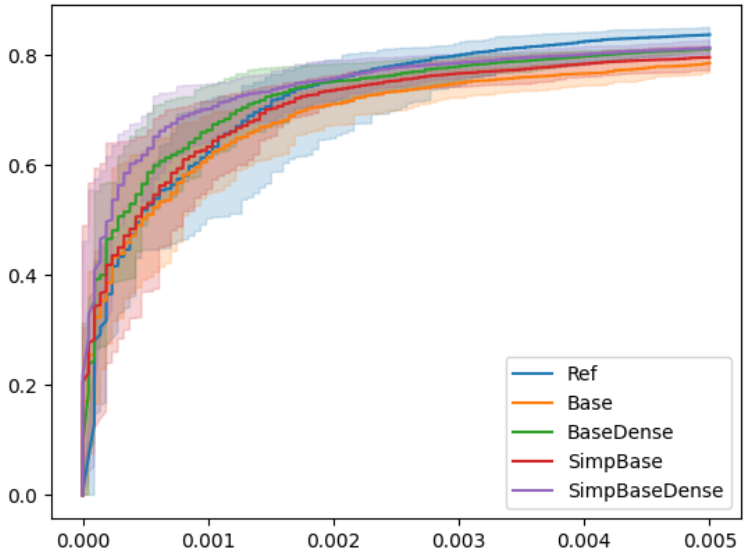

Abbildung 6.12: ROC-Kurven nach Modellgrundlage bis zur FPR von 0,005

	Base	Dyn	MTL
Ref	0,5112	0,4452	0,5315
SimpBase	0,4039	0,4193	0,5673
SimpBaseDense	0,6739	0,5349	**0,6757**

Tabelle 6.9: Geringste TPR je nach verwendeter Technik und Modellgrundlage bei einer FPR von 0.0005 im Vergleich zur besten TPR der Basisvariante

Modell	0.0001	0.0003	0.0005	0.0007
Ref	0,2975 (1,0000)	0,4189 (0,9997)	0,4872 (0,9994)	0,5333 (0,9989)
Ref2	0,3745 (1,0000)	0,4607 (0,9997)	0,5112 (0,9992)	0,5566 (0,9985)
RefDyn2	0,4252 (1,0000)	0,4896 (1,0000)	0,5287 (0,9999)	0,5418 (0,9998)
RefDyn	0,1446 (1,0000)	0,3296 (1,0000)	0,4452 (1,0000)	0,4486 (1,0000)
RefMTL2	0,1472 (1,0000)	0,4877 (0,9999)	0,5315 (0,9998)	0,545 (0,9998)
RefMTL	0,3612 (0,9998)	0,4478 (0,9996)	0,5574 (0,9990)	0,6391 (0,9979)
RefDynMTL	0,2276 (1,0000)	0,4006 (0,9999)	0,5679 (0,9996)	0,6349 (0,9992)
Base	0,1658 (1,0000)	0,3211 (0,9998)	0,3911 (0,9996)	0,4213 (0,9995)
BaseDyn	0,4259 (0,9999)	0,6015 (0,9995)	0,6245 (0,9994)	**0,65** (0,9991)
BaseMTL	0,2581 (0,9995)	0,3393 (0,9989)	0,451 (0,9976)	0,4943 (0,9968)
BaseDynMTL	0,4435 (0,9996)	0,4934 (0,9994)	0,5455 (0,9990)	0,5806 (0,9985)
BaseDense	0,552 (0,9999)	0,6171 (0,9999)	**0,6692** (0,9996)	**0,7166** (0,9982)
BaseDenseDyn	0,3708 (1,0000)	0,3888 (1,0000)	0,4898 (1,0000)	0,5364 (1,0000)
BaseDenseMTL	0,3773 (0,9999)	0,5059 (0,9995)	0,5501 (0,9992)	0,5719 (0,9990)
BaseDenseDynMTL	0,2662 (1,0000)	0,5151 (0,9997)	0,5542 (0,9996)	0,6301 (0,9992)
SimpBase2	0,1231 (1,0000)	0,3371 (1,0000)	0,4039 (0,9999)	0,4681 (0,9998)
SimpBase	0,2443 (1,0000)	0,284 (0,9999)	0,3693 (0,9998)	0,4536 (0,9994)
SimpBaseDyn2	0,2868 (1,0000)	0,5258 (0,9993)	0,5949 (0,9966)	0,6126 (0,9953)
SimpBaseDyn	0,2807 (1,0000)	0,354 (1,0000)	0,4193 (1,0000)	0,469 (0,9999)
SimpBaseMTL2	0,4416 (0,9905)	0,4752 (0,9873)	0,5673 (0,9755)	0,6076 (0,9667)
SimpBaseMTL	0,4337 (0,9997)	0,5341 (0,9993)	**0,6514** (0,9975)	**0,7003** (0,9936)
SimpBaseDynMTL	0,5938 (0,9998)	0,6423 (0,9997)	**0,6523** (0,9996)	**0,6683** (0,9994)
SimpBaseDense2	0,2682 (1,0000)	0,4933 (0,9999)	0,5646 (0,9995)	0,6244 (0,9982)
SimpBaseDense	0,4978 (1,0000)	**0,6559** (1,0000)	**0,6739** (0,9999)	**0,7034** (0,9997)
SimpBaseDenseDyn2	0,379 (1,0000)	0,5061 (1,0000)	0,5349 (1,0000)	0,6039 (0,9999)
SimpBaseDenseDyn	0,205 (1,0000)	0,4966 (1,0000)	0,5766 (1,0000)	0,6135 (1,0000)
SimpBaseDenseMTL2	0,5751 (0,9999)	0,636 (0,9998)	**0,6757** (0,9995)	**0,6967** (0,9993)
SimpBaseDenseMTL	0,4524 (0,9999)	0,6045 (0,9995)	**0,689** (0,9979)	**0,7306** (0,9920)
SimpBaseDenseDynMTL	0,502 (0,9996)	0,5534 (0,9993)	0,6134 (0,9987)	**0,7163** (0,9957)
Base Durchschnitt	0,3154	0,4485	0,5088	0,5597
Dyn Durchschnitt	0,3147	0,4615	0,5267	0,5595
MTL Durchschnitt	0,3808	0,5038	0,5842	0,6232
DynMTL Durchschnitt	0,4066	0,5210	0,5867	0,6461

Tabelle 6.10: Sensitivität aller Modelle bei geringer FPR mit Grenzwertangaben. Die 10 % höchsten Ergebnisse sind fettgedruckt. Pro FPR ist die höchste Sensitivität grau hinterlegt. Alle Werte wurden auf die vierte Nachkommastelle gerundet.

Beim Vergleich der Heatmaps (siehe Anhang A.5) der einzelnen Modellvarianten konnte nur wenig festgestellt werden. Mehrere Stellen waren bei fast allen Modellen mit ausschlaggebend für die Klassifizierung als Malware, doch viele Stellen fielen nur bei manchen Modellen in die 30 wichtigsten Teilsequenzen, wobei hier die Unterschiede zwischen den einzelnen Trainings des gleichen Modells nicht markant geringer sind, als zwischen den unterschiedlichen Techniken. Die Unterschiede zwischen den einzelnen Modellvarianten sind mindestens genauso groß. Auffällig ist jedoch, dass bei den Modellen mit den Modellgrundlagen Ref und SimpBase einzelne Teilsequenzen bei den Basismodellen einen größeren Einfluss haben als bei den anderen Modellen. Bei der Modellgrundlage Base ist dies jedoch nicht der Fall, weshalb dies bei den ersten zwei Modellen auch nur Zufall sein kann, da dies auch nicht bei allen dieser trainierten Basismodelle der Fall ist. Zudem lässt sich aber auch erkennen, dass generell bei allen besseren Modellen einzelne Sequenzen geringeren Einfluss haben und mehr die Kombination mehrerer Sequenzen entscheidender sind.

Modell	0.0001	0.0003	0.0005	0.0007
SimpBaseNoFunc	0.0442 (1.0000)	0.3613 (0.9989)	0.5308 (0.9970)	0.5643 (0.9962)
SimpBaseDynNoFunc	0.1671 (1.0000)	0.3414 (1.0000)	0.471 (0.9999)	0.5171 (0.9998)
SimpBaseMTLNoFunc	0.4763 (0.9983)	0.5587 (0.9964)	**0.6393** (0.9888)	**0.6458** (0.9872)
SimpBaseDenseNoFunc	0.1795 (1.0000)	0.5085 (0.9994)	0.5605 (0.9988)	0.5935 (0.9979)
SimpBaseDenseDynNoFunc	0.293 (1.0000)	0.5265 (0.9997)	0.5694 (0.9996)	**0.6183** (0.9989)
SimpBaseDenseMTLNoFunc	0.1524 (1.0000)	0.2737 (0.9998)	0.3946 (0.9996)	0.4746 (0.9994)
Base Durchschnitt	0.1119	0.4349	0.5456	0.5789
Dyn Durchschnitt	0.2301	0.4339	0.5202	0.5677
MTL Durchschnitt	0.3143	0.4162	0.5169	0.5602

Tabelle 6.11: Sensitivität der zuvor besten Modelle ohne Funktionsauflösung bei der Eingabe. Die 10 % höchsten Ergebnisse sind fettgedruckt. Pro FPR ist die höchste Sensitivität grau hinterlegt. Alle Werte wurden auf die vierte Nachkommastelle gerundet.

7 Fazit

Diese Bachelorarbeit untersuchte, inwiefern die Verwendung von MTL und DynConvs in der merkmalslosen Malware-Erkennung über neuronale Netze zu einer zuverlässigeren Unterscheidung zwischen Malware und gutartiger Software und zu zusätzlichen Erkenntnissen über die übergebenen PEs führen kann. Dafür wurde eine neue, möglichst geeignete Form der Daten erarbeitet, in welcher die PEs den Modellen übergeben werden können.Mit Hilfe einer Ähnlichkeitsanalyse der Trainingsdaten über die prozentuale Abweichung der verwendeten DLL-Funktionen und den TLSH der Programmanweisungen wurde eine neue Methode vorgeschlagen und verwendet, über welche möglichst viele gutartige PEs gesammelt werden können, ohne zu ähnliche Daten im Datensatz zu haben. Erweitert wurde dies über eine Sammlung verschiedenster oft verwendeter Software, wodurch in Kombination mit dem teilweisen Ausschluss von nativen Programmen von Microsoft sichergestellt wurde, dass kein Hersteller-Bias beim Training entstand. Zudem wurde eine leicht modifizierte Art der Augmentation von Anweisungssequenzen von PEs vorgestellt, durch welche die Erhaltung der ursprünglichen Programmlogik sichergestellt werden kann. Die Effektivität dieser Augmentation muss jedoch noch in einer späteren Untersuchung belegt werden.

Die Eignung wurde dabei anhand von 5 sich unterscheidenden Basismodellen, in welchen DynConvs und die Verhaltenserkennung der Malware als zweites Ziel für das MTL integriert wurden, und insgesamt 29 trainierten Modellen untersucht. Dabei konnte gezeigt werden, dass vor allem das MTL eine effektive Lösung sein kann, um sowohl die Sensitivität des Modells auch bei niedriger FPR zu verbessern als auch zeitgleich mehr Einblicke darüber zu bekommen, warum die PE als Malware oder gutartig klassifiziert wurde.Auch DynConvs können zur Qualitätsteuerung des Modells beitragen, wenn dies auch etwas seltener zu einer Steigerung der Sensitivität führt und der Anstieg im Durchschnitt zudem meist geringer ausfällt. Beide Techniken können auch kombiniert werden, wodurch die höchste durchschnittliche Sensitivität bei geringer FPR erreicht wird. Die Kombination führt jedoch nicht immer zu einer weiteren Verbesserung, selbst wenn beide Techniken allein vorteilhaft sind.

