Cyber Security

Malware-Analyse und die Integration von Künstlicher Intelligenz

vorgelegt von

Claudio Stanullo, Stefan Prasser (Matrikelnummer: 3003650, Kontaktstudent)

Studiengang IT-Sicherheitsmanagement
Semester 1

Hochschule Aalen

Hochschule für Technik und Wirtschaft

Betreut durch Prof. Roland Hellmann

15.12.2023

Erklärung

Ich versichere, dass ich die Ausarbeitung mit dem Thema „Malware-Analyse und die Integration von Künstlicher Intelligenz“ selbstständig verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt habe. Die Stellen, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind in jedem einzelnen Fall unter Angabe der Quelle als Entlehnung (Zitat) kenntlich gemacht. Das Gleiche gilt für beigefügte Skizzen und Darstellungen.

Aalen, den 14. Dezember 2023	Claudio Stanullo, Stefan Prasser
Ort, Datum	Autor

Kurzfassung

Die rasante Entwicklung der Cybersicherheitstechnologien, insbesondere durch die Anwendung von Künstlicher Intelligenz (KI) und Machine Learning (ML), hat eine revolutionäre Ära in der Malware-Erkennung eingeläutet. Diese Studie konzentriert sich auf die Anwendung von ML-Algorithmen zur effektiven Identifizierung und Abwehr von Cyberbedrohungen. Mit dem zunehmenden Aufkommen komplexer Malware, die traditionelle, signaturbasierte Erkennungssysteme übersteigt, erweisen sich ML-basierte Methoden als unverzichtbar. Sie bieten die Möglichkeit, aus umfangreichen Datensätzen zu lernen und dynamisch auf neue Malware-Varianten zu reagieren. Die Arbeit beginnt mit einer detaillierten Untersuchung der traditionellen Malware-Erkennungsmethoden, darunter Signaturerkennung, heuristische Analyse, Reverse Engineering und Verhaltensanalyse. Diese Methoden, obwohl effektiv gegen bekannte Bedrohungen, stoßen bei der Erkennung von neuartigen oder modifizierten Malware-Typen an ihre Grenzen. In der Folge liegt der Schwerpunkt auf ML-basierten Techniken. Verschiedene Ansätze wie Supervised und Unsupervised Learning, sowie spezifische Algorithmen wie Random Forest, Gradient Boosting und XGBoost werden untersucht. Die Herausforderungen und Chancen, die sich aus der Anwendung dieser fortschrittlichen Technologien ergeben, werden ebenso beleuchtet wie die Notwendigkeit kontinuierlicher Datenaktualisierung und -aufbereitung zur Erhaltung der Effektivität der Modelle. Abschließend werden die Ergebnisse und Zukunftsperspektiven der KI-basierten Malware-Erkennung vorgestellt. Die Studie unterstreicht das Potenzial dieser Technologien, weist aber auch auf die Notwendigkeit hin, Datenschutzaspekte zu berücksichtigen und die Modelle stetig an sich verändernde Bedrohungsszenarien anzupassen. Die Forschung zeigt, dass ML-Modelle, insbesondere in Kombination mit tiefen Lernverfahren, eine erhebliche Verbesserung in der Malware-Erkennung bewirken können, wobei Herausforderungen wie die Handhabung großer Datenmengen und die Anpassung an neue Malware-Taktiken bestehen bleiben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Abbildungsverzeichnis

Abkürzungssverzeichnis

1. Einleitung

Machine Learning hat sich in den letzten Jahren als eine der bahnbrechendsten Technologien in der Welt der Computertechnik etabliert. Seine Anwendungen erstrecken sich über verschiedene Bereiche, von der Datenanalyse bis hin zur Bilderkennung. Ein besonders faszinierendes und bedeutendes Anwendungsgebiet ist die Malware-Erkennung. In einer Welt, in der die Zahl und Komplexität von Cyberangriffen stetig zunimmt[1] , wird die traditionelle Malware-Erkennung, die auf statischen Signaturen basiert, zunehmend unzureichend, da sie nicht effektiv auf neuartige oder modifizierte Malware-Typen reagieren können [2]. Hier bietet das Machine Learning eine revolutionäre Alternative.[3]

Durch die Implementierung von Machine Learning-Algorithmen in die Malware-Erkennung eröffnen sich neue Möglichkeiten zur Identifizierung und Abwehr von Cyberbedrohungen. Diese Algorithmen sind in der Lage, aus großen Mengen von Daten zu lernen, Muster zu erkennen und Vorhersagen zu treffen, die weit über die Fähigkeiten herkömmlicher Erkennungsmethoden hinausgehen. Sie können sich dynamisch an die sich ständig weiterentwickelnde Landschaft von Malware anpassen, indem sie neue Varianten und Verhaltensweisen identifizieren, die von herkömmlichen, auf Signaturen basierenden Methoden möglicherweise übersehen werden.

Beim Eintauchen in die Welt der Künstlichen Intelligenz (KI) und des Machine Learnings stößt man auf eine Fülle von miteinander verknüpften Begriffen. U.a. ist Künstliche Intelligenz ein zentraler Begriff in diesem Bereich und ein prominenter Forschungszweig innerhalb der Informatik. Aufgrund der fehlenden einheitlichen Definition von "Intelligenz"erstreckt sich dieses Gebiet über das bloße Ausführen vordefinierter Befehlsfolgen hinaus. Die Einbeziehung weiterer Aspekte wie Sprachverarbeitung, visuelle Wahrnehmung, Kreativität, deduktives Denken und allgemeines Lernvermögen variiert je nach der spezifischen Perspektive. Die Lösung vieler dieser Aufgaben basiert auf der Analyse von Daten. Ein separater Bereich innerhalb der KI ist die starke KI, die sich von datenbasierten Methoden abhebt. Diese Methoden fallen unter den Oberbegriff des Machine Learnings, welches darauf basiert, dass externe Informationen und Feedback zur Verbesserung der Aufgabenbewältigung beitragen. Die Hauptfunktionen des Machine Learnings umfassen das Treffen von Vorhersagen (beispielsweise über Eigenschaften oder zukünftige Ereignisse), das Ermitteln und Verstehen von Ursachen und das Identifizieren von Mustern in oft unstrukturierten und umfangreichen Datensätzen.[4] Die Entwicklung statistischer Methoden und mathematischer Modelle zur Bewältigung dieser Herausforderungen ist ebenfalls Teil des Machine Learnings.[5] Die Anwendung dieser Methoden bildet das Herzstück des relativ neuen Feldes der Data Science, welches sich mit den praktischen Aspekten der Datensammlung, -aufbereitung, -analyse und -präsentation befasst.[6] Im Kontext der Datenanalyse wird maschinelles Lernen eingesetzt. [7] und Keshav Thosar et al. [8] haben gezeigt, dass Machine Learning Methoden wie Gradient Boosting, Random Forests und XGBoost signifikante Verbesserungen in der Malware-Erkennung ermöglichen können. Die vorliegende Ausarbeitung zielt darauf ab, die Effektivität verschiedener maschineller Lernmodelle in der Malware-Erkennung zu untersuchen und zu vergleichen. Hierbei wird ein spezieller Fokus auf die Anwendung von Ensemble-Methoden und tiefen Lernalgorithmen gelegt, um die Herausforderungen der modernen Malware-Erkennung

zu adressieren. Durch die Analyse und das Training verschiedener Modelle auf einem umfangreichen Malware-Datensatz wird diese Studie nicht nur zur aktuellen Forschung in der Cybersecurity beitragen, sondern auch praktische Implikationen für die Entwicklung robusterer Malware-Erkennungssysteme aufzeigen.

Bevor sich mit der Nutzung des Machine Learnings für die Detektion von Schadsoftware beschäftigt wird, ist es wichtig, zunächst einige fundamentale Techniken der klassischen Malware-Erkennung und den grundlegenden Methoden des Machine Learnings darzustellen.

2. Methoden der klassischen Maleware-Erkennung

Im Folgenden werden die klassischen Methoden der Malware-Erkennung dargestellt, welche die Signaturerkennung, heuristische Analyse, Reverse Engineering und Verhaltensanalyse umfasst. Jeder dieser Ansätze hat seine eigenen Stärken und Schwächen, die sie für verschiedene Anwendungsfälle geeignet machen.

2.1 Signaturerkennung

Die Signaturerkennung ist die älteste und am weitesten verbreitete Methode der Malware-Erkennung. Sie basiert auf dem Vergleich von verdächtigen Dateien mit einer Datenbank bekannter Malware-Signaturen.[9] D.h. es ist von zentralem Bestandteil, eine umfangreiche und ständig aktualisierte Datenbank zu haben, welche Signaturen von bekannter MalwareVarianten enthält. Diese Signaturen sind in der Regel eindeutige Strings oder Hash-Werte, die aus bekannten Malware-Varianten extrahiert wurden. Ein Algorithmus vergleicht verdächtige Dateien mit den Signaturen in der Datenbank. Bei Übereinstimmung wird die Datei als Malware klassifiziert. Die Effektivität der Signaturerkennung hängt von der regelmäßigen Aktualisierung der Signaturdatenbank ab, um neue Malware-Varianten einzuschließen. Der Hauptvorteil dieser Methode liegt in ihrer Effizienz (die Methode ist schnell und benötigt vergleichsweise wenig Rechenressourcen) und Zuverlässigkeit (bei bekannter Malware bietet die Signaturerkennung eine hohe Erkennungsrate) bei der Erkennung bekannter Malware.[10] Allerdings hat die Signaturerkennung den Nachteil, dass sie nicht in der Lage ist, neue, unbekannte Malware-Varianten zu erkennen, was sie anfällig für Zero-Day-Angriffe macht. Weiter hängt die Wirksamkeit stark von der Häufigkeit und Aktualität der Datenbank-Updates ab. Ein weiteres Problem stellt fortgeschrittene MalwareTechniken wie Polymorphismus und Metamorphismus, welche die Signaturerkennung leicht umgehen können dar.[11]

2.2 Heuristische Analyse

Die heuristische Analyse, im Gegensatz zur Signaturerkennung, basiert nicht auf bekannten Signaturen, sondern auf allgemeinen Verhaltensregeln und -mustern, die auf potenzielle Malware hinweisen könnten. Diese Methodik basiert auf der Analyse von Verhaltensmustern und Eigenschaften von Software, um potentielle Bedrohungen zu identifizieren, die von herkömmlichen, signaturbasierten Erkennungssystemen möglicherweise nicht erfasst werden.[10]

Heuristische Erkennung zielt darauf ab, neue, unbekannte oder veränderte Malware zu identifizieren, die noch keine bekannte Signatur aufweist. Der Prozess verwendet Algorithmen und Modelle, um das Verhalten und die Eigenschaften von Dateien zu analysieren, wobei Anomalien, die auf schädliches Verhalten hindeuten könnten, hervorgehoben werden. Insbesondere wird die Verhaltensanalyse eingesetzt, bei der das Verhalten von Programmen in einer sicheren Umgebung, einer sogenannten Sandbox, beobachtet

wird. Verdächtige Aktionen, wie das Modifizieren von Systemdateien oder das unbefugte Senden von Daten, gelten als potenzielle Indikatoren für Malware.[12]

Die heuristische Analyse bietet zahlreiche Vorteile. Sie ist in der Lage, neue Bedrohungen wie Zero-Day-Exploits und unbekannte Malware zu erkennen, was einen wesentlichen Vorteil gegenüber traditionellen Methoden darstellt. Diese Flexibilität ermöglicht es heuristischen Tools, sich an die ständig wechselnden Taktiken von Cyberkriminellen anzupassen. Zudem verringert sich die Abhängigkeit von Malware-Signaturen, die bei herkömmlichen Methoden oft eine zentrale Rolle spielen.[13]

Allerdings gibt es auch Herausforderungen und Grenzen. Eine davon ist die erhöhte Wahrscheinlichkeit von Falschpositiven, da legale Software manchmal ähnliche Verhaltensmuster wie Malware aufweist. Heuristische Analysen sind zudem ressourcenintensiv und können Systeme verlangsamen. Die Entwicklung effektiver heuristischer Algorithmen ist komplex und erfordert kontinuierliche Anpassungen, um mit den sich ändernden Bedrohungsszenarien Schritt zu halten.[9]

2.3 Reverse Engineering

Reverse Engineering ist eine bewährte Methode in der Cybersicherheit, insbesondere in der Malware-Analyse. Dieser Ansatz beinhaltet das sorgfältige Zerlegen und Untersuchen von Software, um deren Aufbau, Funktionsweise und potenzielle Bedrohungen zu verstehen. In diesem Kontext ist Reverse Engineering ein unverzichtbares Werkzeug zur Identifizierung, Analyse und Bekämpfung von Malware.