Schlussendlich kann festgehalten werden, dass es sich bei neuen Modellen meist lohnt, beide Techniken auszuprobieren, wobei gerade die Verwendung von DynConvs aufgrund der einfachen Nutzung anstelle von Standard-Faltungsschichten zumindest in der Entwicklung keinen wirklichen Mehraufwand verursacht. Die in dieser Arbeit durchgeführten Untersuchungen zeigen jedoch Anzeichen dafür, dass DynConvs nicht gut funktionieren, wenn nach den Faltungsschichten neben der Ausgabenschicht weitere vollständig verbundene Schichten verwendet werden, auch wenn diese gegebenenfalls ohne DynConvs zu einer Verbesserung des Modells führen. Die längeren Klassifizierungszeit und der erhöhten Speicherbedarf bei der Verwendung von DynConvs könnten jedoch je nach Anwendungszweck als Nachteile der teils nur geringen Steigerung der Genauigkeit überwiegen.

Je nach weiterem Ziel erfordert MTL deutlich mehr Vorbereitungsaufwand, doch kann dies zu einer markant größeren Verbesserung des Modells führen. Gerade, wenn sowieso ein zweites Ziel in einem externen Modell gelernt werden soll, dann scheint es meist vorteilhaft zu sein, dieses auch neben der Malware-Erkennung im gleichen Modell zu trainieren.

8 Ausblick

Hyperparameter wurden in dieser Arbeit nur geringfügig optimiert, wobei gerade bei der Anzahl der Experten bei den DynConvs und die Skalierung der Verluste zwischen den zwei Zielen genauer betrachtet werden könnte. Auch die Effektivität der Funktionsauflösung für einen größeren Informationsgehalt in den Eingaben und der in dieser Arbeit beschriebenen Augmentation könnte noch weiter untersucht werden. Offen bleibt zudem die Frage, was die Modelle tatsächlich gelernt haben. Eine einfache Erzeugung von Heatmaps lässt deutet zwar nicht direkt darauf, dass die Modelle durch diese Techniken andere Arten von Merkmalen erkennen können, doch eine tiefgreifendere Analyse könnte auf etwas anderes hindeuten und eventuell auch erklären, warum die Modelle mit einer zusätzlichen vollständig verbundenen Schicht schlechtere Ergebnisse erzielen, wenn DynConvs anstelle von normalen Faltungsschichten verwendet werden. Offen bleibt zudem die verwandte Frage, ob diese Techniken einen Einfluss auf die Anfälligkeit für Konzeptdrift oder Adversarial Attacks haben.

Neben diesen zwei in dieser Arbeit betrachteten Techniken gibt es jedoch noch viele weitere vielversprechende Möglichkeiten, die (statische) merkmalslose Malware-Erkennung über neuronale Netze zu verbessern. Neben den bereits in der Einleitung erwähnten Block-Recurrent Transformers[42], wurde während der Forschung für diese Arbeit von Gu und Dao [38] *Mamba* als Alternative für Transformer-Architekturen vorgestellt, die anders als Transformer nur eine lineare Skalierung mit steigender Eingabelänge hat und damit möglicherweise eine Alternative für CNN für die merkmalslose Malware-Erkennung darstellt [38, S. 1].

Literatur

[1] Faitouri A. Aboaoja u. a. „Malware Detection Issues, Challenges, and Future Directions: A Survey“. In: *Applied Sciences* 12.17 (2022), S. 8482. ISSN: 2076-3417. DOI: 10.3390/app12178482.

[2] Nadeem Akhtar und U. Ragavendran. „Interpretation of intelligence in CNN-pooling processes: a methodological survey“. en. In: *Neural Computing and Applications* 32.3 (Feb. 2020), S. 879–898. ISSN: 1433-3058. DOI: 10.1007/s00521-019-04296-5. URL: https://doi.org/10.1007/s00521-019-04296-5 (besucht am 10. 07. 2023).

[3] Ahmed Amer und Normaziah A. Aziz. „Malware Detection through Machine Learning Techniques“. In: *International Journal of Advanced Trends in Computer Science and Engineering* 8.5 (2019), S. 2408–2413. DOI: 10.30534/ijatcse/2019/82852019.

[4] Eslam Amer und Ivan Zelinka. „A dynamic Windows malware detection and prediction method based on contextual understanding of API call sequence“. In: *Computers & Security* 92 (2020), S. 101760. DOI: 10.1016/j.cose.2020.101760.

[5] Andr&eacute Araujo, Wade Norris und Jack Sim. „Computing Receptive Fields of Convolutional Neural Networks“. en. In: *Distill* 4.11 (Nov. 2019), e21. ISSN: 2476-0757. DOI: 10.23915/distill.00021. URL: https://distill.pub/2019/computing-receptive-fields (besucht am 08. 10. 2023).

[6] Stefan Axelsson. „The base-rate fallacy and the difficulty of intrusion detection“. In: *ACM Trans. Inf. Syst. Secur.* 3.3 (2000), S. 186–205. ISSN: 1094-9224. DOI: 10.1145/357830.357849. URL: https://dl.acm.org/doi/pdf/10.1145/357830.357849 (besucht am 26. 01. 2024).

[7] Jangseong Bae und Changki Lee. „Easy Data Augmentation for Improved Malware Detection: A Comparative Study“. In: *2021 IEEE International Conference on Big Data and Smart Computing (BigComp)*. ISSN: 2375-9356. Jan. 2021, S. 214–218. DOI: 10.1109/BigComp51126.2021.00048. URL: https://ieeexplore.ieee.org/document/9373283 (besucht am 13. 10. 2023).

[8] Tim Berghoff. *Warum Malware-Erkennung nicht einfach ist – Mythen rund um Erkennungsraten*. 2022. URL: https://www.gdata.de/blog/2022/07/37482-warum-malware-erkennung-nicht-einfach-ist (besucht am 26. 01. 2024).

[9] *CAPEv2*. 2023. URL: https://github.com/kevoreilly/CAPEv2/blob/4dccd3b89a7d52b0dab2244306d438373afbb738/README.md (besucht am 21. 06. 2023).

[10] Rich Caruana. „Multitask Learning“. In: *Machine Learning* 28.1 (1997), S. 41–75. ISSN: 1573-0565. DOI: 10.1023/A:1007379606734.

[11] Lorenzo Cavallaro u. a. „Are Machine Learning Models for Malware Detection Ready for Prime Time?“ In: *IEEE Secur. Priv.* 21.2 (2023), S. 53–56. ISSN: 1558-4046. DOI: 10.1109/MSEC.2023.3236543.

[12] Yinpeng Chen u. a. „Dynamic Convolution: Attention over Convolution Kernels“. In: *arXiv* (2019). DOI: 10.48550/arXiv.1912.03458. eprint: 1912.03458.

[13] Fred Cohen. „Computer viruses“. PhD Thesis. University of Southern California Janvier, 1985.

[14] Elijah Cole u. a. „Multi-Label Learning From Single Positive Labels“. In: 2021, S. 933–942. URL: https://openaccess.thecvf.com/content/CVPR2021/html/Cole_Multi-Label_Learning_From_Single_Positive_Labels_CVPR_2021_paper.html (besucht am 25. 05. 2023).

[15] Yann N. Dauphin u. a. *Language Modeling with Gated Convolutional Networks*. 2016. DOI: 10.48550/ARXIV.1612.08083.

[16] Dimitrios E. Diamantis und Dimitris K. Iakovidis. „Fuzzy Pooling“. In: *IEEE Transactions on Fuzzy Systems* 29.11 (Nov. 2021), S. 3481–3488. ISSN: 1941-0034. DOI: 10.1109/TFUZZ.2020.3024023.

[17] Steven H. H. Ding, Benjamin C. M. Fung und Philippe Charland. „Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization“. In: *2019 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. 2019, S. 472–489. DOI: 10.1109/SP.2019.00003.

[18] David Eigen, Marc'Aurelio Ranzato und Ilya Sutskever. *Learning Factored Representations in a Deep Mixture of Experts*. Techn. Ber. 2013. URL: http://arxiv.org/abs/1312.4314 (besucht am 13. 08. 2023).

[19] Ahmed Eissa. *Honeypot is a crucial defense layer in network security architecture. Honeypots appear to be a weak entry point into an organization network to distract attackers from looking at other sensitive systems.* 2023. URL: https://www.linkedin.com/pulse/honeypots-types-technologies-detection-techniques-tools-ahmed-eissa (besucht am 16. 05. 2023).

[20] William Engelmann. *Bioruebe/UniExtract2: Universal Extractor 2 is a tool to extract files from any type of archive or installer.* 2022. URL: https://github.com/Bioruebe/UniExtract2/blob/master/README.md (besucht am 15. 07. 2023).

[21] William Fleshman u. a. „Static Malware Detection & Subterfuge: Quantifying the Robustness of Machine Learning and Current Anti-Virus". In: *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*. 2018, S. 1–10. DOI: 10.1109/MALWARE.2018.8659360.

[22] Joint Task Force. *Security and Privacy Controls for Federal Information Systems and Organizations*. Techn. Ber. NIST Special Publication (SP) 800-53, Rev. 4, Includes updates as of January 22, 2015. Gaithersburg, MD: National Institute of Standards und Technology, 2013. DOI: 10.6028/NIST.SP.800-53r4.

[23] Joint Task Force. *Security and Privacy Controls for Federal Information Systems and Organizations*. Techn. Ber. NIST Special Publication (SP) 800-53, Rev. 5, Includes updates as of 12-10-2020. Gaithersburg, MD: National Institute of Standards und Technology, 2020. DOI: 10.6028/NIST.SP.800-53r5.

[24] Kunihiko Fukushima. „Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". en. In: *Biological Cybernetics* 36.4 (Apr. 1980), S. 193–202. ISSN: 1432-0770. DOI: 10.1007/BF00344251.

[25] Bundesamt für Sicherheit in der Informationstechnik. *Adware und Spyware – wo liegen die Unterschiede?* 2023. URL: https://www.bsi.bund.de/DE/Themen/Verbraucherinnen-und-Verbraucher/Cyber-Sicherheitslage/Methoden-der-Cyber-Kriminalitaet/Schadprogramme/Adware-und-Spyware/adware-und-spyware_node.html (besucht am 27. 05. 2023).

[26] Bundesamt für Sicherheit in der Informationstechnik. *Backdoor*. 2023. URL: https://www.bsi.bund.de/SharedDocs/Glossareintraege/DE/B/Backdoor.html (besucht am 27. 05. 2023).

[27] Bundesamt für Sicherheit in der Informationstechnik. *Botnetze – Auswirkungen und Schutzmaßnahmen*. 2023. URL: https://www.bsi.bund.de/DE/Themen/Verbraucherinnen-und-Verbraucher/Cyber-Sicherheitslage/Methoden-der-Cyber-Kriminalitaet/Botnetze/botnetze_node.html (besucht am 24. 05. 2023).

[28] Bundesamt für Sicherheit in der Informationstechnik. *Dropper*. 2023. URL: https://www.bsi.bund.de/SharedDocs/Glossareintraege/DE/D/Dropper.html (besucht am 27. 05. 2023).

[29] Bundesamt für Sicherheit in der Informationstechnik. *Malware*. URL: https://www.bsi.bund.de/DE/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/Empfehlungen-nach-Gefaehrdungen/Malware/malware.html?nn=522742 (besucht am 24. 05. 2023).