Beim Reverse Engineering wird die Malware in einer kontrollierten Umgebung ausgeführt, um ihr Verhalten zu beobachten. Dieser Prozess umfasst oft das Dekompilieren des Codes, um die ursprüngliche Quelle zu verstehen, sowie das Analysieren der Netzwerkkommunikation, Dateioperationen und anderer systemrelevanter Aktivitäten. Ziel ist es, die Absichten und Fähigkeiten der Malware zu entschlüsseln.

Verschiedene Techniken und Werkzeuge werden eingesetzt, um Reverse Engineering effektiv zu betreiben. Dazu gehören Disassembler, wie IDA Pro oder Ghidra, die Maschinencode in eine lesbare Form übersetzen, und Debugger, wie x64dbg, die das Schrittfür-Schritt-Verfolgen der Malware-Ausführung ermöglichen. Sandboxing-Tools isolieren die Malware, um eine sichere Analyse zu gewährleisten.

Reverse Engineering ist mit Herausforderungen verbunden, wie der Erkennung von AntiReverse-Engineering-Techniken, die von Malware-Entwicklern eingesetzt werden. Solche Techniken umfassen das Verschleiern von Code, die Erkennung von Debugging-Umgebungen oder die Selbstmodifikation des Codes. Fortgeschrittene Ansätze wie dynamische Analyse und verhaltensbasierte Erkennungsmethoden werden eingesetzt, um diese Herausforderungen zu bewältigen.[10]

2.4 Verhaltensanalyse

Die Erkennung von Malware durch Verhaltensanalysen stellt einen innovativen und zunehmend wichtigen Ansatz in der Welt der Cybersicherheit dar. Im Unterschied zu herkömmlichen Methoden, die sich auf die Erkennung bekannter Malware-Signaturen verlassen, fokussiert sich die Verhaltensanalyse auf die Beobachtung und Analyse des Verhaltens von Programmen und Prozessen innerhalb eines Systems. Dieser Ansatz ist besonders effektiv bei der Identifizierung von neuartiger oder sich verändernder Malware, die durch traditionelle signaturbasierte Methoden möglicherweise unentdeckt bleiben würde.

Das Kernprinzip der Verhaltensanalyse beruht auf der Erkennung von Anomalien im Systemverhalten. Malware-Aktivitäten zeigen häufig signifikante Abweichungen von normalen Betriebsmustern. Indem Systemaktivitäten wie Dateizugriffe, Netzwerkverkehr und Prozessinteraktionen überwacht werden, können ungewöhnliche Verhaltensmuster, die auf Malware hindeuten, identifiziert werden.

Moderne Tools zur Verhaltensanalyse setzen zunehmend auf Algorithmen des maschinellen Lernens und der künstlichen Intelligenz, um aus umfangreichen Datenmengen zu lernen und normale Verhaltensmuster von potenziell schädlichen zu unterscheiden. Diese Technologien ermöglichen eine kontinuierliche Anpassung und Verbesserung der Erkennungsfähigkeiten.

Neben maschinellem Lernen werden auch heuristische Methoden eingesetzt, die auf vordefinierten Regeln basieren, um verdächtige Aktivitäten zu identifizieren. Diese können beispielsweise Programme umfassen, die ungewöhnliche Änderungen an Systemkonfigurationen vornehmen.

Ein wesentliches Problem bei der Verhaltensanalyse ist das Risiko von Falschpositiven, bei denen legitime Programme fälschlicherweise als Malware eingestuft werden. Dies erfordert eine sorgfältige Kalibrierung und ständige Anpassung der Erkennungsalgorithmen, um die Genauigkeit zu verbessern.

Die ständige Evolution von Malware stellt eine fortwährende Herausforderung dar. Verhaltensanalyse-Tools müssen daher kontinuierlich mit den neuesten Bedrohungsdaten aktualisiert und trainiert werden, um effektiv zu bleiben.

Da die Verhaltensanalyse eine fortlaufende Überwachung und Auswertung erfordert, kann sie ressourcenintensiv sein. Optimierungen in den Algorithmen und eine effiziente Datenverarbeitung sind daher entscheidend, um die Belastung für Systemressourcen zu minimieren.[10]

3. Herausforderung bei der Malware-Erkennung

Die Herausforderungen in der Malware-Erkennung sind ein zentrales und sich stetig entwickelndes Problemfeld in der Welt der Cybersicherheit. Diese Herausforderungen ergeben sich aus der kontinuierlichen Evolution und der steigenden Komplexität von Malware, die in verschiedenen Formen und mit immer raffinierteren Methoden auftritt.

Evolvierende Malware-Techniken sind ein Hauptproblem. Malware-Autoren passen ihre Methoden ständig an, um Erkennungssysteme zu umgehen. Hierbei kommen Techniken wie Polymorphismus und Metamorphismus zum Einsatz, bei denen Malware ihren Code bei jeder Ausführung oder Verbreitung ändert. Diese Dynamik macht die traditionelle, signaturbasierte Erkennung, die auf bekannten Malware-Mustern basiert, zunehmend ineffektiv. Angesichts dieser sich ständig verändernden Bedrohungslage müssen Sicherheitssysteme fortlaufend aktualisiert und verfeinert werden, um mit den sich wandelnden Malware-Techniken Schritt zu halten.[10]

Ein weiteres großes Problem ist die **Zunahme von Zero-Day-Exploits**. Diese bezeichnen Schwachstellen in Software, die dem Hersteller noch nicht bekannt sind und somit noch nicht gepatcht wurden. Da für diese neu entdeckten Schwachstellen noch keine Erkennungssignaturen existieren, sind herkömmliche Antivirenprogramme oft nicht in der Lage, sie zu erkennen. Die schnelle Identifizierung und Schließung dieser Sicherheitslücken ist entscheidend, um potenzielle Schäden durch ausgenutzte Zero-Day-Exploits zu minimieren.[14]

Der **Einsatz von Künstlicher Intelligenz (KI) und maschinellem Lernen** bietet zwar neue Möglichkeiten in der Malware-Erkennung, bringt jedoch auch Herausforderungen mit sich. Diese Technologien können dabei helfen, Muster und Anomalien zu identifizieren, die für Menschen schwer erkennbar sind. Gleichzeitig besteht jedoch die Gefahr, dass Malware-Entwickler dieselben Technologien nutzen, um ihre Angriffe zu verfeinern und Erkennungssysteme zu umgehen. Dies erfordert eine ständige Weiterentwicklung und Anpassung der KI-basierten Erkennungssysteme, um derartige Täuschungsversuche zu durchschauen.[15]

Der **Anstieg von Fileless Malware** stellt eine weitere große Herausforderung dar. Diese Art von Malware nutzt vorhandene legitime Software und Skripte auf einem Computer, um sich zu verbreiten. Sie hinterlässt keine herkömmlichen Malware-Dateien auf der Festplatte, was ihre Entdeckung erheblich erschwert. Fileless Malware erfordert fortschrittliche Erkennungsmethoden, die auf verdächtiges Verhalten und Anomalien in legitimen Prozessen und Anwendungen achten.[16]

Weiter haben im Laufe der Zeit Malware-Entwickler zahlreiche Methoden entwickelt, um der Entdeckung zu entgehen. Zu den häufigsten und effektivsten Methoden zählen[4]:

- Code-Obfuscation: Die Programmlogik und -struktur der Malware-Datei kann durch Maßnahmen wie das Einfügen leerer Berechnungen, das Aufteilen von Funktionsblöcken in kleine Segmente oder eine komplizierte Umorganisation der Befehlsabfolgen

unleserlich gemacht werden. Bei der maschinellen Verarbeitung werden solche bekannten Verschleierungsmethoden normalerweise im Rahmen der Vorverarbeitung normalisiert.[10]

- Anti-Analyse: Beim Reverse Engineering von Malware, bei dem der Schadcode rekonstruiert und als Control Flow Graph dargestellt wird, werden notwendige Werkzeuge wie Disassembler, Debugger und Emulatoren oft durch die Malware selbst gestört.[10]
- Laufzeitpacker: Programme, die den eigentlichen Programmcode komprimieren oder schützen (ähnlich wie bei ZIP-Archiven), sind oft in Malware integriert. Viele tausend unterschiedliche Laufzeitpacker existieren, einige davon werden ausschließlich in Malware verwendet, während die meisten auch in legaler Software zum Einsatz kommen. Malware verwendet nicht selten mehrere Laufzeitpacker hintereinander, wodurch in der äußersten Schicht der Datei nur der Code des Packers sichtbar ist, nicht der eigentliche ausgeführte Code.[10]

ML-basierte Erkennungssysteme müssen diese Hürden überwinden, indem sie eine Vielzahl von Merkmalen und Inhalten ausführbarer Dateien berücksichtigen. Dazu gehören beispielsweise Name, Anzahl und Größe der Sektionen, Namen der importierten Funktionen aus DLLs, Packertyp, Entropie der Datei sowie Dateigröße, Anzahl der lesbaren, schreibbaren und ausführbaren Pages, Daten aus der Resource Section (wie Programm-Icons, Schriftarten), Artefakte von Compiler und Linker, enthaltene Strings mit Pfaden oder URLs, vorhandene Herstellersignaturen und Datumsangaben. Mit ausreichend großen Datenmengen können Maschinen darauf trainiert werden, zwischen harmloser und schädlicher Software zu unterscheiden.[4]

haben, da sie oft unbemerkt bleiben, bis erheblicher Schaden entstanden ist. Traditionelle Malware-Erkennungsmethoden sind oft ineffektiv gegen Zero-Day-Angriffe, da sie auf bekannten Signaturen und Verhaltensweisen basieren, die bei diesen neuen Bedrohungen fehlen.[17]

Schließlich führt die **wachsende Komplexität von Cyberangriffen** dazu, dass die Erkennung und Abwehr von Malware immer schwieriger wird. Moderne Cyberangriffe können mehrere Phasen und Techniken umfassen, von der initialen Infiltration bis hin zur Verbreitung innerhalb eines Netzwerks. Dies erfordert eine kontinuierliche Weiterentwicklung und Anpassung von Erkennungstechnologien, um mit den sich wandelnden Angriffsstrategien mithalten zu können.

Insgesamt ist die Malware-Erkennung ein dynamisches Feld, das fortwährende Aufmerksamkeit, Innovation und Anpassung erfordert. Die Entwicklung effektiverer Erkennungstechnologien, die ständige Aktualisierung von Sicherheitsmaßnahmen und eine proaktive Sicherheitsstrategie sind entscheidend, um sich gegen die ständig weiterentwickelnden Bedrohungen durch Malware zu schützen.

4. Einführung in die KI-basierte Malware-Erkennung

In Antwort auf diese Herausforderungen haben Forscher KI-basierte Ansätze zur Malware-Erkennung entwickelt. Diese nutzen maschinelles Lernen und künstliche Intelligenz, um Muster in Daten zu erkennen, die auf Malware hinweisen könnten. Im Gegensatz zu traditionellen Methoden, die auf festgelegten Regeln basieren, können KI-basierte Systeme lernen und sich anpassen, um auch unbekannte und sich entwickelnde Bedrohungen zu identifizieren. Ein signifikanter Vorteil von KI-basierten Systemen ist ihre Fähigkeit, aus vorhandenen Daten zu lernen und Muster zu identifizieren, die Menschen möglicherweise übersehen. Sie können große Mengen an Daten effizient verarbeiten und bieten somit einen bedeutenden Vorteil bei der Erkennung von Malware in großen Netzwerken. Zudem sind KIbasierte Ansätze effektiver bei der Erkennung von Zero-Day-Angriffen und fortschrittlichen persistenten Bedrohungen, da sie nicht ausschließlich von bekannten Signaturen abhängig sind. Allerdings sind auch KI-basierte Methoden nicht ohne Herausforderungen. Sie benötigen große Mengen an Trainingsdaten, und die Qualität dieser Daten ist entscheidend für die Effektivität des Systems. Darüber hinaus können sie anfällig für sogenannte Adversarial Attacks sein, bei denen Angreifer die KI-Modelle durch manipulierte Eingabedaten täuschen.[17]

4.1 Maschinelles Lernen für die Malware-Erkennung

Für das maschinelle Lernen sind Daten, die in ihrer Quantität und Qualität geeignet sind, um für spezifische Fragestellungen adäquate Lösungen zu generieren, unerlässlich. Die Effektivität und Genauigkeit der resultierenden Lösungen hängen direkt von der Qualität und dem Umfang dieser Daten ab. Inwieweit Maschinen diese Daten effizient nutzen können, ist abhängig davon, ob diese Daten im Vorfeld manuell hinsichtlich ihrer Relevanz und Richtigkeit für die zu lösende Aufgabe überprüft und bewertet wurden.[4]

Im Folgenden wird ein grober Überblick über maschinelles Lernen und wie dies speziell für Malware-Erkennung genutzt werden kann gegeben. Zuerst wird sich kurz der Datenbeschaffung gewidmet. Danach werden die Kategorien des Supervised und des Unsupervised Learnings angeschaut, bevor letztlich noch verschiedene Modelle / Algorithmen, welche im praktischen Teil der Arbeit für das maschinelle Lernen herangezogen werden, präsentiert werden.