[30] Bundesamt für Sicherheit in der Informationstechnik. *Ransomware – Vorsicht vor Erpressersoftware*. 2023. URL: https://www.bsi.bund.de/DE/Themen/Verbraucherinnen-und-Verbraucher/Cyber-Sicherheitslage/Methoden-der-Cyber-Kriminalitaet/Schadprogramme/Ransomware/ransomware_node.html (besucht am 24. 05. 2023).

[31] Bundesamt für Sicherheit in der Informationstechnik. *Trojaner – wie erkenne ich getarnte Schadprogramme?* 2023. URL: https://www.bsi.bund.de/DE/Themen/Verbraucherinnen-und-Verbraucher/Cyber-Sicherheitslage/Methoden-der-Cyber-Kriminalitaet/Schadprogramme/Trojaner/trojaner_node.html (besucht am 24. 05. 2023).

[32] Bundesamt für Sicherheit in der Informationstechnik. *Viren und Würmer*. 2023. URL: https://www.bsi.bund.de/DE/Themen/Verbraucherinnen-und-Verbraucher/Cyber-Sicherheitslage/Methoden-der-Cyber-Kriminalitaet/Schadprogramme/Viren-und-Wuermer/viren-und-wuermer_node.html (besucht am 24. 05. 2023).

[33] Daniel Gibert, Carles Mateu und Jordi Planes. „A Hierarchical Convolutional Neural Network for Malware Classification“. In: (2019). DOI: 10.1109/IJCNN.2019.8852469.

[34] Daniel Gibert, Carles Mateu und Jordi Planes. „HYDRA: A multimodal deep learning framework for malware classification“. In: *Computers & Security* 95 (2020). verfügbar in [36]. DOI: 10.1016/j.cose.2020.101873.

[35] Daniel Gibert, Carles Mateu und Jordi Planes. „The rise of machine learning for detection and classification of malware: Research developments, trends and challenges“. In: *Journal of Network and Computer Applications* 153 (2020. Eingereicht 2019), S. 102526. DOI: 10.1016/j.jnca.2019.102526.

[36] Daniel Gibert Llauradó. „Going Deep into the Cat and the Mouse Game: Deep Learning for Malware Classification“. Diss. Lleida, Spain: Universitat de Lleida, 2020. URL: http://hdl.handle.net/10803/671776 (besucht am 02. 06. 2023).

[37] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. MIT Press, 2016, S. 326–366. URL: http://www.deeplearningbook.org (besucht am 24. 06. 2023).

[38] Albert Gu und Tri Dao. *Mamba: Linear-Time Sequence Modeling with Selective State Spaces*. Techn. Ber. arXiv:2312.00752 [cs] type: article. arXiv, Dez. 2023. DOI: 10.48550/arXiv.2312.00752. URL: http://arxiv.org/abs/2312.00752 (besucht am 02. 02. 2024).

[39] Irfan Ul Haq und Juan Caballero. „A Survey of Binary Code Similarity“. In: *ACM Computing Surveys* 54.3 (Apr. 2021), 51:1–51:38. ISSN: 0360-0300. DOI: 10.1145/3446371.

[40] Jie Hu u. a. *Squeeze-and-Excitation Networks*. Techn. Ber. arXiv:1709.01507 [cs] type: article. arXiv, Mai 2019. URL: http://arxiv.org/abs/1709.01507 (besucht am 19. 08. 2023).

[41] Wenyi Huang und Jack W. Stokes. „MtNet: A Multi-Task Neural Network for Dynamic Malware Classification“. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Hrsg. von Juan Caballero, Urko Zurutuza und Ricardo J. Rodríguez. Cham: Springer International Publishing, 2016, S. 399–418. ISBN: 978-3-319-40667-1. URL: https://www.microsoft.com/en-us/research/publication/mtnet-multi-task-neural-network-dynamic-malware-classification/ (besucht am 03. 06. 2023).

[42] DeLesley Hutchins u. a. *Block-Recurrent Transformers*. Techn. Ber. arXiv:2203.07852 [cs] type: article. arXiv, Nov. 2022. DOI: 10.48550/arXiv.2203.07852. URL: http://arxiv.org/abs/2203.07852 (besucht am 22. 06. 2023).

[43] Intel Corporation. „Instruction Set Reference, A-Z“. In: *Intel® 64 and IA-32 Architectures Software Developer's Manual* 2 (2A, 2B, 2C, & 2D).253665 (2023). URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (besucht am 21. 05. 2023).

[44] ItzLevvie. *Teracopy package hijacked? Is it really this easy to take control of a package?* Okt. 2021. URL: https://github.com/microsoft/winget-pkgs/discussions/31547#discussioncomment-1530984 (besucht am 14. 06. 2023).

[45] Seungho Jeon und Jongsub Moon. „Malware-Detection Method with a Convolutional Recurrent Neural Network Using Opcode Sequences“. In: *Information Sciences* 535 (2020), S. 1–15. ISSN: 0020-0255. DOI: 10.1016/j.ins.2020.05.026.

[46] Karl-Bridge-Microsoft. *PE-Format - Win32 apps*. 2023. URL: https://learn.microsoft.com/de-de/windows/win32/debug/pe-format (besucht am 17. 05. 2023).

[47] Kaleem Nawaz Khan u. a. „Op2Vec: An Opcode Embedding Technique and Dataset Design for End-to-End Detection of Android Malware“. In: *Security and Communication Networks* 2022 (2022), e3710968. ISSN: 1939-0114. DOI: 10.1155/2022/3710968.

[48] Jonas Knupp. „Einfuhrung in Deep Learning – LSTM & CNN“. Magisterarb. Technische Universitat München, 2017. URL: https://www5.in.tum.de/lehre/seminare/datamining/ss17/paper_pres/13_nn_deep/essay.pdf (besucht am 24. 06. 2023).

[49] Marek Krčál u. a. „Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only“. In: *OpenReview* (2018). URL: https://openreview.net/forum?id=HkHrmM1PM (besucht am 12. 05. 2023).

[50] Marek Krčál u. a. „Deep Convolutional Malware Classifiers Can Learn from Raw Executables and Labels Only | OpenReview“. In: *OpenReview* (2018). Forum. URL: https://openreview.net/forum?id=HkHrmM1PM (besucht am 12. 05. 2023).

[51] Frank La. *Künstlich intelligent: Wie lernen neuronale Netze?* de-de. 2019. URL: https://learn.microsoft.com/de-de/archive/msdn-magazine/2019/april/artificially-intelligent-how-do-neural-networks-learn (besucht am 23. 09. 2023).

[52] Y. LeCun. „Generalization and Network Design Strategies“. In: *Connectionism in Perspective*. Hrsg. von R. Pfeifer u. a. an extended version was published as a technical report of the University of Toronto. Zurich, Switzerland: Elsevier, 1989. URL: http://yann.lecun.com/exdb/publis/pdf/lecun-89.pdf (besucht am 16. 07. 2023).

[53] Y. Lecun u. a. „Gradient-based learning applied to document recognition“. In: *Proceedings of the IEEE* 86.11 (Nov. 1998), S. 2278–2324. ISSN: 1558-2256. DOI: 10.1109/5.726791.

[54] Yann LeCun u. a. „Handwritten Digit Recognition with a Back-Propagation Network“. In: *Advances in Neural Information Processing Systems*. Bd. 2. Morgan-Kaufmann, 1989. URL: https://proceedings.neurips.cc/paper/1989/hash/53c3bce66e43be4f209556518c2fcb54-Abstract.html (besucht am 11. 07. 2023).

[55] Hyunjong Lee u. a. „Robust IoT Malware Detection and Classification Using Opcode Category Features on Machine Learning“. In: *IEEE Access* 11 (2023), S. 18855–18867. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2023.3247344. (Besucht am 16. 05. 2023).

[56] Miles Q. Li u. a. „I-MAD: Interpretable malware detector using Galaxy Transformer“. In: *Computers & Security* 108 (2021), S. 102371. ISSN: 0167-4048. DOI: 10.1016/j.cose.2021.102371.

[57] *Malware Protection Test September 2023*. 2023. URL: https://www.av-comparatives.org/tests/malware-protection-test-september-2023 (besucht am 26. 01. 2024).

[58] Niall McLaughlin und Jesus Martinez del Rincon. *Data Augmentation for Opcode Sequence Based Malware Detection*. Techn. Ber. arXiv:2106.11821 [cs] type: article. arXiv, März 2022. URL: http://arxiv.org/abs/2106.11821 (besucht am 11. 01. 2024).

[59] Tomas Mikolov u. a. „Efficient Estimation of Word Representations in Vector Space“. In: *arXiv* (2013). DOI: 10.48550/arXiv.1301.3781. eprint: 1301.3781.

[60] L. Nataraj u. a. „Malware images: visualization and automatic classification“. In: *VizSec '11: Proceedings of the 8th International Symposium on Visualization for Cyber Security*. New York, NY, USA: Association for Computing Machinery, 2011, S. 1–7. ISBN: 978-1-45030679-9. DOI: 10.1145/2016904.2016908.

[61] Jonathan Oliver. *TLSH - A Locality Sensitive Hash*. Online. Trend Micro, 2021. URL: https://tlsh.org (besucht am 01.06.2023).

[62] Jonathan Oliver. *TLSH - Blog. Notes on Function Re-ordering*. Online. Trend Micro, 2021. URL: https://tlsh.org/blog.html (besucht am 01.06.2023).

[63] Jonathan Oliver, Chun Cheng und Yanggui Chen. „TLSH - A Locality Sensitive Hash“. In: Trend Micro. Trend Micro, 2013. URL: https://github.com/trendmicro/tlsh/blob/master/TLSH_CTC_final.pdf (besucht am 01.06.2023).

[64] Feargus Pendlebury u. a. „TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time“. In: *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 2019, S. 729–746. ISBN: 9781939133069. URL: https://www.usenix.org/conference/usenixsecurity19/presentation/pendlebury (besucht am 29.05.2023).

[65] Christophe Pere. *What are Loss Functions?* en. Juni 2020. URL: https://towardsdatascience.com/what-is-loss-function-1e2605aeb904 (besucht am 23.09.2023).

[66] Konstantinos Poulinakis. *Are Transformers replacing CNNs in Object Detection?* 2023. URL: https://www.picsellia.com/post/are-transformers-replacing-cnns-in-object-detection (besucht am 28.10.2023).

[67] Edward Raff u. a. „Malware Detection by Eating a Whole EXE“. In: *arXiv* (2017). DOI: 10.48550/arXiv.1710.09435.

[68] Charles Nicholas Richard Zak Edward Raff. „What Can N-Grams Learn for Malware Detection?“ In: *12th International Conference on Malicious and Unwanted Software (MALWARE)* (2017). URL: https://rjzak.github.io/what_can_ngrams_learn.pdf (besucht am 18.05.2023).

[69] Felp Roza. „End-to-end learning, the (almost) every purpose ML method“. In: *Medium* (2022). URL: https://towardsdatascience.com/e2e-the-every-purpose-ml-method-5d4f20dafee4 (besucht am 18.05.2023).

[70] Ethan M. Rudd u. a. „ALOHA: auxiliary loss optimization for hypothesis augmentation“. In: *Proceedings of the 28th USENIX Conference on Security Symposium*. SEC'19. USA: USENIX Association, 2019, S. 303–320. ISBN: 9781939133069. (Besucht am 02.05.2023).

[71] David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams. „Learning representations by back-propagating errors“. In: *Nature* 323.6088 (1986), S. 533–536. DOI: 10.1038/323533a0.

[72] Asaf Shabtai u. a. „Detecting unknown malicious code by applying classification techniques on OpCode patterns“. In: *Secur. Inform.* 1.1 (2012), S. 1–22. ISSN: 2190-8532. DOI: 10.1186/2190-8532-1-1.