4.2 Datenbeschaffung und Datenverarbeitung beim Machine Learning

Es gibt oft die Fehlannahme, dass automatisierte maschinelle Methoden ohne großen Aufwand zuverlässige Ergebnisse liefern können. Diese Annahme ist jedoch irreführend. Eine entscheidende Voraussetzung für den erfolgreichen Einsatz des Machine Learnings ist das Vorhandensein von geeigneten Daten in hinreichender Menge. Die Güte und Aussagekraft der erzielten Ergebnisse sind direkt abhängig von der Qualität dieser Daten.[17] Die Sammlung und die Aufbereitung von nutzbaren Daten in eine Form, die

von Maschinen verarbeitet werden kann, ist ein aufwändiger und mühevoller Prozess. Auch wenn die Daten mit größter Sorgfalt aufbereitet werden, sind die resultierenden Ergebnisse häufig nicht fehlerfrei.[4]

Der erste Schritt in der Anwendung von maschinellem Lernen für die Malware-Erkennung ist die Beschaffung und Vorbereitung geeigneter Daten. Ein gutes Beispiel hierfür ist die Nutzung eines Malware-Datensatzes, wie in den Studien von Thosar et al. [8] und Hariharan et al. [7] erwähnt. In der Regel umfassen diese Datensätze sowohl legitime als auch bösartige Softwarebeispiele:

- **Öffentliche Datensätze**: Quellen wie der EMBER-Datensatz (Endgame Malware BEnchmark for Research)[21] oder der Datensatz von Chebi et al. [22] bieten eine reichhaltige Sammlung von Merkmalen aus PE-Dateien, die sowohl legitime Software als auch Malware umfassen.
- **Sammlung aus verschiedenen Quellen**: Datensätze können auch durch Kombination verschiedener öffentlicher Quellen oder aus internen Sicherheitsaufzeichnungen von Unternehmen zusammengestellt werden, um eine breitere Vielfalt an Malware-Proben zu erfassen.

4.3 Merkmalsextraktion und -auswahl

Die effektive Erkennung von Malware mittels Machine Learnings beginnt mit der sorgfältigen Auswahl und Extraktion von Merkmalen (Features) aus Malware-Daten. Ein konkretes Beispiel hierfür findet sich in der Studie von Saima Naz und Dushyant Kumar Singh [2], *Review of Machine Learning Methods for Windows Malware Detection*. In dieser Arbeit werden verschiedene Merkmale von Windows-Executable-Dateien (PE-Dateien) hervorgehoben, die zur Unterscheidung zwischen legitimer Software und Malware verwendet werden können.

Bei der Merkmalsextraktion werden verschiedene statische Aspekte einer PE-Datei analysiert. Dazu gehören:

- **Strukturmerkmale**: Diese umfassen grundlegende Informationen über die PE-Datei, wie die Größe der Datei, die Anzahl der Sektionen, die Größe des Headers und andere strukturelle Eigenschaften.
- **Byte-Histogramm**: Ein Histogramm, das die Häufigkeit des Auftretens jedes Byte-Wertes in der Datei anzeigt. Dieses Merkmal kann Aufschluss über die allgemeine Zusammensetzung der Datei geben und auffällige Muster aufzeigen, die auf Malware hindeuten könnten.
- **Importierte Funktionen**: Eine Liste der von der Datei importierten Systemfunktionen, die Einblicke in das Verhalten der Software geben kann. Bestimmte

Funktionen, wie solche, die Netzwerkzugriffe oder Dateioperationen ermöglichen, können bei Malware häufiger vorkommen.

- **Abschnittsmerkmale**: Dazu gehören die Entropie jeder Sektion, ihre Größe und andere Eigenschaften. Hohe Entropiewerte können auf verschlüsselte oder gepackte Bereiche hinweisen, was typisch für Malware ist.

Nach der Extraktion dieser Merkmale erfolgt die Feature-Auswahl, um die für die Klassifizierung relevantesten Merkmale zu identifizieren. Im Kontext der Malware-Erkennung ist es entscheidend, Merkmale zu wählen, die zwischen legitimer Software und Malware effektiv unterscheiden können. Hierbei kann der Einsatz von Machine Learning-Techniken wie dem ExtraTreesClassifier von Bedeutung sein. Dieser Algorithmus bewertet die Wichtigkeit jedes Merkmals für die Klassifizierung und ermöglicht es, eine reduzierte Menge von Merkmalen zu wählen, die die höchste Aussagekraft haben.

Die in der Studie von Naz und Singh [2] erwähnten Merkmale sind ein Beispiel dafür, wie aus einer großen Menge potenzieller Daten eine handhabbare und effektive Auswahl getroffen werden kann, um die Genauigkeit der Malware-Erkennung zu maximieren. Diese systematische Herangehensweise an die Merkmalsextraktion und -auswahl ist entscheidend für die Entwicklung leistungsstarker und effizienter Malware-Erkennungssysteme unter Verwendung von Machine Learning-Methoden.

Die beschriebene Vorgehensweise der Datenbeschaffung und -vorbereitung ist essentiell, um die zugrundeliegenden Muster und Anomalien in Malware-Daten effektiv zu erkennen und zu lernen. Die daraus resultierenden Modelle können dann in der Praxis angewendet werden, um neue und unbekannte Malware-Formen zuverlässig zu identifizieren. [8] [7]

4.4 Überwachtes Lernen

Supervised Learning ist ein zentraler Ansatz im Bereich des Machine Learnings, bei dem ein Algorithmus aus einer gegebenen Menge von Trainingsdaten lernt. Diese Daten bestehen typischerweise aus einer Reihe von Eingabebeispielen und den zugehörigen Ausgaben. Ziel ist es, aus diesen Trainingsdaten ein Modell zu entwickeln, das Vorhersagen oder Entscheidungen ohne menschliche Intervention treffen kann.[17]

Bei Supervised Learning besteht der erste Schritt darin, eine umfangreiche Menge an Trainingsdaten zu sammeln, die sowohl die Eingabemerkmale (Features) als auch die zugehörigen Zielwerte (Labels) enthalten. Anschließend wird ein Modell entwickelt, das aus den Eingabedaten lernt. Dies kann ein einfaches lineares Modell, ein komplexes neuronales Netzwerk oder irgendein anderer Algorithmus sein. Während des Trainings passt das Modell seine Parameter an, um die Abweichung zwischen seinen Vorhersagen und den tatsächlichen Ausgabewerten zu minimieren. Dieser Prozess wird oft durch eine Verlustfunktion (Loss Function) gesteuert.[18]

Das Modell wird regelmäßig anhand eines Validierungsdatensatzes bewertet, um Overfitting (ein Zustand, in dem das Modell zu gut an die Trainingsdaten angepasst ist und schlecht auf neue, unbekannte Daten reagiert) zu vermeiden. Es gibt verschiedene Typen von Supervised Learning, wie Klassifizierung, bei der Eingabedaten in Kategorien eingeteilt werden, und Regression, bei der eine kontinuierliche Ausgabe prognostiziert wird.[19] Herausforderungen im Supervised Learning umfassen die Sicherstellung der Datenqualität und -quantität, das Feature-Engineering und die Generalisierungsfähigkeit des Modells. Die Auswahl und Vorbereitung der richtigen Merkmale ist entscheidend für die Modellleistung. Ein guter Supervised Learning Algorithmus sollte nicht nur auf den Trainingsdaten, sondern auch auf neuen, unbekannten Daten gut funktionieren.

4.5 Unüberwachtes Lernen

Unsupervised Learning ist ein Bereich des Machine Learnings, der darauf abzielt, Muster und Strukturen in Daten zu entdecken, ohne dass dabei auf vorgegebene Labels oder Zielwerte zurückgegriffen wird. Im Gegensatz zum Supervised Learning, bei dem Modelle anhand bekannter Eingabe-Ausgabe-Paare trainiert werden, verwendet Unsupervised Learning Trainingsdaten ohne explizite Anweisungen. Das Ziel ist es, die zugrundeliegende Struktur der Daten zu erkennen und nützliche Informationen daraus abzuleiten.

Bei Unsupervised Learning werden Daten explorativ analysiert, um verborgene Muster oder Anomalien zu entdecken. Ein wichtiger Anwendungsbereich ist die Reduzierung der Dimensionalität, bei der Techniken wie die Principal Component Analysis (PCA) eingesetzt werden, um komplexe Daten auf ihre wesentlichen Merkmale zu reduzieren. Da es keine vordefinierten Labels gibt, muss das Modell die Struktur der Daten aus den Eigenschaften der Daten selbst verstehen.[19]

Zu den Methoden des Unsupervised Learning gehören Clustering, Assoziationsanalyse und Autoencoder. Beim Clustering werden Datenpunkte in Gruppen eingeteilt, sodass die Punkte innerhalb eines Clusters einander ähnlicher sind als die in anderen Clustern. Die Assoziationsanalyse wird verwendet, um Assoziationsregeln in großen Datenmengen zu identifizieren, und Autoencoder lernen effiziente Codierungen der Eingabedaten.[19]

Die Herausforderungen bei Unsupervised Learning umfassen die Interpretation der Ergebnisse, die Auswahl der richtigen Methoden und die Datenqualität und -vorbereitung. Die Interpretation kann ohne klare Zielwerte herausfordernd sein und erfordert oft eine sorgfältige Analyse und Verständnis des Kontextes der Daten. Die Wahl des richtigen Algorithmus oder der passenden Parameter ist entscheidend, da unterschiedliche Methoden zu verschiedenen Ergebnissen führen können. Wie bei allen maschinellen Lernmethoden ist auch hier die Qualität der Daten von großer Bedeutung.[20]

Unsupervised Learning spielt eine wichtige Rolle in der Datenexploration, Mustererkennung und in der Vorverarbeitung von Daten für andere maschinelle Lernverfahren. Es

bietet leistungsstarke Werkzeuge zur Analyse und zum Verständnis großer und komplexer Datensätze, was in der Ära von Big Data besonders wichtig ist.

4.6 Algorithmen für das maschinelle Lernen zur Malware-Erkennung

Der praktische Teil der Arbeit bedient sich des sogenannten Ensemble Learnings, welcher dem Supervised Learning zugeordnet werden kann. Das Ensemble Learning unterscheidet zwei Ansätze, zum einen dem Boosting und zum anderen dem Bagging.

4.6.1 Random Forest (Bagging)

Random Forest ist eine Methode des Machine Learnings, die zur Lösung von Klassifikations- und Regressionsproblemen eingesetzt wird. Sie gehört zur Kategorie des Ensemble-Lernens, was bedeutet, dass sie auf der Kombination mehrerer Modelle – in diesem Fall Entscheidungsbäume – basiert, um robustere und genauere Vorhersagen zu ermöglichen.

Die Kernidee hinter Random Forest ist relativ einfach: Anstatt sich auf einen einzelnen Entscheidungsbaum zu verlassen, erstellt Random Forest einen "Wald"aus vielen Bäumen. Jeder Baum im Wald wird aus einer zufälligen Stichprobe der Trainingsdaten generiert, wobei die Methode des Bootstrapping (oder Bagging) verwendet wird. Dies bedeutet, dass für jeden Baum eine zufällige Auswahl an Datenpunkten mit Zurücklegen gezogen wird.

Ein weiteres Schlüsselelement ist die Feature-Randomisierung. Während des Aufbaus eines jeden Baumes wird nur eine zufällige Teilmenge der Merkmale (Features) zur Entscheidungsfindung herangezogen. Diese Strategie trägt dazu bei, die Korrelation zwischen den Bäumen im Wald zu verringern und verbessert die Generalisierbarkeit des Modells, indem es die Tendenz zur Overfitting (Overfitting) reduziert.

Bei der Vorhersage kombiniert Random Forest die Ergebnisse seiner vielen Bäume. In einem Klassifikationskontext stimmt jeder Baum für eine Klasse ab, und die Klasse mit den meisten Stimmen wird als Vorhersage des Modells ausgewählt. Bei Regressionsproblemen wird der Durchschnitt der Vorhersagen aller Bäume genommen.[19]

Random Forest bietet mehrere Vorteile. Er ist in der Regel robuster und genauer als einzelne Entscheidungsbäume und kann automatisch wichtige Merkmale identifizieren, was bei der Interpretation der Daten hilfreich ist. Allerdings ist diese Methode auch rechenintensiver und erzeugt Modelle, die aufgrund ihrer Größe und Komplexität schwieriger zu interpretieren sind.

Random Forest ist folglich ein effektiver Ansatz für die Klassifizierung von Malware. Garcia et al. [23] zeigten die Wirksamkeit dieser Methode, indem sie Malware-Binärdateien

in Bilder umwandelten und diese mit einem Random-Forest-Algorithmus klassifizierten. Die hohe Genauigkeit, die dabei erreicht wurde, unterstreicht die Stärke dieses Ansatzes, insbesondere im Umgang mit Code-Obfuskationstechniken, die von Malware-Autoren genutzt werden.