[73] Asaf Shabtai u. a. „Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey“. In: *Information Security Technical Report* 14.1 (2009), S. 16–29. ISSN: 1363-4127. DOI: 10.1016/j.istr.2009.03.003.

[74] Arindam Sharma, Pasquale Malacaria und MHR Khouzani. „Malware Detection Using 1-Dimensional Convolutional Neural Networks“. In: *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2019. DOI: 10.1109/eurospw.2019.00034.

[75] Noam Shazeer u. a. „Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer“. In: *arXiv* (2017). DOI: 10.48550/arXiv.1701.06538. eprint: 1701.06538.

[76] Laurent Sifre. „Rigid-Motion Scattering For Image Classification“. Diss. Ecole Polytechnique, CMAP, 2014. URL: https://www.di.ens.fr/data/publications/papers/phd_sifre.pdf (besucht am 10. 07. 2023).

[77] Michael R. Smith u. a. „Mind the Gap: On Bridging the Semantic Gap between Machine Learning and Malware Analysis“. In: *Proceedings of the 13th ACM Workshop on Artificial Intelligence and Security*. AISec'20. New York, NY, USA: Association for Computing Machinery, 2020, S. 49–60. ISBN: 9781450380942. DOI: 10.1145/3411508.3421373.

[78] SonicWall Inc. „2022 SonicWall Cyber Threat Report“. In: (2022). URL: https://www.sonicwall.com/medialibrary/en/white-paper/2022-sonicwall-cyber-threat-report.pdf (besucht am 02. 05. 2023).

[79] SonicWall Inc. „2023 SonicWall Cyber Threat Report“. In: (2023). URL: https://www.sonicwall.com/medialibrary/en/white-paper/2023-cyber-threat-report.pdf (besucht am 03. 06. 2023).

[80] Jost Tobias Springenberg u. a. „Striving for Simplicity: The All Convolutional Net“. In: (Apr. 2015). arXiv:1412.6806 [cs] type: article. DOI: 10.48550/arXiv.1412.6806.

[81] Christian Szegedy u. a. „Going deeper with convolutions“. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. ISSN: 1063-6919. Juni 2015, S. 1–9. DOI: 10.1109/CVPR.2015.7298594.

[82] *tf.nn.convolution | TensorFlow v2.12.0*. 2023. URL: https://www.tensorflow.org/api_docs/python/tf/nn/convolution (besucht am 24. 06. 2023).

[83] Jonathan Tompson u. a. *Efficient Object Localization Using Convolutional Networks*. Techn. Ber. arXiv:1411.4280 [cs] type: article. arXiv, Juni 2015. DOI: 10.48550/arXiv.1411.4280. URL: http://arxiv.org/abs/1411.4280 (besucht am 06. 01. 2024).

[84] *Types of Dependencies Handled By Dependency Walker*. 2019. URL: https://www.dependencywalker.com/help/html/dependency_types.htm (besucht am 25. 05. 2023).

[85] A. Waibel u. a. „Phoneme recognition using time-delay neural networks“. In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37.3 (März 1989), S. 328–339. ISSN: 0096-3518. DOI: 10.1109/29.21701.

[86] Brandon Yang u. a. *CondConv: Conditionally Parameterized Convolutions for Efficient Inference*. Techn. Ber. arXiv:1904.04971 [cs] type: article. arXiv, Sep. 2019. DOI: 10.48550/arXiv.1904.04971. URL: http://arxiv.org/abs/1904.04971 (besucht am 29. 06. 2023).

A Anhang

A.1 Vergleich der Methoden zur Überprüfung der Ähnlichkeit

1: BellSoft\LibericaJDK-19-Full\bin\java.exe
2: Amazon Corretto\jdk19.0.2_7\bin\java.exe
3: openjdk-17.0.2\bin\java.exe
4: java\zulu16\java.exe
5: AdoptOpenJDK\jdk-16.0.1.9-hotspot\bin\java.exe
6: azul-13.0.11\bin\java.exe
7: jdk-11.0.18.10-hotspot\bin\java.exe
8: minecraft\Install\java\Jre_8\bin\java.exe
9: Zulu\zulu-8\bin\java.exe
10: openlogic-openjdk-8u372-b07-windows-64\bin\java.exe
11: WindowsApps\40174MouriNaruto.NanaZip_2.0.450.0_x64__
gnj4mf6z9tkrc\NanaZip.exe
12: NVIDIA Corporation\NVIDIA GeForce Experience\7z.exe
13: MiniTool Partition Wizard 12\7z.exe
14: WingetUI\choco-cli\tools\7z.exe
15: JetBrains\Toolbox\bin\7z.exe
16: ExamDiff Pro\Plug-Ins\7-Zip\7z.exe
17: Microsoft Visual Studio\2022\Community\Common7\IDE\
CommonExtensions\Microsoft\TeamFoundation\Team Explorer\Git\
mingw64\bin\git.exe
18: Git\mingw64\libexec\git-core\git.exe
19: Postman\app-10.11.1\resources\data\git\mingw64\bin\git.exe
20: Postman\app-10.11.1\resources\data\git\cmd\git.exe
21: Git\cmd\git.exe

22: `MiKTeX\miktex\bin\x64\zip.exe`
23: `Windows Kits\10\Windows Performance Toolkit\wpa.exe`
24: `MiKTeX\miktex\bin\x64\biber.exe`
25: `jdk-11.0.18.10-hotspot\bin\unpack200.exe`

Alle PEs, die als ähnlich erkannt wurden —der Vergleichswert also unter dem Schwellwert liegt—, sind mit der gleichen Farbe hinterlegt. Wenn jedoch keine große Ähnlichkeit zu erwarten ist und manuell auch nicht festgestellt werden konnte, dann ist der Wert in Rot geschrieben. Die Zelle ist dann in der Farbe der niedrigeren Nummer markiert. Wenn jedoch zwei Dateien aufgrund eines manuellen Vergleichs über ExamDiff Pro als ähnlich eingestuft wurden, diese jedoch als unterschiedlich erkannt wurden, dann ist der Wert ebenfalls in Rot geschrieben. Die Schwellwerte wurden mit folgender Formel berechnet: $\frac{\max(S)+D_{0.1}}{2}+1$, wobei S die Menge der Ähnlichkeitsbewertungen der PEs ist, die als ähnlich erkannt werden sollen, und D die der PEs, die größere Unterschiede zueinander aufweisen. $D_{0.1}$ sei dabei definiert als $\{d_i | d_i \in D, i \leq 0.1 \cdot |D|\}$, ist also die Menge der 10% kleinsten Werte der Menge D. Wenn der Vergleich anhand des TLSH durchgeführt wurde, dann wurde der Schwellwert in das Intervall $[50, 100]$ erzwungen. Die Eingrenzung zwischen 50 und 100 ist damit begründet, dass laut Oliver, Cheng u. a. [63, S. 5] in diesem Bereich die Erkennungsrate noch relativ hoch und die Falsch-Positiv-Rate noch relativ gering ist. Falls der Schwellwert ohne der Eingrenzung anders wäre, dann ist dieser in Klammern hinter dem Schwellwert angegeben.

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	1	63	63	49	24	24	203	219	212	144	152	148
2	1	0	63	63	49	24	24	203	219	212	144	152	148
3	63	63	0	1	37	93	93	163	180	166	117	137	131
4	63	63	1	0	37	93	93	163	180	166	117	137	131
5	49	49	37	37	0	62	62	206	201	211	151	157	143
6	24	24	93	93	62	0	1	221	213	229	164	160	150
7	24	24	93	93	62	1	0	221	213	229	164	160	150
8	203	203	163	163	206	221	221	0	65	41	154	168	184
9	219	219	180	180	201	213	213	65	0	61	175	190	210
10	212	212	166	166	211	229	229	41	61	0	150	162	182
11	144	144	117	117	151	164	164	154	175	150	0	96	109
12	152	152	137	137	157	160	160	168	190	162	96	0	16
13	148	148	131	131	143	150	150	184	210	182	109	16	0
14	288	288	241	241	258	274	274	159	137	153	194	245	256
15	280	280	231	231	250	267	267	159	134	150	193	241	250
16	280	280	231	231	250	267	267	159	134	150	193	241	250
17	199	199	160	160	198	203	203	130	146	128	148	140	154
18	199	199	160	160	198	203	203	130	146	128	148	140	154
19	191	191	154	154	197	200	200	125	148	126	145	133	149
20	147	147	162	162	160	139	139	220	247	210	175	132	120
21	147	147	187	187	165	135	135	251	256	250	210	156	148
22	225	225	175	175	219	248	248	71	73	61	156	181	199
23	140	140	103	103	144	168	168	134	159	141	71	113	126
24	125	125	102	102	114	138	138	213	223	219	109	107	115
25	202	202	165	165	202	216	216	83	87	81	156	176	192

Tabelle A.1: TLSH der Codebytes. Schwellwert: 90. (Part 1 von 2)

	14	15	16	17	18	19	20	21	22	23	24	25
1	288	280	280	199	199	191	147	147	225	140	125	202
2	288	280	280	199	199	191	147	147	225	140	125	202
3	241	231	231	160	160	154	162	187	175	103	102	165
4	241	231	231	160	160	154	162	187	175	103	102	165
5	258	250	250	198	198	197	160	165	219	144	114	202
6	274	267	267	203	203	200	139	135	248	168	138	216
7	274	267	267	203	203	200	139	135	248	168	138	216
8	159	159	159	130	130	125	220	251	71	134	213	83
9	137	134	134	146	146	148	247	256	73	159	223	87
10	153	150	150	128	128	126	210	250	61	141	219	81
11	194	193	193	148	148	145	175	210	156	71	109	156
12	245	241	241	140	140	133	132	156	181	113	107	176
13	256	250	250	154	154	149	120	148	199	126	115	192
14	0	21	21	170	170	180	271	268	154	203	262	150
15	21	0	1	161	161	175	265	263	152	207	263	144
16	21	1	0	161	161	175	265	263	152	207	263	144
17	170	161	161	0	0	23	128	181	139	147	202	132
18	170	161	161	0	0	23	128	181	139	147	202	132
19	180	175	175	23	23	0	128	174	143	138	194	130
20	271	265	265	128	128	128	0	48	218	185	168	208
21	268	263	263	181	181	174	48	0	250	222	184	225
22	154	152	152	139	139	143	218	250	0	131	211	84
23	203	207	207	147	147	138	185	222	131	0	112	144
24	262	263	263	202	202	194	168	184	211	112	0	211
25	150	144	144	132	132	130	208	225	84	144	211	0

Tabelle A.2: TLSH der Codebytes. Schwellwert: 90. (Part 2 von 2)

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	0	18	18	17	5	5	137	149	159	132	160	168
2	0	0	18	18	17	5	5	137	149	159	132	160	168
3	18	18	0	0	2	14	14	152	165	175	133	162	170
4	18	18	0	0	2	14	14	152	165	175	133	162	170
5	17	17	2	2	0	13	13	151	164	174	132	161	169
6	5	5	14	14	13	0	0	149	162	172	137	163	171
7	5	5	14	14	13	0	0	149	162	172	137	163	171
8	137	137	152	152	151	149	149	0	17	24	128	61	70
9	149	149	165	165	164	162	162	17	0	12	153	86	100
10	159	159	175	175	174	172	172	24	12	0	169	97	107
11	132	132	133	133	132	137	137	128	153	169	0	109	97
12	160	160	162	162	161	163	163	61	86	97	109	0	8
13	168	168	170	170	169	171	171	70	100	107	97	8	0
14	137	137	150	150	149	145	145	114	89	86	166	154	168
15	148	148	161	161	160	156	156	102	77	74	178	142	156
16	148	148	161	161	160	156	156	102	77	74	178	142	156
17	170	170	187	187	186	181	181	111	89	77	165	163	177
18	170	170	187	187	186	181	181	111	89	77	165	163	177
19	170	170	184	184	183	181	181	115	89	75	171	167	181
20	190	190	197	197	196	190	190	86	109	123	126	66	65
21	201	201	209	209	208	202	202	108	134	148	112	75	64
22	162	162	174	174	173	173	173	77	59	41	175	149	161
23	156	156	154	154	155	158	158	86	111	125	90	67	58
24	195	195	191	191	195	210	210	262	237	230	195	260	268
25	166	166	187	187	186	179	179	54	39	34	177	114	128