4.6.2 Gradient Boosting

Gradient Boosting ist eine fortgeschrittene Technik im maschinellen Lernen, die häufig für Klassifikations- und Regressionsaufgaben eingesetzt wird. Diese Methode gehört zur Familie des Ensemble-Lernens und baut auf der Idee auf, mehrere einfache Modelle (typischerweise Entscheidungsbäume) sequenziell zu kombinieren, um ein starkes Gesamtmodell zu erzeugen.

Im Gegensatz zu anderen Ensemble-Methoden wie Random Forest, wo die einzelnen Modelle parallel und unabhängig voneinander erstellt werden, folgt Gradient Boosting einem sequenziellen Ansatz. Es beginnt mit einem einfachen Anfangsmodell und fügt dann iterativ neue Modelle hinzu, wobei jeder neue Baum speziell darauf ausgerichtet ist, die Fehler des aktuellen Gesamtmodells zu korrigieren. Diese Fehlerkorrektur erfolgt durch das Training der neuen Bäume auf den Residuen, also den Unterschieden zwischen den tatsächlichen und vorhergesagten Werten der Daten.

Ein Schlüsselelement des Gradient Boosting ist die Anwendung von Techniken des Gradientenabstiegs. Diese Optimierungsmethode wird verwendet, um den Gesamtverlust des Modells, eine Maßzahl für seine Ungenauigkeit, zu minimieren. Der Prozess wird fortgesetzt, indem weitere Bäume hinzugefügt werden, bis entweder eine vorgegebene Anzahl von Bäumen erreicht ist oder keine signifikante Verbesserung der Vorhersagegenauigkeit mehr erzielt wird.

Ein großer Vorteil von Gradient Boosting ist seine hohe Vorhersagegenauigkeit. Das Verfahren ist in der Lage, komplexe nichtlineare Beziehungen in den Daten zu erfassen. Es ist jedoch auch wichtig zu beachten, dass Gradient Boosting anfällig für Overfitting sein kann, insbesondere wenn es nicht richtig reguliert wird. Außerdem erfordert das Verfahren aufgrund seines sequenziellen Aufbaus mehr Rechenleistung als einige andere Methoden. Zudem ist die Auswahl der richtigen Parameter, wie die Lernrate und die Anzahl der Bäume, entscheidend für die Leistung des Modells und kann eine Herausforderung darstellen.

Zusammengefasst ist Gradient Boosting eine leistungsstarke Methode im maschinellen Lernen, die für ihre Genauigkeit und Flexibilität bekannt ist. Sie erfordert jedoch eine sorgfältige Abstimmung der Parameter und eine kontinuierliche Überwachung, um optimale Ergebnisse zu erzielen und Overfitting zu vermeiden.[19]

4.6.3 XGBoost

XGBoost, kurz für Extreme Gradient Boosting, ist eine hochentwickelte Implementierung der Gradient-Boosting-Methode, die in der Welt des Machine Learnings für ihre hohe Effizienz, Leistungsfähigkeit und Flexibilität geschätzt wird. Diese Methode hat sich insbesondere in Datenwissenschafts-Wettbewerben und komplexen Anwendungen aufgrund ihrer überlegenen Vorhersagegenauigkeit und Verarbeitungsgeschwindigkeit einen Namen gemacht.

Im Kern nutzt XGBoost die Prinzipien des Gradient Boosting, wobei mehrere schwache Vorhersagemodelle – meist Entscheidungsbäume – sequenziell kombiniert werden, um ein starkes Gesamtmodell zu erstellen. Jedes neue Modell in der Sequenz zielt darauf ab, die Fehler des vorherigen Modells zu korrigieren. Einzigartig bei XGBoost ist die Einführung zusätzlicher Regularisierungsterme (L1 und L2) in die Kostenfunktion, welche die Gefahr der Overfitting (Overfitting) verringern und die Gesamtleistung des Modells verbessern.

Die Baumkonstruktion in XGBoost ist besonders effizient und effektiv. Bei der Erstellung jedes Baumes werden optimale Aufteilungspunkte unter Berücksichtigung von Gradienteninformationen und einer speziellen Bewertungsfunktion, die sowohl den Gewinn aus der Aufteilung als auch die Regularisierung einbezieht, berechnet. Dies führt zu präziseren und effizienteren Modellen. Ein weiterer Vorteil von XGBoost ist seine Fähigkeit, automatisch mit fehlenden Daten umzugehen, was den Prozess der Datenverarbeitung vereinfacht.

XGBoost zeichnet sich durch seine Skalierbarkeit und Effizienz aus. Es ist auf verschiedenen Hardware-Plattformen lauffähig und nutzt die Ressourcen wie Mehrkern-CPUs und GPUs effizient. Der iterative Prozess der Modellerweiterung in XGBoost, bei dem in jedem Schritt ein neuer Baum hinzugefügt wird, der die verbleibenden Fehler des aktuellen Modells minimiert, wird solange fortgesetzt, bis eine vorgegebene Anzahl von Bäumen erreicht ist oder keine signifikante Verbesserung mehr erzielt wird. Dabei wird jeder neue Baum mit einem Faktor gewichtet, der auf der Lernrate basiert, und in das Gesamtmodell integriert.

Einer der größten Vorteile von XGBoost ist seine hohe Vorhersagegenauigkeit. Das Verfahren eignet sich hervorragend zur Modellierung komplexer Datensätze und kann für eine Vielzahl von Problemen, einschließlich Klassifikation, Regression und Rangordnung, verwendet werden. XGBoost unterstützt verschiedene Verlustfunktionen und bietet eine breite Palette an einstellbaren Parametern, einschließlich Baumtiefe, Lernrate und Regularisierung.

Allerdings erfordert XGBoost eine sorgfältige Abstimmung seiner Hyperparameter, was eine Herausforderung darstellen kann. Auch kann die Komplexität und Vielfalt der einstellbaren Parameter für Anfänger im Bereich des Machine Learnings einschüchternd sein.

Zusammenfassend ist XGBoost eine extrem leistungsfähige Methode im maschinellen Lernen, die sich durch ihre Fähigkeit auszeichnet, effizient und wirkungsvoll mit einer Vielzahl von Daten und Problemstellungen umzugehen. Ihre Effizienz, kombiniert mit der Fähigkeit, präzise Modelle zu erstellen, macht sie zu einem bevorzugten Werkzeug in der Welt der Datenwissenschaft.[19]

4.7 Hyperparameter-Optimierung

Hyperparameter-Optimierung hilft bei der Auswahl optimaler Hyperparameter, um die höchstmögliche Leistung des Modells zu erzielen. Im Gegensatz zu Modellparametern, die während des Lernprozesses direkt aus den Daten abgeleitet werden, müssen Hyperparameter vorab festgelegt werden. Diese Voreinstellungen können signifikante Auswirkungen auf die Effektivität und Präzision des Modells haben.

Ein wesentlicher Bestandteil des Hyperparameter-Optimierungen ist der Prozess der Kreuzvalidierung, der häufig mit Techniken wie der GridSearch implementiert wird. Kreuzvalidierung ist eine Methode zur Bewertung der Generalisierbarkeit eines Modells und zur Vermeidung von Overfitting. Sie beinhaltet die Aufteilung der Daten in mehrere Teilmengen, das Trainieren des Modells auf einem Teil dieser Mengen und das Testen auf den verbleibenden. Dieser Vorgang wird wiederholt, um eine umfassende Bewertung der Modellleistung zu ermöglichen. Die GridSearch-Methode durchsucht systematisch verschiedene Kombinationen von Hyperparametern, um diejenige zu identifizieren, die die beste Leistung bietet. Jede Kombination von Hyperparametern wird dabei unter Verwendung der Kreuzvalidierung bewertet. [25]

4.8 Evaluierung von KI-Modellen

Die Evaluierung von KI-Modellen in der Malware-Erkennung ist entscheidend, um ihre Wirksamkeit und Genauigkeit zu bestimmen. Diese Sektion behandelt verschiedene Methoden und Metriken, die in der Literatur für die Leistungsbewertung von KI-Modellen verwendet werden.

4.8.1 Performance-Metriken

Die Leistung von KI-Modellen wird oft anhand von Metriken wie Genauigkeit, Präzision, Sensitivität und F1-Score bewertet. Diese Metriken sind entscheidend für die Beurteilung, wie gut das Modell Malware identifiziert und falsche Alarme minimiert [26]. Die AUC-ROC-Kurve ist eine weitere wichtige Metrik, die das Verhältnis von wahren positiven zu falsch positiven Raten aufzeigt und in [27] diskutiert wird.

4.8.2 Konfusionsmatrix

Eine Konfusionsmatrix ist ein nützliches Werkzeug, um die Leistung eines Klassifikationsmodells zu visualisieren und wird in [28] detailliert beschrieben. Sie zeigt die Anzahl der korrekten und falschen Vorhersagen nach Klassen aufgeteilt, was hilft, die Stärken und Schwächen des Modells zu verstehen.

4.8.3 Cross-Validation

Die k-fache Cross-Validation wird häufig eingesetzt, um die Generalisierbarkeit von KI-Modellen zu überprüfen. Diese Methode ermöglicht es, die Stabilität und Robustheit des Modells in verschiedenen Datensätzen zu testen, wie in [29] erörtert.

4.8.4 Real-World Testing

Abschließend ist es unerlässlich, die Modelle in realen Umgebungen zu testen, um ihre Effektivität in praktischen Anwendungsfällen zu bewerten. Dies beinhaltet das Testen der Modelle mit aktuellen Malware-Proben, um ihre Fähigkeit zur Erkennung neuer Bedrohungen zu beurteilen, wie es in [8] und [7] beschrieben wird.

Eine Kombination dieser Evaluierungsmethoden bildet ein umfassendes Bild der Leistungsfähigkeit und Zuverlässigkeit von KI-Modellen in der Malware-Erkennung.

4.9 Anwendungsfälle der KI-basierten Malware-Erkennung

Die Integration von Künstlicher Intelligenz (KI) in die Malware-Erkennung revolutioniert die Cybersicherheit, indem sie vielfältige Anwendungsmöglichkeiten in verschiedenen Umgebungen bietet. Dieser Abschnitt beleuchtet, wie KI-Technologien zur Erkennung und Prävention von Malware beitragen.

4.9.1 Erweiterte Erkennungsfähigkeiten

Einer der herausragenden Vorteile der KI in der Malware-Erkennung ist ihre Fähigkeit, Zero-Day-Angriffe zu identifizieren, die traditionelle, signaturbasierte Sicherheitssysteme oft übersehen. Diese neuen, unbekannten Bedrohungen erfordern eine flexible und adaptive Herangehensweise, die KI-basierte Modelle bieten. Durch die Analyse von Mustern und Anomalien können diese Modelle Bedrohungen erkennen, ohne sich auf vorher bekannte Signaturen zu verlassen [26].

Neben der Erkennung von Zero-Day-Angriffen spielen KI-Systeme auch eine entscheidende Rolle in der Netzwerksicherheit. Sie analysieren kontinuierlich den Datenverkehr und identifizieren verdächtige Aktivitäten, indem sie aus früheren Angriffen lernen und sich an neue Bedrohungsszenarien anpassen. Diese Fähigkeit macht sie zu einem unverzichtbaren Werkzeug in der Netzwerksicherheit [28].

4.9.2 Einsatz in verschiedenen Umgebungen

KI-basierte Sicherheitslösungen sind auch im Bereich der Endpoint-Sicherheit von Bedeutung. Endpoint-Geräte, wie Computer und Smartphones, sind häufige Ziele für Malware. KI-Modelle, die auf diesen Geräten implementiert werden, können verdächtige Prozesse und Dateien in Echtzeit erkennen, was die Gefahr von Malware-Infektionen signifikant reduziert [29].

Darüber hinaus sind KI-Algorithmen in der Lage, in Cloud-Umgebungen große Mengen an Sicherheitsdaten effizient zu analysieren. Sie unterstützen die Erkennung von Anomalien in Nutzerverhalten und Anwendungsaktivitäten, was besonders wichtig ist, da Cloud-Dienste zunehmend im Fokus von Cyberangriffen stehen [27].

KI-Modelle können nicht nur Malware erkennen, sondern auch automatisierte Reaktionen auf identifizierte Bedrohungen ausführen. Sie treffen Entscheidungen in Echtzeit und ergreifen geeignete Maßnahmen zur Eindämmung von Angriffen, wodurch sie eine umfassende Sicherheitslösung bieten [30].

Diese Anwendungsfälle zeigen, dass KI-Technologien in der Malware-Erkennung ein breites Spektrum an Möglichkeiten bieten, um die Sicherheit in verschiedenen Umgebungen und Anwendungen zu verbessern.

5. Praktische Umsetzung einer Malwareanalyse mit KI

Die Durchführung der praktischen Malwareanalyse mittels KI-Techniken erfolgt unter sorgfältiger Beachtung des Werkes "Mastering Machine Learning for Penetration Testing" von Chebbi [22]. Der hierbei verwendete Datensatz, welcher sowohl legitime als auch Malware-Daten umfasst, entstammt direkt aus dieser Quelle und dient als fundamentale Grundlage für die nachstehenden Analysen und Erkenntnisse.

5.1 Methodik

5.1.1 Datenerfassung und -aufbereitung

Der analytische Prozess unserer Studie beginnt mit dem Import relevanter Bibliotheken und dem Laden des Malware-Datensatzes, welcher aus dem zum Buch Mastering Machine Learning for Penetration Testing"gehörenden GitHub-Repository [31] entnommen wurde. Dieser Datensatz umfasst eine Sammlung von 42.323 legitimen Binärdateien (exe, dll) und 96.724 Malware-Dateien, die vom Security Blogger Prateek Lalwani zusammengestellt wurden. Die Malware-Dateien wurden dabei von der Webseite VirusShare bezogen. In der anschließenden Phase der Datenvorbereitung erfolgt die Aufteilung in legitime Software und Malware. Danach werden spezifische Spalten, die für die Analyse der Eigenschaften oder des Verhaltens der Dateien nicht ausschlaggebend sind (wie Name, MD5-Hash und die direkte Klassifikation in legitime oder Malware-Daten), entfernt, um eine fokussierte und effiziente Datenbasis für die weiterführende Analyse zu schaffen.

5.1.2 Feature-Engineering

In der Umsetzung der Feature Selection spielt der Einsatz des ExtraTreesClassifier eine wesentliche Rolle, wie auch in der Studie "Towards Optimization of Malware Detection" [32] hervorgehoben wird. Dieser Classifier, ein fortschrittliches Ensemble-Lernverfahren, basiert auf zahlreichen Entscheidungsbäumen und eignet sich besonders gut für die Analyse komplexer Malware-Daten. Seine Fähigkeit, die Wichtigkeit einzelner Features effektiv zu bewerten, ermöglicht es, die relevantesten Merkmale zu identifizieren, die für die Unterscheidung zwischen legitimer Software und Malware entscheidend sind. Der ExtraTreesClassifier wird auf verschiedenen Teilmengen des Datensatzes trainiert, um eine präzise Feature-Selektion durchzuführen. Diese präzise Auswahl von Merkmalen reduziert nicht nur die Komplexität des Modells, sondern verbessert auch dessen Vorhersagegenauigkeit. Solche Verbesserungen in der Vorhersagegenauigkeit sind unerlässlich für die Entwicklung robuster und effizienter Malware-Erkennungssysteme. Diese Studie unterstreicht die Bedeutung des ExtraTreesClassifier als Schlüsselkomponente in modernen Malware-Erkennungsverfahren.

5.1.3 Modelltraining und -evaluation

Nun werden der RandomForestClassifier, GradientBoostingClassifier und XGBoost-Classifier anhand des vorbereiteten Datensatzes trainiert und evaluiert. Wichtige Evaluationsmetriken wie Genauigkeit, Präzision, Recall, F1-Score und ROC-AUC-Score werden herangezogen, um die Leistung der Modelle bei der Klassifizierung von Malware zu bewerten [28]. Die Konfusionsmatrix, visualisiert durch Heatmaps, bietet detaillierte Einblicke in die Klassifizierungsergebnisse [33]. Cross-Validation wird angewendet, um die Stabilität und Zuverlässigkeit der Modelle zu überprüfen [27]. Nach der Evaluation werden die Modelle gespeichert, was ihre spätere Verwendung in Sicherheitssystemen ermöglicht.

5.2 Implementierung

Die Implementierung der ML-Modelle zu Malware-Erkennung wird mit der Programmiersprache Python realisiert. Python ist in der Welt des Machine Learnings aufgrund seiner Benutzerfreundlichkeit und der Verfügbarkeit umfangreicher Bibliotheken und Pakete besonders beliebt. Diese Eigenschaften machen Python zu einer effizienten und effektiven Wahl für die Entwicklung von ML-Modellen, wie sie in dieser Studie erforderlich sind.

Für die Umsetzung des Projekts wurden diverse Python-Bibliotheken genutzt, die jeweils spezifische Funktionen erfüllen. *Pandas*, eine leistungsstarke Bibliothek für Datenmanipulation und -analyse, wurde verwendet, um den Malware-Datensatz zu verarbeiten und vorzubereiten. Die Bibliothek *Scikit-learn*, ein vielseitiges Werkzeug für maschinelles Lernen, kam bei der Modellbildung und -evaluation zum Einsatz. Für die Visualisierung der Daten und der Ergebnisse wurden *matplotlib* und *seaborn* verwendet, die umfangreiche Möglichkeiten zur Erstellung aussagekräftiger Grafiken bieten. Zusätzlich wurde xgboost, eine optimierte Implementierung von Gradient Boosting, für die Entwicklung des XGBoost-Modells verwendet.

Der gesamte Code ist digital in Github zugänglich und wird zusätzlich im Anhang in gedruckter Form bereitgestellt.

5.3 Anwendung

Neben der Entwicklung und Optimierung von Klassifizierungsmodellen wurde in dieser Arbeit eine wichtige Funktionalität implementiert, die es ermöglicht, externe Dateien direkt auf die Modelle anzuwenden. Dieser Ansatz dient der Evaluierung der praktischen Anwendbarkeit und Effektivität der Modelle in realen Anwendungsszenarien.

Die Integration der Dateiverarbeitung wurde durch die Entwicklung eines spezialisierten Moduls realisiert. Dieses Modul ist verantwortlich für das Einlesen der Dateien, deren

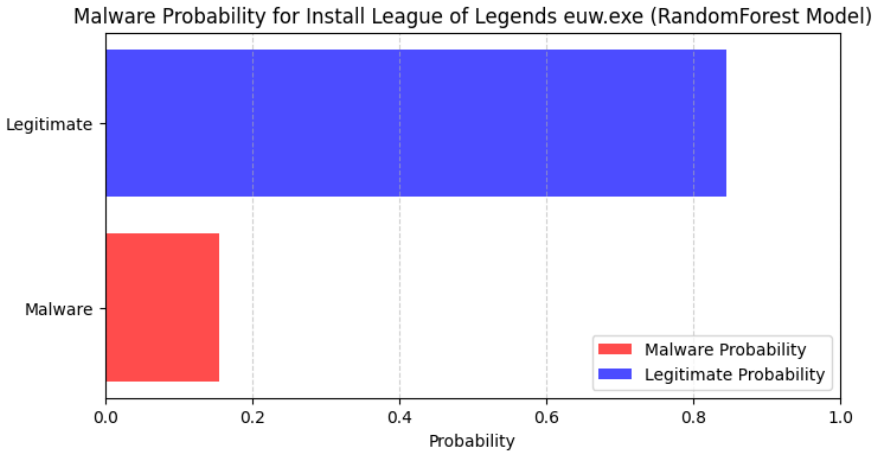


Abbildung 5.1: RandomForestClassifier

Vorverarbeitung und die anschließende Anwendung des Modells auf die vorbereiteten Daten. Die Fähigkeit, neue Daten einzulesen und zu verarbeiten, ist entscheidend, um die Modelle unter realen Bedingungen zu testen und zu validieren.

Der Prozess beginnt mit dem Einlesen der Datei, gefolgt von mehreren Vorverarbeitungsschritten, die auf die Bedürfnisse der jeweiligen Modellarchitektur abgestimmt sind. Nach der Vorverarbeitung werden die Daten durch das Modell geführt, welches eine Klassifizierungsvorhersage trifft. Die resultierenden Vorhersagen werden anschließend für weitere Analysen oder die direkte Anwendung bereitgestellt.

Zur Demonstration der Funktionalität sind die Malwarewahrscheinlichkeiten der Datei ”League of Legends euw.exe” für verschiedene Modelle dargestellt. Abbildung 5.1 zeigt die Wahrscheinlichkeiten für RandomForest, Abbildung 5.2 für GradientBoost und Abbildung 5.3 für XGBoost.

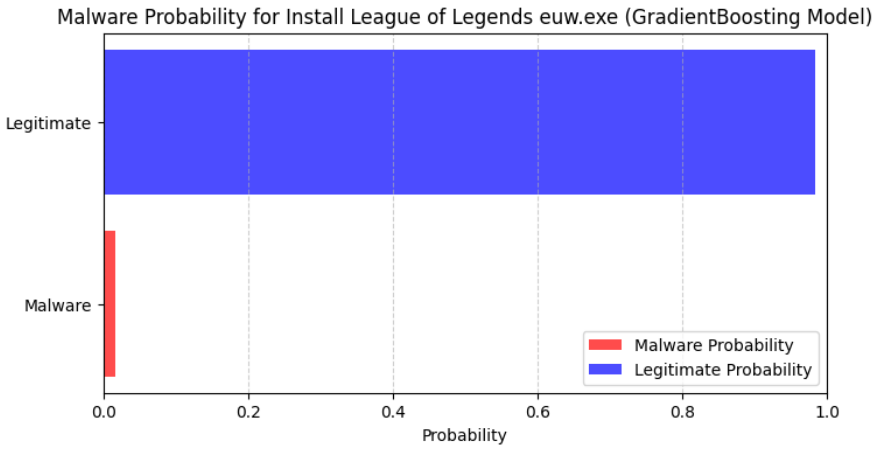


Abbildung 5.2: GradientBoostClassifier

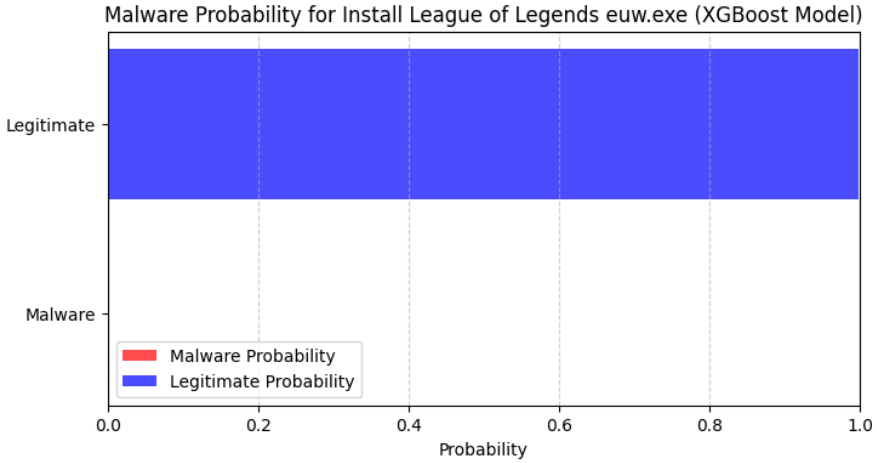


Abbildung 5.3: XGBClassifier

6. Evaluierung

6.1 Feature-Selektion

Im Rahmen der Entwicklung von Machine Learning Modellen zur Malware-Detektion wurde eine umfassende Feature-Selektion mittels des *ExtraTreesClassifier* implementiert. Ziel dieser Selektion war es, die ursprünglich umfangreiche Merkmalsmenge, bestehend aus 53 verschiedenen Attributen, auf eine essentielle Auswahl an Schlüsselmerkmalen zu reduzieren. Diese Reduktion dient der Befreiung des Modells von redundanten sowie weniger informativen Daten. Ein solcher Schritt minimiert die Komplexität der Modellstruktur, verkürzt parallel dazu die Rechenzeiten und steigert die Interpretierbarkeit der Resultate. Die relativen Wichtigkeiten dieser Features werden in Abbildung 6.1 visualisiert.

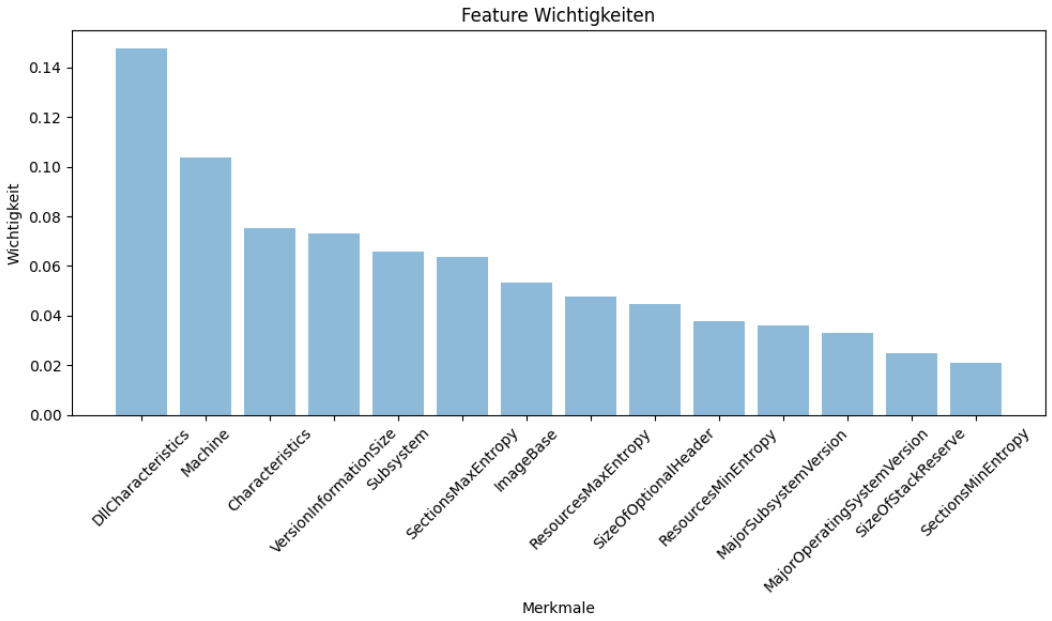


Abbildung 6.1: Visualisierung der relativen Wichtigkeit selektierter Merkmale aus einem Datensatz von ausführbaren Windows-Dateien zur Differenzierung zwischen Malware und legitimer Software.