Tabelle A.3: TLSH der Opcode-Sequenzen mit Funktionsnamen. Schwellwert: 50(40). (Part 1 von 2)

	14	15	16	17	18	19	20	21	22	23	24	25
1	137	148	148	170	170	170	190	201	162	156	195	166
2	137	148	148	170	170	170	190	201	162	156	195	166
3	150	161	161	187	187	184	197	209	174	154	191	187
4	150	161	161	187	187	184	197	209	174	154	191	187
5	149	160	160	186	186	183	196	208	173	155	195	186
6	145	156	156	181	181	181	190	202	173	158	210	179
7	145	156	156	181	181	181	190	202	173	158	210	179
8	114	102	102	111	111	115	86	108	77	86	262	54
9	89	77	77	89	89	89	109	134	59	111	237	39
10	86	74	74	77	77	75	123	148	41	125	230	34
11	166	178	178	165	165	171	126	112	175	90	195	177
12	154	142	142	163	163	167	66	75	149	67	260	114
13	168	156	156	177	177	181	65	64	161	58	268	128
14	0	2	2	86	86	86	175	202	80	189	227	99
15	2	0	0	87	87	87	163	190	79	177	239	87
16	2	0	0	87	87	87	163	190	79	177	239	87
17	86	87	87	0	0	11	181	204	39	208	167	69
18	86	87	87	0	0	11	181	204	39	208	167	69
19	86	87	87	11	11	0	181	206	39	206	171	73
20	175	163	163	181	181	181	0	6	175	96	318	139
21	202	190	190	204	204	206	6	0	200	95	314	162
22	80	79	79	39	39	39	175	200	0	175	175	54
23	189	177	177	208	208	206	96	95	175	0	222	135
24	227	239	239	167	167	171	318	314	175	222	0	212
25	99	87	87	69	69	73	139	162	54	135	212	0

Tabelle A.4: TLSH der Opcode-Sequenzen mit Funktionsnamen. Schwellwert: 50(40). (Part 2 von 2)

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	0	17	17	27	10	10	147	161	133	185	195	195
2	0	0	17	17	27	10	10	147	161	133	185	195	195
3	17	17	0	0	5	30	30	133	145	119	187	195	195
4	17	17	0	0	5	30	30	133	145	119	187	195	195
5	27	27	5	5	0	41	41	120	132	106	200	182	182
6	10	10	30	30	41	0	0	144	158	130	162	170	170
7	10	10	30	30	41	0	0	144	158	130	162	170	170
8	147	147	133	133	120	144	144	0	30	8	105	103	101
9	161	161	145	145	132	158	158	30	0	27	92	94	92
10	133	133	119	119	106	130	130	8	27	0	115	113	111
11	185	185	187	187	200	162	162	105	92	115	0	40	40
12	195	195	195	195	182	170	170	103	94	113	40	0	3
13	195	195	195	195	182	170	170	101	92	111	40	3	0
14	157	157	185	185	198	158	158	211	199	221	136	155	157
15	181	181	209	209	222	182	182	187	175	197	113	131	133
16	181	181	209	209	222	182	182	187	175	197	113	131	133
17	199	199	209	209	220	174	174	127	115	137	59	82	84
18	199	199	209	209	220	174	174	127	115	137	59	82	84
19	163	163	197	197	208	162	162	163	151	173	93	117	119
20	185	185	213	213	224	182	182	171	159	181	89	106	108
21	199	199	203	203	214	172	172	157	147	167	78	95	97
22	107	107	89	89	76	104	104	34	50	23	137	137	135
23	161	161	131	131	120	164	164	66	57	63	78	76	76
24	123	123	134	134	147	137	137	274	238	258	190	202	199
25	201	201	209	209	222	176	176	97	81	105	52	61	63

Tabelle A.5: TLSH der Opcode-Sequenzen ohne Funktionsnamen. Schwellwert: 50(41). (Part 1 von 2)

	14	15	16	17	18	19	20	21	22	23	24	25
1	157	181	181	199	199	163	185	199	107	161	123	201
2	157	181	181	199	199	163	185	199	107	161	123	201
3	185	209	209	209	209	197	213	203	89	131	134	209
4	185	209	209	209	209	197	213	203	89	131	134	209
5	198	222	222	220	220	208	224	214	76	120	147	222
6	158	182	182	174	174	162	182	172	104	164	137	176
7	158	182	182	174	174	162	182	172	104	164	137	176
8	211	187	187	127	127	163	171	157	34	66	274	97
9	199	175	175	115	115	151	159	147	50	57	238	81
10	221	197	197	137	137	173	181	167	23	63	258	105
11	136	113	113	59	59	93	89	78	137	78	190	52
12	155	131	131	82	82	117	106	95	137	76	202	61
13	157	133	133	84	84	119	108	97	135	76	199	63
14	0	3	3	109	109	84	108	126	227	214	185	121
15	3	0	0	96	96	74	97	115	227	190	198	98
16	3	0	0	96	96	74	97	115	227	190	198	98
17	109	96	96	0	0	16	33	30	161	139	214	36
18	109	96	96	0	0	16	33	30	161	139	214	36
19	84	74	74	16	16	0	34	41	197	175	178	59
20	108	97	97	33	33	34	0	12	201	167	186	72
21	126	115	115	30	30	41	12	0	187	157	198	71
22	227	227	227	161	161	197	201	187	0	70	221	139
23	214	190	190	139	139	175	167	157	70	0	189	119
24	185	198	198	214	214	178	186	198	221	189	0	205
25	121	98	98	36	36	59	72	71	139	119	205	0

Tabelle A.6: TLSH der Opcode-Sequenzen ohne Funktionsnamen. Schwellwert: 50(41). (Part 2 von 2)

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	0	4	4	4	0	0	122	119	123	107	127	130
2	0	0	4	4	4	0	0	122	119	123	107	127	130
3	4	4	0	0	0	4	4	124	120	125	108	129	132
4	4	4	0	0	0	4	4	124	120	125	108	129	132
5	4	4	0	0	0	4	4	124	120	125	108	129	132
6	0	0	4	4	4	0	0	122	119	123	107	127	130
7	0	0	4	4	4	0	0	122	119	123	107	127	130
8	122	122	124	124	124	122	122	0	19	30	101	126	130
9	119	119	120	120	120	119	119	19	0	36	103	127	132
10	123	123	125	125	125	123	123	30	36	0	102	125	130
11	107	107	108	108	108	107	107	101	103	102	0	92	92
12	127	127	129	129	129	127	127	126	127	125	92	0	5
13	130	130	132	132	132	130	130	130	132	130	92	5	0
14	127	127	129	129	129	127	127	126	127	125	93	60	65
15	127	127	129	129	129	127	127	126	127	125	93	60	65
16	127	127	129	129	129	127	127	126	127	125	93	60	65
17	108	108	109	109	109	108	108	97	96	97	148	93	93
18	108	108	109	109	109	108	108	97	96	97	148	93	93
19	109	109	110	110	110	109	109	98	99	99	153	96	96
20	150	150	152	152	152	150	150	145	144	145	113	112	113
21	151	151	153	153	153	151	151	145	144	145	113	111	111
22	119	119	121	121	121	119	119	63	68	42	106	134	136
23	86	86	88	88	88	86	86	113	114	130	116	158	159
24	67	67	64	64	64	67	67	142	138	144	113	145	149
25	53	53	56	56	56	53	53	143	139	145	113	144	148

Tabelle A.7: Vergleich anhand der Funktionen. Schwellwert: 47. (Part 1 von 2)

	14	15	16	17	18	19	20	21	22	23	24	25
1	127	127	127	108	108	109	150	151	119	86	67	53
2	127	127	127	108	108	109	150	151	119	86	67	53
3	129	129	129	109	109	110	152	153	121	88	64	56
4	129	129	129	109	109	110	152	153	121	88	64	56
5	129	129	129	109	109	110	152	153	121	88	64	56
6	127	127	127	108	108	109	150	151	119	86	67	53
7	127	127	127	108	108	109	150	151	119	86	67	53
8	126	126	126	97	97	98	145	145	63	113	142	143
9	127	127	127	96	96	99	144	144	68	114	138	139
10	125	125	125	97	97	99	145	145	42	130	144	145
11	93	93	93	148	148	153	113	113	106	116	113	113
12	60	60	60	93	93	96	112	111	134	158	145	144
13	65	65	65	93	93	96	113	111	136	159	149	148
14	0	0	0	107	107	110	138	136	133	157	144	144
15	0	0	0	107	107	110	138	136	133	157	144	144
16	0	0	0	107	107	110	138	136	133	157	144	144
17	107	107	107	0	0	12	85	85	99	116	113	113
18	107	107	107	0	0	12	85	85	99	116	113	113
19	110	110	110	12	12	0	86	87	103	119	114	115
20	138	138	138	85	85	86	0	2	133	148	174	175
21	136	136	136	85	85	87	2	0	133	147	178	177
22	133	133	133	99	99	103	133	133	0	145	136	137
23	157	157	157	116	116	119	148	147	145	0	91	84
24	144	144	144	113	113	114	174	178	136	91	0	86
25	144	144	144	113	113	115	175	177	137	84	86	0

Tabelle A.8: Vergleich anhand der Funktionen. Schwellwert: 47. (Part 2 von 2)

Der Vergleich der Funktion wurde anhand folgender auf Seite 43 beschriebenen Formel durchgeführt: $\frac{|A \triangle B|}{\max(|A|,|B|)}$

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	0	50	50	50	0	0	223	246	193	212	153	151
2	0	0	50	50	50	0	0	223	246	193	212	153	151
3	50	50	0	0	0	50	50	219	239	211	220	176	190
4	50	50	0	0	0	50	50	219	239	211	220	176	190
5	50	50	0	0	0	50	50	219	239	211	220	176	190
6	0	0	50	50	50	0	0	223	246	193	212	153	151
7	0	0	50	50	50	0	0	223	246	193	212	153	151
8	223	223	219	219	219	223	223	0	15	22	144	133	124
9	246	246	239	239	239	246	246	15	0	35	140	159	148
10	193	193	211	211	211	193	193	22	35	0	113	108	99
11	212	212	220	220	220	212	212	144	140	113	0	124	139
12	153	153	176	176	176	153	153	133	159	108	124	0	9
13	151	151	190	190	190	151	151	124	148	99	139	9	0
14	176	176	182	182	182	176	176	176	174	155	99	92	107
15	176	176	182	182	182	176	176	176	174	155	99	92	107
16	176	176	182	182	182	176	176	176	174	155	99	92	107
17	161	161	198	198	198	161	161	183	211	158	157	50	52
18	161	161	198	198	198	161	161	183	211	158	157	50	52
19	166	166	199	199	199	166	166	186	216	163	160	54	60
20	274	274	311	311	311	274	274	156	149	183	204	180	165
21	250	250	309	309	309	250	250	169	164	172	206	158	143
22	179	179	199	199	199	179	179	29	50	14	127	94	85
23	86	86	103	103	103	86	86	285	285	258	175	175	183
24	189	189	126	126	126	189	189	207	191	224	213	286	299
25	52	52	88	88	88	52	52	218	241	189	235	200	186

Tabelle A.9: TLSH der Funktionen. Schwellwert: 54. (Part 1 von 2)

	14	15	16	17	18	19	20	21	22	23	24	25
1	176	176	176	161	161	166	274	250	179	86	189	52
2	176	176	176	161	161	166	274	250	179	86	189	52
3	182	182	182	198	198	199	311	309	199	103	126	88
4	182	182	182	198	198	199	311	309	199	103	126	88
5	182	182	182	198	198	199	311	309	199	103	126	88
6	176	176	176	161	161	166	274	250	179	86	189	52
7	176	176	176	161	161	166	274	250	179	86	189	52
8	176	176	176	183	183	186	156	169	29	285	207	218
9	174	174	174	211	211	216	149	164	50	285	191	241
10	155	155	155	158	158	163	183	172	14	258	224	189
11	99	99	99	157	157	160	204	206	127	175	213	235
12	92	92	92	50	50	54	180	158	94	175	286	200
13	107	107	107	52	52	60	165	143	85	183	299	186
14	0	0	0	131	131	136	246	248	139	142	244	218
15	0	0	0	131	131	136	246	248	139	142	244	218
16	0	0	0	131	131	136	246	248	139	142	244	218
17	131	131	131	0	0	13	135	113	144	169	304	218
18	131	131	131	0	0	13	135	113	144	169	304	218
19	136	136	136	13	13	0	139	115	147	170	307	223
20	246	246	246	135	135	139	0	5	192	261	261	274
21	248	248	248	113	113	115	5	0	171	261	283	250
22	139	139	139	144	144	147	192	171	0	241	241	173
23	142	142	142	169	169	170	261	261	241	0	179	118
24	244	244	244	304	304	307	261	283	241	179	0	158
25	218	218	218	218	218	223	274	250	173	118	158	0

Tabelle A.10: TLSH der Funktionen. Schwellwert: 54. (Part 2 von 2)

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	0	10	10	15	5	5	134	140	128	146	161	162
2	0	0	10	10	15	5	5	134	140	128	146	161	162
3	10	10	0	0	2	17	17	128	132	122	147	162	163
4	10	10	0	0	2	17	17	128	132	122	147	162	163
5	15	15	2	2	0	22	22	122	126	115	154	155	157
6	5	5	17	17	22	0	0	133	138	126	134	148	150
7	5	5	17	17	22	0	0	133	138	126	134	148	150
8	134	134	128	128	122	133	133	0	24	19	103	114	115
9	140	140	132	132	126	138	138	24	0	31	97	110	112
10	128	128	122	122	115	126	126	19	31	0	108	119	120
11	146	146	147	147	154	134	134	103	97	108	0	66	66
12	161	161	162	162	155	148	148	114	110	119	66	0	4
13	162	162	163	163	157	150	150	115	112	120	66	4	0
14	142	142	157	157	163	142	142	168	163	173	114	107	111
15	154	154	169	169	175	154	154	156	151	161	103	95	99
16	154	154	169	169	175	154	154	156	151	161	103	95	99
17	153	153	159	159	164	141	141	112	105	117	103	87	88
18	153	153	159	159	164	141	141	112	105	117	103	87	88
19	136	136	153	153	159	135	135	130	125	136	123	106	107
20	167	167	182	182	188	166	166	158	151	163	101	109	110
21	175	175	178	178	183	161	161	151	145	156	95	103	104
22	113	113	105	105	98	111	111	48	59	32	121	135	135
23	123	123	109	109	104	125	125	89	85	96	97	117	117
24	95	95	99	99	105	102	102	208	188	201	151	173	174
25	127	127	132	132	139	114	114	120	110	125	82	102	105

Tabelle A.11: Vergleich der Funktionen kombiniert mit TLSH der Opcode-Sequenzen ohne Funktionsnamen. Schwellwert: 50(46). (Part 1 von 2)

	14	15	16	17	18	19	20	21	22	23	24	25
1	142	154	154	153	153	136	167	175	113	123	95	127
2	142	154	154	153	153	136	167	175	113	123	95	127
3	157	169	169	159	159	153	182	178	105	109	99	132
4	157	169	169	159	159	153	182	178	105	109	99	132
5	163	175	175	164	164	159	188	183	98	104	105	139
6	142	154	154	141	141	135	166	161	111	125	102	114
7	142	154	154	141	141	135	166	161	111	125	102	114
8	168	156	156	112	112	130	158	151	48	89	208	120
9	163	151	151	105	105	125	151	145	59	85	188	110
10	173	161	161	117	117	136	163	156	32	96	201	125
11	114	103	103	103	103	123	101	95	121	97	151	82
12	107	95	95	87	87	106	109	103	135	117	173	102
13	111	99	99	88	88	107	110	104	135	117	174	105
14	0	1	1	108	108	97	123	131	180	185	164	132
15	1	0	0	101	101	92	117	125	180	173	171	121
16	1	0	0	101	101	92	117	125	180	173	171	121
17	108	101	101	0	0	14	59	57	130	127	163	74
18	108	101	101	0	0	14	59	57	130	127	163	74
19	97	92	92	14	14	0	60	64	150	147	146	87
20	123	117	117	59	59	60	0	7	167	157	180	123
21	131	125	125	57	57	64	7	0	160	152	188	124
22	180	180	180	130	130	150	167	160	0	107	178	138
23	185	173	173	127	127	147	157	152	107	0	140	101
24	164	171	171	163	163	146	180	188	178	140	0	145
25	132	121	121	74	74	87	123	124	138	101	145	0

Tabelle A.12: Vergleich der Funktionen kombiniert mit TLSH der Opcode-Sequenzen ohne Funktionsnamen. Schwellwert: 50(46). (Part 2 von 2)

Einblick in die Funktionen der Dateien:

PE Nr.	Anzahl DLL-Funktionen
1	48
2	48
3	50
4	50
5	50
6	48
7	48
8	102
9	108
10	103
11	316
12	131
13	125
14	133
15	133
16	133
17	399
18	399
19	369
20	88
21	86
22	126
23	128
24	73
25	72

Tabelle A.13: Anzahl der DLL-Funktionen der verglichenen PE-Dateien

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	0	2	2	2	0	0	124	128	127	338	167	163
2	0	0	2	2	2	0	0	124	128	127	338	167	163
3	2	2	0	0	0	2	2	126	130	129	340	169	165
4	2	2	0	0	0	2	2	126	130	129	340	169	165
5	2	2	0	0	0	2	2	126	130	129	340	169	165
6	0	0	2	2	2	0	0	124	128	127	338	167	163
7	0	0	2	2	2	0	0	124	128	127	338	167	163
8	124	124	126	126	126	124	124	0	20	31	318	165	163
9	128	128	130	130	130	128	128	20	0	39	326	167	165
10	127	127	129	129	129	127	127	31	39	0	321	164	162
11	338	338	340	340	340	338	338	318	326	321	0	291	291
12	167	167	169	169	169	167	167	165	167	164	291	0	6
13	163	163	165	165	165	163	163	163	165	162	291	6	0

Tabelle A.14: Absolute Anzahl der abweichenden Funktionen. (Part 1 von 2)

	14	15	16	17	18	19	20	21	22	23	24	25
14	0	0	0	428	428	406	183	181	177	209	192	191
15	0	0	0	428	428	406	183	181	177	209	192	191
16	0	0	0	428	428	406	183	181	177	209	192	191
17	428	428	428	0	0	49	340	338	397	463	452	451
18	428	428	428	0	0	49	340	338	397	463	452	451
19	406	406	406	49	49	0	318	320	379	439	422	423
20	183	183	183	340	340	318	0	2	168	190	153	154
21	181	181	181	338	338	320	2	0	168	188	153	152
22	177	177	177	397	397	379	168	168	0	186	171	172
23	209	209	209	463	463	439	190	188	186	0	117	108
24	192	192	192	452	452	422	153	153	171	117	0	63
25	191	191	191	451	451	423	154	152	172	108	63	0

Tabelle A.15: Absolute Anzahl der abweichenden Funktionen. (Part 2 von 2)

A.2 Hinweise zur Augmentation

Alle 6 der in Tabelle A.16 aufgeführten Sequenzen aus nur einer Anweisung wurden doppelt so häufig verwendet, wie die anderen aufgelisteten Sequenzen. Zu 13,5% wurde anstelle einer der aufgelisteten Sequenzen eine der 34 Sprunganweisungen für die Augmentation eingefügt.

Sequenz	Beispiel
mov	mov eax, eax
xor	xor eax, eaxeax
and	and eax, eax
xchg	xchg eax, eax
nop	nop
lea	lea eax, [eax]
push pop	push eax pop eax
inc dec	inc eax dec eax
dec inc	dec eax inc eax
inc mov dec	inc eax mov eax, eax dec eax
dec mov inc	dec eax mov eax, eax inc eax
inc xor dec	inc eax xor eax, eax dec eax
dec xor inc	dec eax xor eax, eax inc eax
push xor pop	push eax xor eax pop eax
push nop pop	push eax nop pop eax
push inc pop	push eax inc eax pop eax
push dec pop	push eax dec eax pop eax
push inc pop dec	push eax inc ebx pop eax dec ebx

Tabelle A.16: In der Augmentation verwendete Sequenzen ohne Sprunganweisungen mit Beispiel, wie eine Sequenz ohne Auswirkung aussehen kann.

A.3 Detaillierte Auflistung der Ergebnisse

Modell \ Grenzwert	0.1	0.2	0.3	0.4
SimpBase2	0.9924|0.3518	0.9900|0.2884	0.9874|0.2396	0.9827|0.1870
SimpBaseDense2	0.9759|0.1733	0.9628|0.1241	0.9497|0.0961	0.9348|0.0754
SimpBase	0.9790|0.1945	0.9707|0.1537	0.9621|0.1220	0.9553|0.1030
SimpBaseDense	0.9724|0.1423	0.9619|0.0993	0.9490|0.0671	0.9380|0.0495
Ref	0.9791|0.1576	0.9714|0.1153	0.9625|0.0892	0.9543|0.0638
Base	0.9798|0.1913	0.9733|0.1567	0.9661|0.1322	0.9573|0.1120
Ref2	0.9692|0.1085	0.9594|0.0785	0.9514|0.0562	0.9415|0.0433
BaseDense	0.9808|0.1930	0.9651|0.1246	0.9393|0.0696	0.9206|0.0431
SimpBaseDenseDyn2	0.9717|0.1352	0.9604|0.0873	0.9459|0.0609	0.9315|0.0436
SimpBaseDyn2	0.9553|0.0950	0.9416|0.0725	0.9287|0.0547	0.9144|0.0426
RefDyn2	0.9542|0.0605	0.9431|0.0412	0.9333|0.0307	0.9260|0.0216
BaseDenseDyn	0.9729|0.1464	0.9609|0.0992	0.9458|0.0692	0.9319|0.0468
SimpBaseDenseDyn	0.9819|0.1926	0.9713|0.1323	0.9579|0.0966	0.9480|0.0742
BaseDyn	0.9659|0.1128	0.9526|0.0872	0.9436|0.0694	0.9354|0.0574
SimpBaseDyn	0.9579|0.0966	0.9453|0.0695	0.9366|0.0555	0.9286|0.0472
RefDyn	0.9807|0.1638	0.9740|0.1222	0.9682|0.1010	0.9627|0.0832
BaseDenseDynMTL	0.9750|0.1432	0.9631|0.0923	0.9534|0.0651	0.9411|0.0475
SimpBaseDenseDynMTL	0.9830|0.1812	0.9683|0.1126	0.9504|0.0739	0.9309|0.0483
BaseDynMTL	0.9698|0.1623	0.9604|0.1228	0.9487|0.0902	0.9384|0.0709
SimpBaseDynMTL	0.9770|0.1603	0.9690|0.1236	0.9616|0.0993	0.9544|0.0829
RefDynMTL	0.9748|0.1444	0.9663|0.1055	0.9593|0.0835	0.9523|0.0654
BaseDenseMTL	0.9665|0.1520	0.9526|0.1108	0.9418|0.0843	0.9301|0.0657
SimpBaseMTL2	0.9600|0.1182	0.9441|0.0855	0.9346|0.0688	0.9236|0.0553
SimpBaseDenseMTL2	0.9777|0.1938	0.9686|0.1355	0.9567|0.0922	0.9430|0.0651
RefMTL2	0.9808|0.1657	0.9731|0.1236	0.9670|0.0946	0.9609|0.0749
RefMTL	0.9759|0.1384	0.9681|0.0993	0.9605|0.0682	0.9474|0.0521
SimpBaseDenseMTL	0.9729|0.1740	0.9521|0.0980	0.9346|0.0590	0.9160|0.0411
SimpBaseMTL	0.9797|0.2101	0.9709|0.1632	0.9617|0.1293	0.9524|0.0998
BaseMTL	0.9736|0.1610	0.9640|0.1204	0.9525|0.0904	0.9444|0.0726