Die durch den Selektionsprozess identifizierten Schlüsselmerkmale spiegeln charakteristische Eigenschaften von PE-Dateien wider, die für die Klassifizierung von hoher Relevanz sind. Besondere Beachtung finden dabei Merkmale wie *DllCharacteristics*, *Machine* und *Characteristics*, die einen direkten Einfluss auf das Verhalten und die Funktionalität der ausführbaren Dateien ausüben könnten und damit als potenzielle Indikatoren für Malware fungieren.

Die signifikante Wichtigkeit dieser Merkmale deutet auf ihre starke Diskriminierungsfähigkeit zwischen den Klassen hin. Insbesondere das Merkmal *DllCharacteristics* könnte auf spezielle Verhaltensmuster oder Ausführungseigenschaften hinweisen, die für Malware

typisch sind. Die Analyse und Interpretation dieser selektierten Merkmale ermöglicht tiefergehende Einblicke in die Struktur und Beschaffenheit von Malware und bildet somit die Grundlage für die Entwicklung robuster und zuverlässiger Malware-Erkennungssysteme.

6.2 Hyperparameter-Optimierung

Zunächst wurden die Klassifizierungsmodelle unter Verwendung von Standardkonfigurationen, also ohne spezifische Anpassungen der Hyperparameter, trainiert. Die erzielten Leistungsmetriken dieser initialen Modelle sind in Abbildung 6.2 zusammengefasst. Im Anschluss erfolgte eine systematische Hyperparameter-Optimierung für jedes der Modelle, um deren Leistungsfähigkeit zu steigern. Die Ergebnisse dieses Prozesses sind in Abbildung 6.3 dargestellt und zeigen eine konsistente Verbesserung über alle bewerteten Metriken hinweg.

Nach der Anpassung der Hyperparameter zeigt der RandomForestClassifier eine marginale Verbesserung in der Genauigkeit und eine leichte Erhöhung des durchschnittlichen Kreuzvalidierungsscores auf 0.9915. Obwohl diese Veränderungen gering erscheinen, bestätigen sie die Effektivität der Modellfeinabstimmung.

Deutlichere Verbesserungen sind beim GradientBoostingClassifier und XGBClassifier zu beobachten. Der GradientBoostingClassifier erfährt eine signifikante Steigerung seiner Genauigkeit auf 0.9922 und der durchschnittliche Kreuzvalidierungsscore erhöht sich auf 0.9904. Der XGBClassifier zeigt ähnliche Verbesserungen mit einer Genauigkeitssteigerung auf 0.9938 und einem Kreuzvalidierungsdurchschnitt von 0.9905, was die Bedeutung einer gründlichen Hyperparameter-Optimierung unterstreicht.

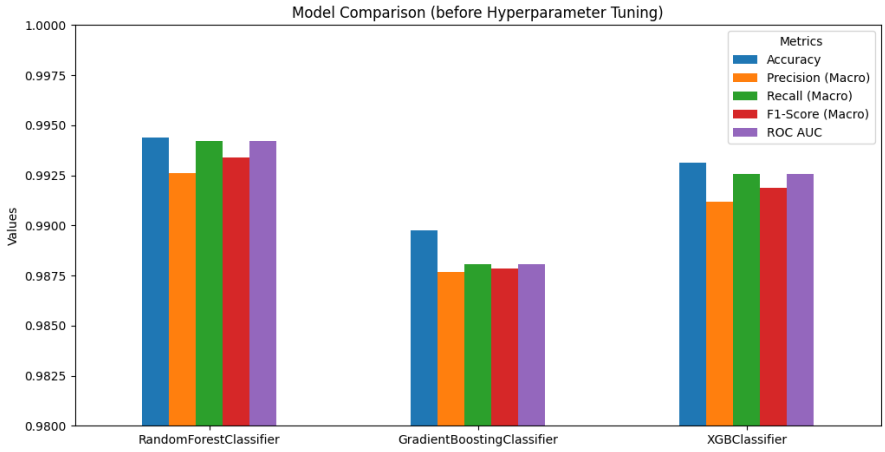


Abbildung 6.2: Vergleich der Modellleistung vor der Hyperparameter-Optimierung.

Die vorliegenden Ergebnisse verdeutlichen, dass eine Hyperparameter-Optimierung das volle Potenzial von Klassifizierungsmodellen auszuschöpft. Es wird deutlich, dass durch

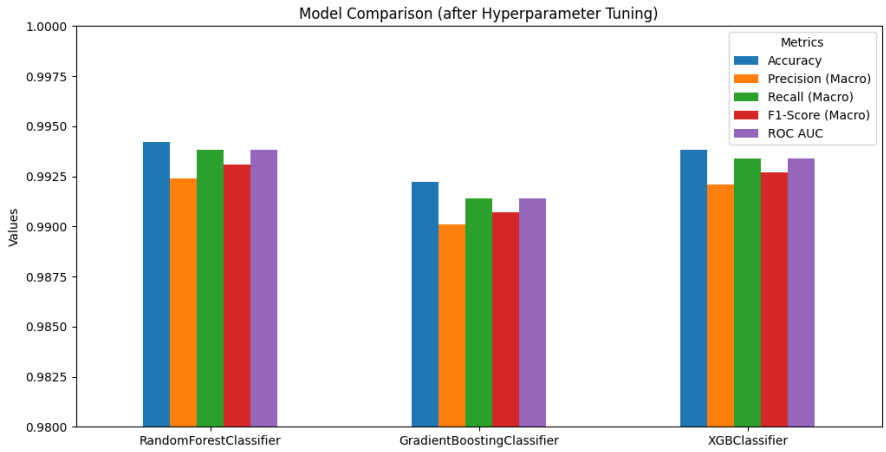


Abbildung 6.3: Vergleich der Modellleistung nach der Hyperparameter-Optimierung.

gezielte Anpassungen eine Verbesserung der Modellperformance erzielt werden kann, insbesondere bei Modellen, die nicht bereits unter Standardparametern optimale Ergebnisse liefern.

6.3 Modellleistung

Um die Leistung verschiedener Modelle zur Malware-Erkennung zu evaluieren, wurden Konfusionsmatrizen und verschiedene Evaluationsmetriken herangezogen. Die Konfusionsmatrizen geben Auskunft über die Anzahl der korrekt und inkorrekt klassifizierten Instanzen und sind für die Modelle GradientBoostingClassifier, RandomForestClassifier und XGBClassifier in den Abbildungen 6.5, 6.4 und 6.6 dargestellt. Die Metriken Genauigkeit (Accuracy), Präzision (Precision), Rückruf (Recall), F1-Score und der Bereich unter der ROC-Kurve (AUC) bieten einen umfassenden Überblick über die Performanz der Modelle.

Die Konfusionsmatrizen visualisieren die Anzahl der korrekt klassifizierten Malware- und legitimen Instanzen (True Positives und True Negatives) sowie die Fehlklassifikationen (False Positives und False Negatives). Die Diagonalelemente der Matrizen stellen die korrekt klassifizierten Instanzen dar, während die Nicht-Diagonalelemente die Fehler aufzeigen.

Die Evaluationsmetriken für jedes Modell sind in Tabelle **??** aufgeführt. Diese Metriken dienen als Indikatoren für die Qualität der Klassifizierung. Ein hoher Wert in einer der Metriken deutet darauf hin, dass das Modell in der Lage ist, Malware präzise zu identifizieren, ohne dabei viele legitime Programme fälschlicherweise als Malware zu klassifizieren.

Die Genauigkeit gibt den Anteil der Gesamtzahl korrekt klassifizierter Instanzen an und ist somit ein Maß für die Gesamtleistung des Modells. Präzision zeigt, wie viele der als Malware klassifizierten Instanzen tatsächlich Malware sind, während der Recall aufzeigt, wie viele der tatsächlichen Malware-Instanzen vom Modell erkannt wurden. Der F1-Score ist das harmonische Mittel von Präzision und Recall und bietet ein ausgewogenes Maß für Modelle, bei denen ein Kompromiss zwischen diesen beiden Metriken erforderlich ist. Der AUC-Wert zeigt die Wahrscheinlichkeit an, dass das Modell eine zufällige positive Instanz höher bewertet als eine zufällige negative Instanz.

Modell	Genauigkeit	Präzision	Recall	F1-Score	AUC
RandomForestClassifier	0.9942	0.9924	0.9938	0.9931	0.0.9915
GradientBoostingClassifier	0.9922	0.9901	0.9914	0.9907	0.9914
XGBClassifier	0.9938	0.9921	0.9934	0.9927	0.9934

Tabelle 6.1: Evaluationsmetriken für die Modelle RandomForestClassifier, GradientBoostingClassifier und XGBClassifier nach der Hyperparameter-Optimierung

Die Ergebnisse zeigen, dass alle Modelle eine hohe Genauigkeit und hervorragende Werte in allen anderen Metriken erzielen, wobei das RandomForestClassifier-Modell leicht besser abschneidet. Diese hohe Leistungsfähigkeit weist darauf hin, dass die Modelle gut geeignet sind, um Malware effektiv von legitimer Software zu unterscheiden. Es ist jedoch wichtig, die Modelle regelmäßig mit neuen Daten zu aktualisieren, um ihre Genauigkeit angesichts der sich ständig weiterentwickelnden Malware-Landschaft zu gewährleisten.

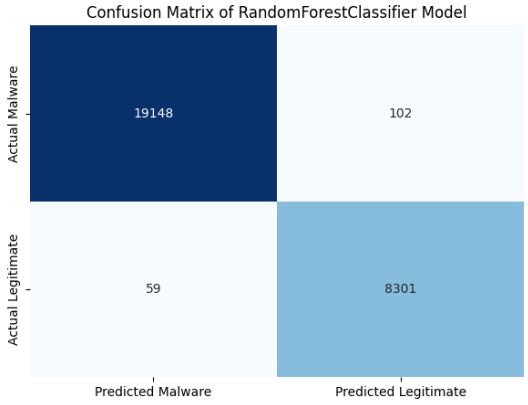


Abbildung 6.4: Konfusionsmatrix des RandomForestClassifier Modells

Die Konfusionsmatrizen und Evaluationsmetriken bietet nicht nur eine Basis für die Beurteilung der Modellleistung, sondern auch wichtige Einblicke für zukünftige Verbesserungen.

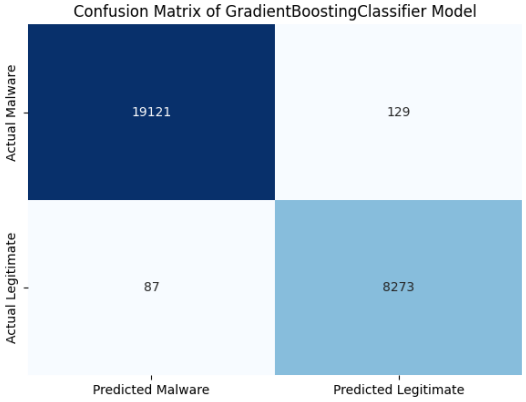


Abbildung 6.5: Konfusionsmatrix des GradientBoostingClassifier Modells

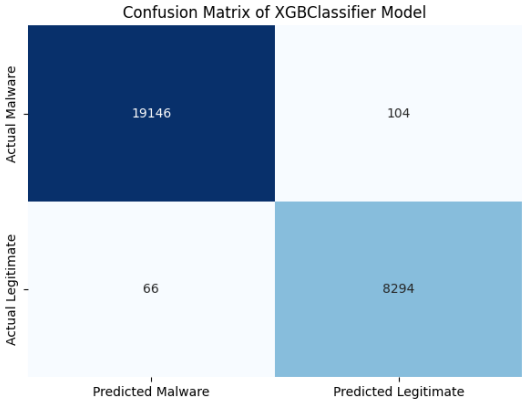


Abbildung 6.6: Konfusionsmatrix des XGBClassifier Modells

7. Zusammenfassung und Ausblick

Diese Arbeit hat sich eingehend mit der Anwendung von Künstlicher Intelligenz und Machine Learning in der Malware-Erkennung befasst. Im Zentrum stand dabei die Entwicklung und Evaluierung verschiedener ML-Modelle, die auf einen umfangreichen Datensatz von Malware- und legitimen Dateien angewendet wurden. Durch den Einsatz von Techniken wie Feature-Selektion, Hyperparameter-Optimierung und verschiedenen Klassifizierungsalgorithmen konnten Modelle entwickelt werden, die eine hohe Genauigkeit in der Unterscheidung zwischen Malware und legitimer Software aufweisen.

Die Feature-Selektion konnte die Komplexität der Modelle zu reduzieren und deren Vorhersagegenauigkeit zu verbessern. Durch die Auswahl relevanter Merkmale konnte eine effizientere Analyse und Klassifizierung von Dateien erreicht werden. Die Hyperparameter-Optimierung führte zu einer weiteren Verbesserung der Modellleistung, insbesondere bei Modellen, die nicht bereits unter Standardparametern optimale Ergebnisse lieferten.

Die Evaluierung der entwickelten Modelle zeigte, dass alle eine hohe Genauigkeit und gute Werte in allen Metriken aufwiesen. Besonders hervorzuheben ist die Leistung des RandomForestClassifier-Modells, das sich als besonders effektiv in der Malware-Erkennung erwies. Die Konfusionsmatrizen und Evaluationsmetriken boten wertvolle Einblicke in die Stärken und Schwächen der Modelle und lieferten wichtige Anhaltspunkte für zukünftige Verbesserungen.

Insgesamt zeigt diese Studie das enorme Potenzial von KI und ML in der Malware-Erkennung. Die entwickelten Modelle demonstrieren eine hohe Effizienz und Genauigkeit, die für die Abwehr moderner Cyberbedrohungen unerlässlich sind. Die kontinuierliche Weiterentwicklung und Anpassung dieser Technologien an neue Bedrohungsszenarien wird jedoch entscheidend sein, um die Sicherheit in der digitalen Welt zu gewährleisten.

In Zukunft könnten sich weitere Forschungsarbeiten auf die Integration von Deep Learning-Techniken konzentrieren, um noch komplexere Muster in Malware-Daten zu identifizieren. Zudem wäre die Untersuchung der Anwendung von KI-Modellen in Echtzeit-Umgebungen ein interessanter Aspekt, um ihre Effektivität und Reaktionsfähigkeit in dynamischen Sicherheitsszenarien zu bewerten.

Abschließend lässt sich festhalten, dass die Integration von KI und ML in die Malware-Erkennung einen bedeutenden Fortschritt in der Cybersicherheit darstellt. Die vorliegende Arbeit liefert einen wertvollen Beitrag zu diesem Forschungsfeld und unterstreicht die Notwendigkeit, die Entwicklungen in diesem Bereich kontinuierlich voranzutreiben.

8. Anlagen

Anlage 1

Jupyter Notebook

Das Jupyter Notebook mit dem vollständigen Code ist zusätzlich im digitalen Format beigefügt und kann in Github abgerufen werden:

Malware Detection with Machine Learning

Overview

In this project, we employ various machine learning algorithms to identify malware based on features extracted from executable files. The goal is to develop robust models capable of reliably detecting malware while achieving high accuracy and precision.

The data for this project is sourced from a collection of 42,323 legitimate binary files (exe, dll) and 96,724 malware files compiled by security blogger Prateek Lalwani. The malware data was obtained from the VirusShare website. The database can be downloaded from the following GitHub repository.

Deployment of Machine Learning

We leverage three different machine learning models to identify patterns in the data that may indicate malware presence. Various ensemble methods are employed, known for their high predictive accuracy and ability to discern complex patterns in large datasets.

In [ ]:
```
# Importing the required libraries and packages

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import os
from joblib import dump, load
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score, learning_curve, StratifiedKFold
from sklearn.ensemble import ExtraTreesClassifier, RandomForestClassifier, GradientBoostingClassifier
from sklearn.feature_selection import SelectFromModel
from sklearn.metrics import confusion_matrix, accuracy_score, roc_auc_score, precision_score, recall_score, f1_score
from xgboost import XGBClassifier
import pefile
import math
from collections import Counter
from skopt import BayesSearchCV
```

In [ ]:
```
# Import of Legitimate and Malware PE Data
malware_data = pd.read_csv("MalwareData.csv.gz", sep="|")
```

Splitting the Dataset into Legitimate Software and Malware

In the following cell, we perform data splitting to separate the entire dataset into two distinct categories: legitimate software and malware.

- **Legitimate Software**: These entries in the dataset represent legitimate and secure software, identified by a value of `1` in the `legitimate` field.
- **Malware**: These entries represent malicious software or malware, marked with a value of `0` in the `legitimate` field.

Data Exploration

To gain an initial understanding of the data, we provide some basic information:

- **Dataset Size**: We display the number of samples (entries) and features (columns) for both legitimate software and malware datasets.
- **First 5 Rows**: We showcase the first 5 rows of the entire dataset to offer a glimpse of the data's contents.

This initial exploration is instrumental in comprehending the dataset's structure and confirming the accuracy of the data split.

In [ ]:
```
# Splitting the dataset into legitimate software and malware
legitimate_data = malware_data[malware_data['legitimate'] == 1]
malware_data_only = malware_data[malware_data['legitimate'] == 0]

# Outputting the size of both datasets
print("Size of the legitimate dataset: {} samples, {} features".format(legitimate_data.shape[0], legitimate_data.shape[1] - 1))
print("Size of the malware dataset: {} samples, {} features".format(malware_data_only.shape[0], malware_data_only.shape[1] - 1))

# Displaying the column names and the first 5 rows of the entire dataset
display(malware_data.head(5))
```

```
Size of the legitimate dataset: 41323 samples, 56 features
Size of the malware dataset: 96724 samples, 56 features
```

	Name	md5	Machine	SizeOfOptionalHeader	Characteristics	MajorLinkerVersion	MinorLinkerVersion	SizeOfCode	SizeOfI
0	memtest.exe	631ea355665f28d4707448e442fbf5b8	332	224	258	9	0	361984	
1	ose.exe	9d10f99a6712e28f8acd5641e3a7ea6b	332	224	3330	9	0	130560	
2	setup.exe	4d92f518527353c0db88a70fddcfd390	332	224	3330	9	0	517120	
3	DW20.EXE	a41e524f8d45f0074fd07805ff0c9b12	332	224	258	9	0	585728	
4	dwtrig20.exe	c87e561258f2f8650cef999bf643a731	332	224	258	9	0	294912	

5 rows × 57 columns

Preparing Data for Machine Learning

In this section, we prepare the dataset for machine learning by extracting the labels and removing irrelevant features.

Code Explanation

We have a function `prepare_data_for_ml(data)` that performs the following steps:

1. **Extracting Labels**: We extract the labels from the dataset, which are stored in the 'legitimate' column, and convert them into an array.
2. **Removing Irrelevant Features**: We remove certain features that are not relevant for our machine learning task. In this case, we drop the 'Name', 'md5', and 'legitimate' columns from the dataset.

Data Preparation

We then call this function to prepare our data for machine learning:

- `features` now contains the relevant dataset features that will be used as input for machine learning algorithms.
- `labels` contains the corresponding labels (1 for legitimate, 0 for malware) that our machine learning model will predict.

This step is crucial in getting the data ready for training and evaluation of machine learning models.

In [ ]:
```
def prepare_data_for_ml(data):

    # Extracting the labels and removing irrelevant features
    labels = data['legitimate'].values
    features = data.drop(['Name', 'md5', 'legitimate'], axis=1).values
    return features, labels

# Preparing the data for machine learning
features, labels = prepare_data_for_ml(malware_data)
```

Feature Selection using ExtraTreesClassifier

In this section, we perform feature selection using the `ExtraTreesClassifier` algorithm to identify important features and create a reduced feature set.

Code Explanation

We have a function `select_features(features, labels, malware_data)` that performs the following steps:

1. **Fitting the Model**: We fit an `ExtraTreesClassifier` model to the input features and labels to evaluate feature importances.
2. **Feature Selection**: We use `SelectFromModel` to select important features and create a reduced feature set (`reduced_features`).
3. **Calculating Feature Importances**: We calculate feature importances and sort them in descending order.
4. **Displaying Feature Counts**: We display the number of features before and after feature selection to show the reduction in dimensionality.
5. **Selected Feature Indices**: We obtain the indices of the selected features and their corresponding names.
6. **Displaying Data Table**: We create a table to display the selected features for the first 5 files in the dataset.
7. **Visualizing Feature Importances**: We visualize feature importances using a bar chart.

Returned Values

The function returns the reduced features (`reduced_features`) and the selected feature indices (`selected_feature_indices`).

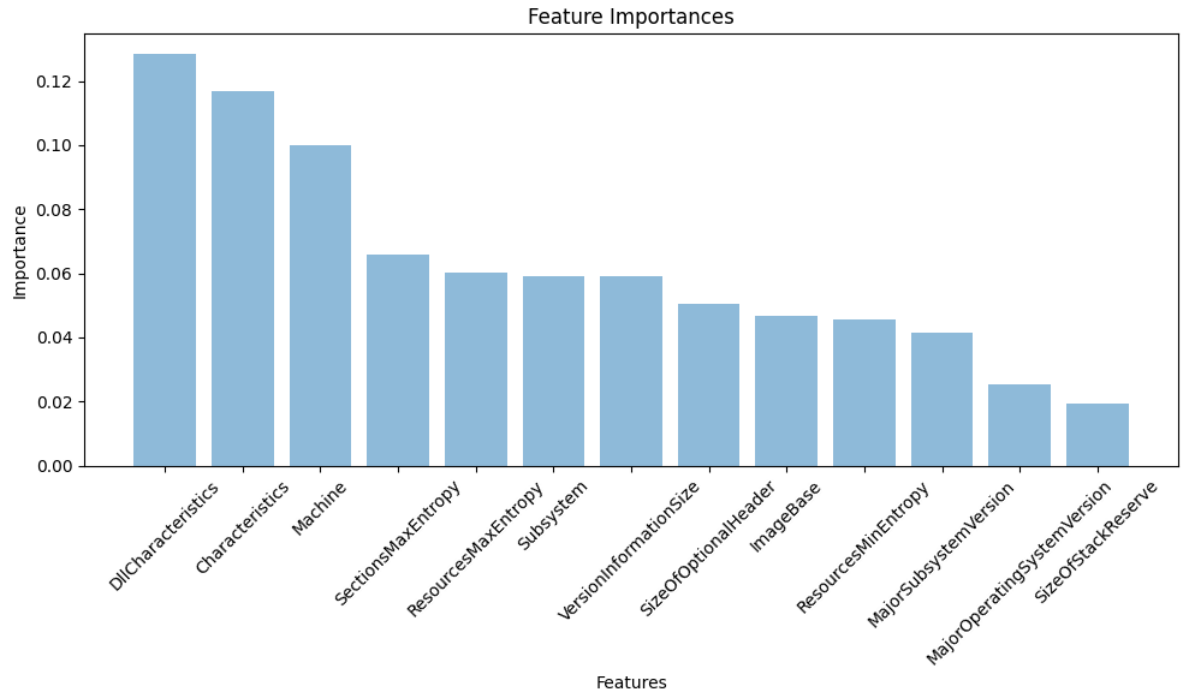
You can use these reduced features for further analysis and modeling.

In [ ]:
```
def select_features(features, labels, malware_data):

    # Fit an ExtraTreesClassifier model, select important features, and create a reduced feature set
    clf = ExtraTreesClassifier().fit(features, labels)
    model = SelectFromModel(clf, prefit=True)
    reduced_features = model.transform(features)

    # Calculate feature importances and sort them in descending order
    importances = clf.feature_importances_
    indices = np.argsort(importances)[::-1]

    # Displaying the number of features before and after feature selection
    print("Features before feature selection:", features.shape[1])
    print("Features after feature selection:", reduced_features.shape[1])

    # Getting the indices of selected features
    selected_feature_indices = model.get_support(indices=True)  # Indices of selected features
    selected_feature_names = malware_data.columns[selected_feature_indices + 2]  # +2 because the first two columns are 'Name' and 'md5'

    # Creating a list to store data for display
    data_to_display = []
    for i in range(5):
        file_name = malware_data['Name'].iloc[i]
        selected_feature_indices = model.get_support(indices=True)  # Indices of selected features
        selected_features = reduced_features[i]
        selected_feature_names = malware_data.columns[selected_feature_indices + 2]  # Headings of selected features

        # Individual values of selected features for the current file
        values_for_file = [file_name] + list(selected_features)
        data_to_display.append(values_for_file)

    # Creating a DataFrame and displaying the table
    table_columns = ['File'] + list(selected_feature_names)
    table_data = pd.DataFrame(data_to_display, columns=table_columns)
    display(table_data)

    # Visualizing feature importances
    plt.figure(figsize=(10, 6))
```

```
    plt.title('Feature Importances')
    plt.bar(range(reduced_features.shape[1]), importances[indices][:reduced_features.shape[1]],
            align='center', alpha=0.5)
    plt.xticks(range(reduced_features.shape[1]), malware_data.columns[2 + indices][:reduced_features.shape[1]], rotation=45)
    plt.xlabel('Features')
    plt.ylabel('Importance')
    plt.tight_layout()
    plt.show()

    # Returning reduced features and selected feature indices
    return reduced_features, selected_feature_indices

reduced_features, selected_feature_indices = select_features(features, labels, malware_data)
```

```
Features before feature selection: 54
Features after feature selection: 13
```

	File	Machine	SizeOfOptionalHeader	Characteristics	ImageBase	MajorOperatingSystemVersion	MajorSubsystemVersion	Subsystem	DllCharacteristic
0	memtest.exe	332.0	224.0	258.0	4194304.0	0.0	1.0	16.0	1024.
1	ose.exe	332.0	224.0	3330.0	771751936.0	5.0	5.0	2.0	33088.
2	setup.exe	332.0	224.0	3330.0	771751936.0	5.0	5.0	2.0	32832.
3	DW20.EXE	332.0	224.0	258.0	771751936.0	5.0	5.0	2.0	33088.
4	dwtrig20.exe	332.0	224.0	258.0	771751936.0	5.0	5.0	2.0	33088.