Tabelle A.17: TPR | FPR aller Modelle je nach Grenzwert. (Part 1 von 2)

Modell \ Grenzwert	0.5	0.6	0.7	0.8	0.9
SimpBase2	0.9748|0.1522	0.9647|0.1205	0.9291|0.0756	0.8982|0.0491	0.8571|0.0275
SimpBaseDense2	0.9167|0.0536	0.8887|0.0308	0.8609|0.0196	0.8334|0.0099	0.7980|0.0049
SimpBase	0.9458|0.0835	0.9163|0.0503	0.8878|0.0331	0.8678|0.0228	0.8316|0.0100
SimpBaseDense	0.9250|0.0365	0.9086|0.0268	0.8923|0.0189	0.8686|0.0107	0.8420|0.0070
Ref	0.9429|0.0458	0.9324|0.0318	0.9073|0.0205	0.8845|0.0128	0.8512|0.0061
Base	0.9478|0.0878	0.9222|0.0506	0.8831|0.0321	0.8553|0.0209	0.8275|0.0129
Ref2	0.9308|0.0326	0.9168|0.0211	0.9005|0.0155	0.8767|0.0090	0.8425|0.0041
BaseDense	0.9039|0.0300	0.8805|0.0180	0.8593|0.0112	0.8391|0.0084	0.8146|0.0051
SimpBaseDenseDyn2	0.9182|0.0328	0.9037|0.0245	0.8802|0.0154	0.8602|0.0110	0.8383|0.0076
SimpBaseDyn2	0.8918|0.0253	0.8713|0.0170	0.8562|0.0116	0.8349|0.0084	0.7928|0.0051
RefDyn2	0.9144|0.0148	0.9011|0.0111	0.8848|0.0084	0.8631|0.0065	0.8363|0.0041
BaseDenseDyn	0.9177|0.0340	0.9070|0.0267	0.8905|0.0204	0.8682|0.0126	0.8447|0.0087
SimpBaseDenseDyn	0.9396|0.0597	0.9310|0.0467	0.9194|0.0324	0.8896|0.0211	0.8581|0.0127
BaseDyn	0.9235|0.0434	0.8992|0.0290	0.8752|0.0207	0.8523|0.0139	0.8289|0.0092
SimpBaseDyn	0.9184|0.0390	0.8991|0.0278	0.8660|0.0136	0.8367|0.0089	0.8089|0.0062
RefDyn	0.9560|0.0671	0.9492|0.0515	0.9358|0.0393	0.9141|0.0304	0.8893|0.0204
BaseDenseDynMTL	0.9265|0.0348	0.9101|0.0244	0.8897|0.0157	0.8593|0.0101	0.8280|0.0064
SimpBaseDenseDynMTL	0.9106|0.0328	0.8870|0.0198	0.8676|0.0134	0.8433|0.0085	0.8184|0.0047
BaseDynMTL	0.9231|0.0526	0.8801|0.0231	0.8480|0.0135	0.8205|0.0074	0.7838|0.0038
SimpBaseDynMTL	0.9454|0.0673	0.9198|0.0410	0.8850|0.0228	0.8482|0.0149	0.8203|0.0099
RefDynMTL	0.9432|0.0481	0.9307|0.0323	0.9140|0.0233	0.8923|0.0159	0.8632|0.0090
BaseDenseMTL	0.9157|0.0485	0.9019|0.0383	0.8828|0.0262	0.8569|0.0155	0.8301|0.0077
SimpBaseMTL2	0.9099|0.0455	0.8908|0.0323	0.8417|0.0082	0.7966|0.0039	0.7385|0.0016
SimpBaseDenseMTL2	0.9301|0.0491	0.9103|0.0359	0.8910|0.0270	0.8707|0.0174	0.8443|0.0092
RefMTL2	0.9524|0.0549	0.9432|0.0436	0.9276|0.0280	0.9054|0.0189	0.8726|0.0108
RefMTL	0.9394|0.0402	0.9302|0.0276	0.9099|0.0185	0.8882|0.0119	0.8610|0.0061
SimpBaseDenseMTL	0.8951|0.0294	0.8735|0.0180	0.8512|0.0117	0.8314|0.0080	0.8025|0.0039
SimpBaseMTL	0.9425|0.0706	0.9220|0.0436	0.8802|0.0214	0.8513|0.0121	0.8231|0.0061
BaseMTL	0.9342|0.0566	0.9193|0.0409	0.8833|0.0253	0.8645|0.0197	0.8340|0.0120

Tabelle A.18: TPR | FPR aller Modelle je nach Grenzwert. (Part 2 von 2)

A.4 Verwendete Tools und Bibliotheken

Auszug der für die Erstellung der Datensatze verwendete Drittanbieter-Software:

- dumpbin.exe von MSVC 14.38.33130 (Hostx64/x64)
- Everything 1.4.1.1022 (x64)
- ES (Befehlszeilenschnittstelle von Everything) 1.1.0.26
- Nauz-File-Detector 0.09
- Detect-It-Easy 3.10 (x64)

- PowerShell 7.3.5

Auszug der für die Erstellung der Datensatze verwendete Drittanbieter-Bibliotheken:

- app.keve.ktlsh:ktlsh:1.0.1
- com.dorkbox:PeParser:3.1 (bearbeitet)
- pefile 2023.2.7
- requests 2.28.2
- beautifulsoup4 4.11.1

Auszug der für die Untersuchungen verwendete Drittanbieter-Bibliotheken:

- keras 2.11.0
- tensorflow 2.11.0
- matplotlib 3.7.1
- pandas 2.0.1
- wandb 0.15.4
- numpy 1.23.5
- scikit-learn 1.3.2
- jupytext 1.15.0 bis 1.16

Als Grundlage verwendetes Docker-Image: tensorflow/tensorflow:2.11.0-gpu-jupyter

A.5 Beispiel Heatmaps für alle Modelle

Die folgenden Heatmaps wurden für die Datei mit dem SHA256-Hash von *3497d6b076a8303e68cbe401a34114* erzeugt. Die Teilabbildungen, in denen die Modelle der einzelnen Techniken miteinander verglichen werden, zeigen jeweils die Heatmaps der Modelle, die den geringsten Verlust-Wert aufwiesen.

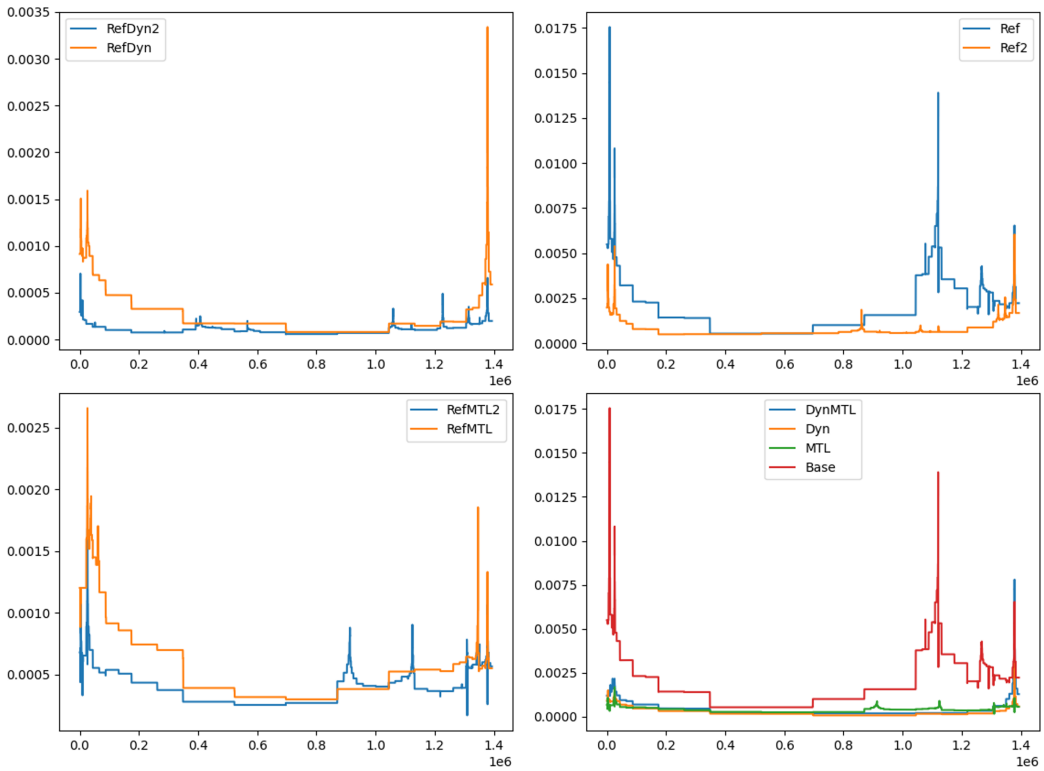

Abbildung A.1: Heatmaps aller Modelle mit der Referenz-Modellgrundlage

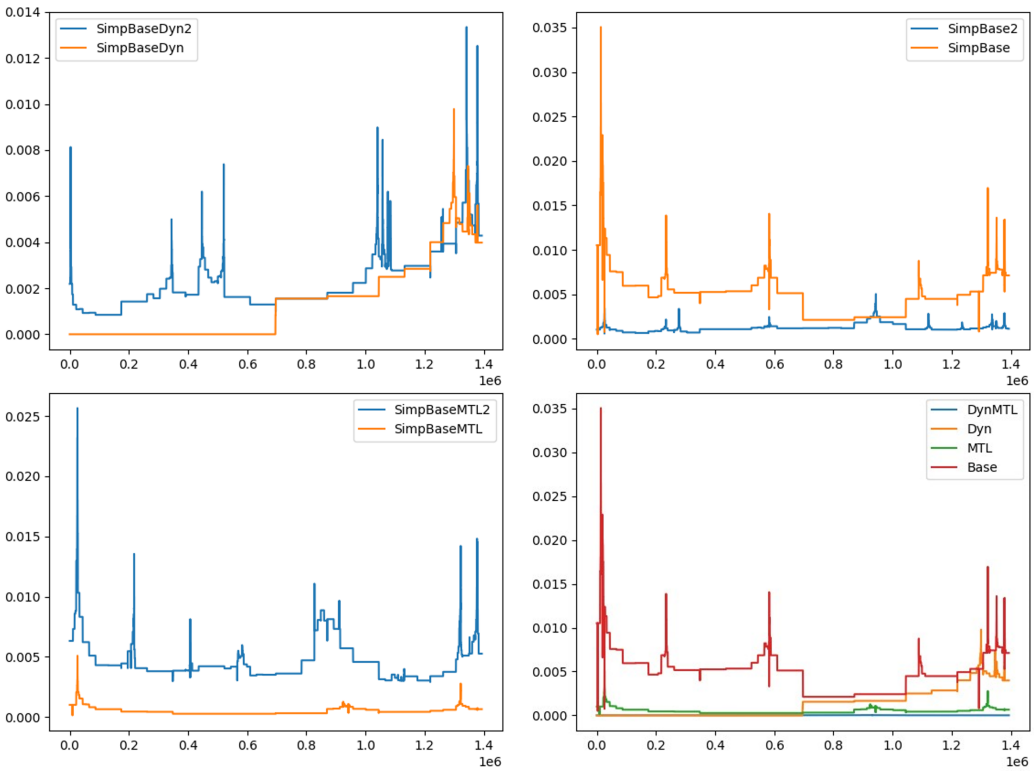

Abbildung A.2: Heatmaps aller Modelle mit der SimpBase-Modellgrundlage

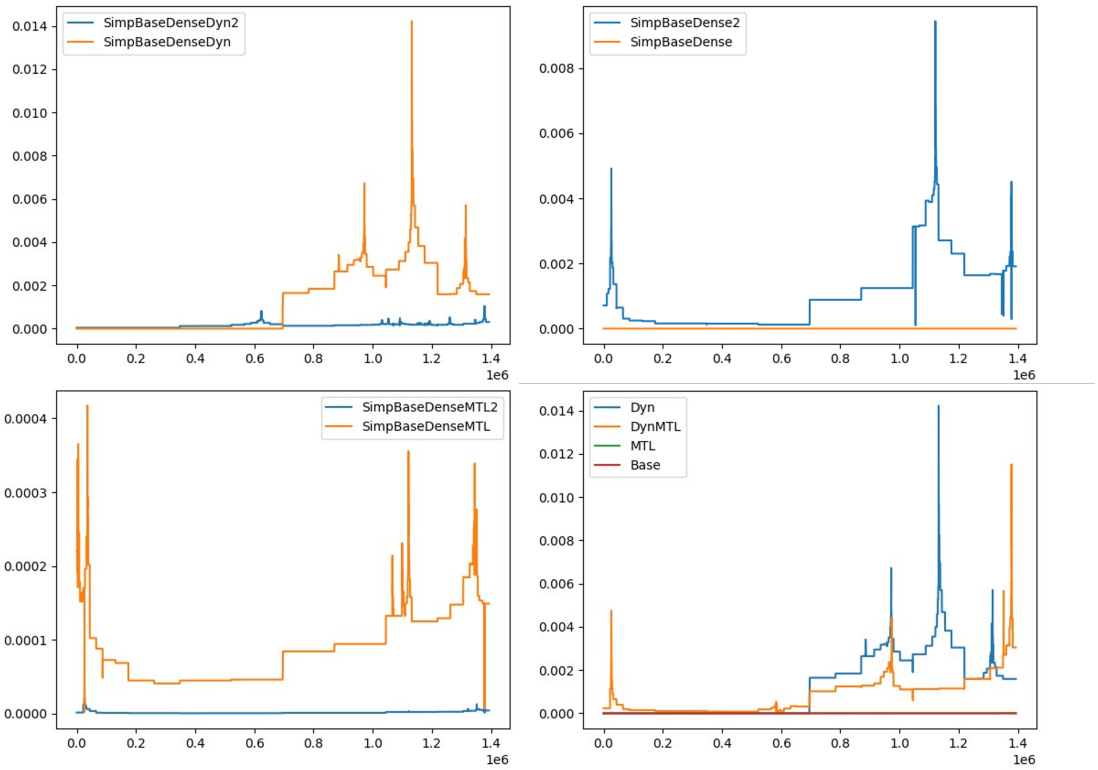

Abbildung A.3: Heatmaps aller Modelle mit der SimpBaseDense-Modellgrundlage

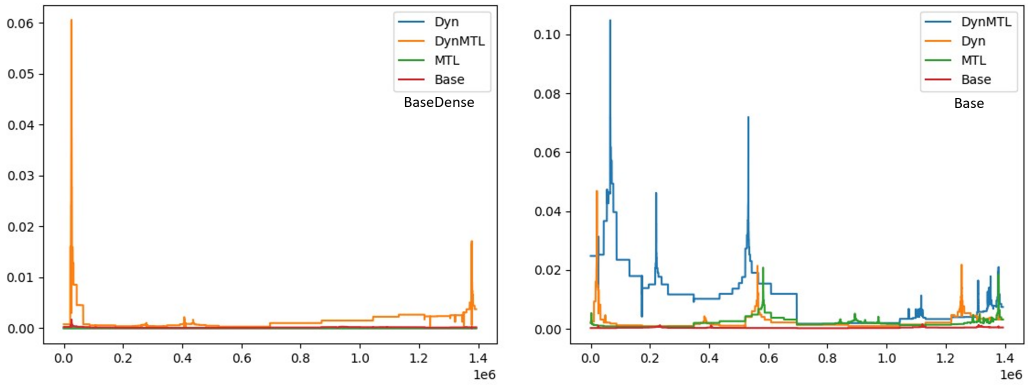

Abbildung A.4: Heatmaps aller Modelle mit der Base und BaseDense-Modellgrundlage

A.6 Als Verhaltenlabel verwendete Signaturen

Checks the version of Bios, possibly for anti-virtualization
Deletes executed files from disk
Looks up the external IP address
Manipulates data from or to the Recycle Bin
Attempts to delete or modify volume shadow copies
Clears Windows events or logs
Attempts to bypass application whitelisting by executing .NET utility in a suspended state, potentially for injection
Attempts to modify proxy settings
NtSetInformationThread: attempt to hide thread from debugger
A system process is generating network traffic likely as a result of process injection
Guard pages use detected - possible anti-debugging.
Uses Windows utilities to create a scheduled task
Executed a process and injected code into it, probably while unpacking
A HTTP/S link was seen in a script or command line
Enumerates services, possibly for anti-virtualization
Fake User-Agent detected
Attempts to restart the guest VM
Application allowlisting bypass and injection detected by executing .NET utility in a suspended state
Queries or connects to DNS-Over-HTTPS/DNS-Over-TLS domain or IP address
Tries to unhook or modify Windows functions monitored by Cuckoo
CAPE detected injection into a browser process, likely for Man-In-Browser (MITB) infostealing
Windows Management Instrumentation (WMI) attempted to create a process
Executed a command line with /V argument which modifies variable behaviour and whitespace allowing for increased obfuscation options
Expresses interest in specific running processes
A process attempted to delay the analysis task.
Connects to a Dynamic DNS Domain
Modifies boot configuration settings
Uses Windows utilities for basic functionality
Checks the CPU name from registry, possibly for anti-virtualization
HTTP traffic contains suspicious features which may be indicative of malware related

traffic
Behavioural detection: Injection with CreateRemoteThread in a remote process
Reads data out of its own binary image
Checks the system manufacturer, likely for anti-virtualization
Attempts to access Bitcoin/ALTCoin wallets
Behavioural detection: Injection (inter-process)
Executable is attempted to be downloaded from an IP
Harvests credentials from local FTP client softwares
Harvests cookies for information gathering
A ping command was executed with the -n argument possibly to delay analysis
Likely virus infection of existing system binary
A process sent information about the computer to a remote location.
Creates a hidden or system file
Multiple direct IP connections
Resolves a suspicious Top Level Domain (TLD)
Created a service that was not started
An executable file was downloaded
Anomalous file deletion behavior detected (10+)
Attempts to remove evidence of file being downloaded from the Internet
Attempts to modify or disable Security Center warnings
Attempts to modify Windows Defender using PowerShell
Enumerates the modules from a process (may be used to locate base addresses in process injection)
Executed a very long command line or script command which may be indicative of chained commands or obfuscation
Network activity contains more than one unique useragent.
Steals private information from local Internet browsers
Creates or sets a registry key to a long series of bytes, possibly to store a binary or malware config
Explorer.exe process established HTTP connections
Checks for the presence of known windows from debuggers and forensic tools
Data downloaded by powershell script
A scripting utility was executed
Installs itself for autorun at Windows startup
Created a process from a suspicious location
Appears to use command line obfuscation

Drops a binary and executes it
Terminates another process
Spoofs its process name and/or associated pathname to appear as a legitimate process
Deletes its original binary from disk
Uses suspicious command line tools or Windows utilities
Queries information on disks, possibly for anti-virtualization
Dynamic (imported) function loading detected
Harvests information related to installed instant messenger clients
Attempts to identify installed AV products by installation directory
Stores JavaScript or a script command in the registry, likely for fileless persistence
Creates RWX memory
Access the NetLogon registry key, potentially used for discovery or tampering
Repeatedly searches for a not-found process, may want to run with startbrowser=1 option
Behavioural detection: Injection (Process Hollowing)
Checks the presence of disk drives in the registry, possibly for anti-virtualization
Performs HTTP requests potentially not found in PCAP.
Uses Windows utilities to enumerate running processes
Attempts to disable Windows Auto Updates
A process attempted to delay the analysis task by a long amount of time.
Collects information to fingerprint the system
Performs some HTTP requests
A script or command line contains a long continuous string indicative of obfuscation
Attempts to create or modify system certificates
Forces a created process to be the child of an unrelated process
Sniffs keystrokes
Harvests information related to installed mail clients
Collects information on the system (ipconfig, netstat, systeminfo)
Attempts to stop active services
Attempts to disable Windows Defender
CAPE detected the Grum, Tofsee malware
Enumerates running processes
Uses IOCTL_SCSI_PASS_THROUGH control codes to manipulate drive/MBR which may be indicative of a bootkit
At least one IP Address, Domain, or File Name was found in a crypto call
Attempts to execute suspicious powershell command arguments

Attempts to interact with an Alternate Data Stream (ADS)
Detects the presence of Wine emulator via function name
HTTPS urls from behavior.
Mimics the system's user agent string for its own requests
Makes a suspicious HTTP request to a commonly exploitable directory with questionable file ext
Collects information about installed applications
Network anomalies occured during the analysis.
Accesses or creates Warzone RAT directories and/or files
Network activity detected but not expressed in API logs
A process created a hidden window
Operates on local firewall's policies and settings
Attempts to disable UAC
Checks adapter addresses which can be used to detect virtual network interfaces
Collects and encrypts information about the computer likely to send to C2 server
Uses Windows APIs to generate a cryptographic key
Connects to crypto currency mining pool
Possible date expiration check, exits too soon after checking local time
Access registry keys
Detects Virtualisation
Detects Antivirus through the presence of a library
Attempts to connect to a dead IP
Code injection in a remote process
Starts servers listening
Attempts to modify Explorer settings
Creates mutexes
Creates a ransomware decryption instruction / key file.
Powershell is sending data to a remote host
Establishes an encrypted HTTPS connection
Exhibits possible ransomware or wiper file modification behavior: overwrites_existing_-files
Exhibits possible ransomware or wiper file modification behavior: mass file deletion

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original