Train and Evaluate a Machine Learning Model

In this section, we have a function `train_and_evaluate_model(features, labels, model, params=None)` that allows us to train and evaluate a machine learning model. This function provides an option for hyperparameter tuning using GridSearchCV.

Code Explanation

The function takes the following parameters:

- `features` : The feature data.
- `labels` : The target labels.
- `model` : The machine learning model or estimator to be trained and evaluated.
- `params` (optional): Hyperparameter grid for GridSearchCV (used for hyperparameter tuning).

Data Splitting

We start by splitting the dataset into training and testing sets using `train_test_split` .

Hyperparameter Tuning

If hyperparameter tuning is required (i.e., `params` is provided), we use `GridSearchCV` to find the best hyperparameters for the model. The best hyperparameters are then used to initialize the model.

Model Evaluation

We calculate various evaluation metrics, including accuracy, precision, recall, F1-score, ROC AUC, and create a confusion matrix. These metrics are displayed in a DataFrame.

We also visualize the confusion matrix using a heatmap.

Finally, we print out the evaluation metrics, including cross-validation scores.

Returned Values

The function returns the trained model and a DataFrame containing model evaluation metrics.

You can use this function to train and evaluate different machine learning models with or without hyperparameter tuning.

In [ ]:
```python
# Train and evaluate a machine learning model with optional hyperparameter tuning using GridSearchCV
def train_and_evaluate_model(features, labels, model, params=None):

    # Split the dataset into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(features, labels, test_size=0.2, random_state=42)

    # Hyperparameter tuning using GridSearchCV
    if params:
        # Use GridSearchCV to find the best hyperparameters
        grid_search = GridSearchCV(estimator=model, param_grid=params, cv=5, scoring='accuracy', verbose=2)
        grid_search.fit(X_train, y_train)

        # Get the best hyperparameters
        best_params = grid_search.best_params_
        print("Best hyperparameters found:", best_params)

        # Initialize the model with the best hyperparameters
        model = model.set_params(**best_params)

    # Fit the model on the training data
    model.fit(X_train, y_train)

    # Make predictions on the test data
    y_pred = model.predict(X_test)

    # Calculate evaluation metrics
    cm = confusion_matrix(y_test, y_pred)
    acc = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, average='macro')
    recall = recall_score(y_test, y_pred, average='macro')
    f1 = f1_score(y_test, y_pred, average='macro')
    roc_auc = roc_auc_score(y_test, y_pred)

    cross_val_scores = cross_val_score(model, X_test, y_test, cv=5, scoring='accuracy')

    # Create a DataFrame to store model evaluation metrics with values rounded to 4 decimal places
    model_evaluation = pd.DataFrame({
        'Confusion Matrix': [cm],
        'Accuracy': [round(acc, 4)],
        'Precision (Macro)': [round(precision, 4)],
        'Recall (Macro)': [round(recall, 4)],
        'F1-Score (Macro)': [round(f1, 4)],
        'ROC AUC': [round(roc_auc, 4)],
        'Cross-Val-Scores (Accuracy)': [cross_val_scores],
    }, index=[type(model).__name__])

    # Display the model evaluation metrics DataFrame
    display(model_evaluation)

    # Visualize the confusion matrix
    plt.figure(figsize=(7, 5))
    sns.heatmap(cm, annot=True, fmt='g', cmap='Blues', cbar=False)
    plt.xticks([0.5, 1.5], ['Predicted Malware', 'Predicted Legitimate'])
    plt.yticks([0.5, 1.5], ['Actual Malware', 'Actual Legitimate'], rotation=90)
    plt.title('Confusion Matrix of {} Model'.format(type(model).__name__))
    plt.show()

    # Print evaluation metrics
    print(f'Accuracy: {acc:.4f}')
    print(f'Precision Score: {precision:.4f}')
    print(f'Recall Score: {recall:.4f}')
    print(f'F1 Score: {f1:.4f}')
    print(f'ROC AUC Score: {roc_auc:.4f}')

    print('Cross-Validation Scores:')
    for i, score in enumerate(cross_val_scores, start=1):
        print(f'Fold {i}: {score:.4f}')
    print(f'Mean Cross-Validation Score: {np.mean(cross_val_scores):.4f}')

    return model, model_evaluation
```

Saving a Machine Learning Model with Joblib

In this section, we have a function `save_model(model, name, directory='.')` that allows us to save a trained machine learning model to a specified directory using the joblib library.

Code Explanation

The function takes the following parameters:

- `model` : The trained machine learning model to be saved.
- `name` : The name of the model (used for the file name).
- `directory` (optional): The directory where the model will be saved. If not specified, it defaults to the current directory.

Returned Value

The function returns the file path at which the model is saved.

You can use this function to save trained machine learning models to a specific directory for later use.

In [ ]:
```python
# Save a machine learning model to a specified directory using joblib
def save_model(model, name, directory='.'):

    # Create the specified directory if it does not exist
    if not os.path.exists(directory):
        os.makedirs(directory)

    # Generate the file path for saving the model using the model name
    path = os.path.join(directory, f"{name.lower()}.joblib")

    # Save the model to the specified path
    dump(model, path)

    # Print a success message with the saved model's path
    print(f"{name} model was successfully saved at '{path}'.")

    return path
```

RandomForestClassifier

In this section, we train, evaluate, and save a RandomForestClassifier model.

Hyperparameter Tuning (Optional)

Hyperparameter tuning for RandomForestClassifier can be performed by specifying hyperparameter settings in the `rf_params` dictionary. Uncomment the desired hyperparameter settings as needed.

Train and Evaluate the Model

We train and evaluate the RandomForestClassifier model using the specified hyperparameters (if provided) and default parameters (commented out).

- The model is trained on `reduced_features` and `labels`.
- The evaluation results are stored in the `rf_model_evaluation` DataFrame.

Save the Trained Model

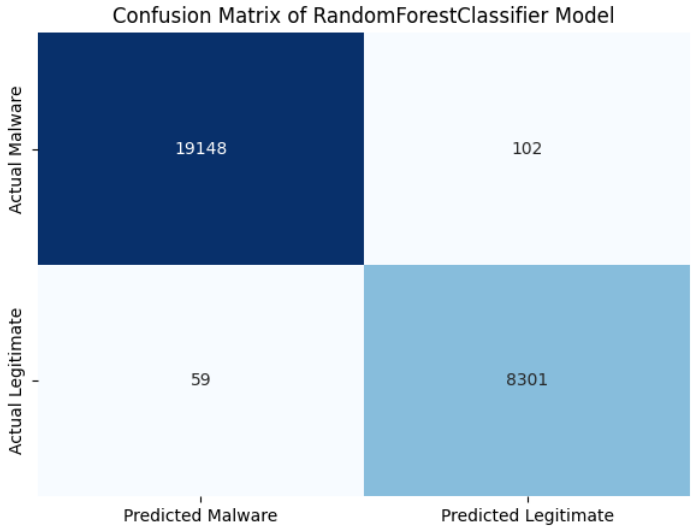
The trained RandomForestClassifier model is saved using the `save_model` function, and the file path is stored in `rf_save_path`.

You can uncomment the hyperparameter settings, as needed, to perform hyperparameter tuning, or use the default parameters to train the model. The saved model can be used for future predictions.

In [ ]:
```python
# RandomForestClassifier
print("RandomForestClassifier:")

rf_params = {
    # Hyperparameter settings go here
    # 'n_estimators': [50, 100, 200],
    # 'max_depth': [None, 10, 20, 30],
    # 'min_samples_split': [2, 5, 10],
    # 'min_samples_leaf': [1, 2, 4],
    # 'max_features': ['auto', 'sqrt'],
    # 'bootstrap': [True, False],
    # 'criterion': ['gini', 'entropy'],
    # 'class_weight': [None, 'balanced', {0: 1, 1: 5}, {0: 1, 1: 10}]
}
# Uncomment the hyperparameter settings above as needed.

# Train and evaluate the RandomForestClassifier model with the specified parameters
rf_model, rf_model_evaluation = train_and_evaluate_model(reduced_features, labels, RandomForestClassifier(random_state=42, max_depth=20, min_samp

# Train and evaluate the RandomForestClassifier model with default parameters
# rf_model, rf_model_evaluation = train_and_evaluate_model(reduced_features, labels, RandomForestClassifier(random_state=42), rf_params)

# Save the trained RandomForestClassifier model
rf_save_path = save_model(rf_model, "rf_model")
```

```
RandomForestClassifier:
```

	Confusion Matrix	Accuracy	Precision (Macro)	Recall (Macro)	F1-Score (Macro)	ROC AUC	Cross-Val-Scores (Accuracy)
RandomForestClassifier	[[19148, 102], [59, 8301]]	0.9942	0.9924	0.9938	0.9931	0.9938	[0.9927562477363274, 0.990220934444042, 0.9923...




```
Accuracy: 0.9942
Precision Score: 0.9924
Recall Score: 0.9938
F1 Score: 0.9931
ROC AUC Score: 0.9938
Cross-Validation Scores:
Fold 1: 0.9928
Fold 2: 0.9902
Fold 3: 0.9924
Fold 4: 0.9919
Fold 5: 0.9902
Mean Cross-Validation Score: 0.9915
rf_model model was successfully saved at '.\rf_model.joblib'.
```

GradientBoostingClassifier

In this section, we train, evaluate, and save a GradientBoostingClassifier model.

Hyperparameter Tuning (Optional)

Hyperparameter tuning for GradientBoostingClassifier can be performed by specifying hyperparameter settings in the `gb_params` dictionary. Uncomment the desired hyperparameter settings as needed.

Train and Evaluate the Model

We train and evaluate the GradientBoostingClassifier model using the specified hyperparameters (if provided) and default parameters (commented out).

- The model is trained on `reduced_features` and `labels`.
- The evaluation results are stored in the `gb_model_evaluation` DataFrame.

Save the Trained Model

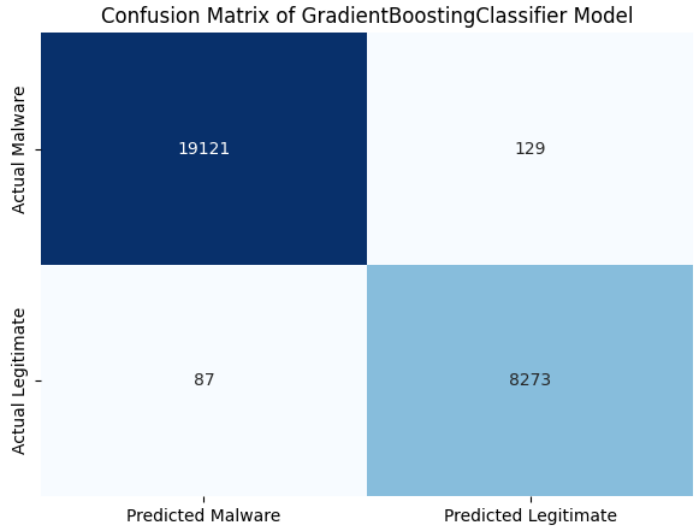
The trained GradientBoostingClassifier model is saved using the `save_model` function, and the file path is stored in `gb_save_path`.

You can uncomment the hyperparameter settings, as needed, to perform hyperparameter tuning, or use the default parameters to train the model. The saved model can be used for future predictions.

In [ ]:
```
# GradientBoostingClassifier
print("\nGradientBoostingClassifier:")

gb_params = {
    # Hyperparameter settings go here
    # 'n_estimators': [50, 100, 200],
    # 'learning_rate': [0.01, 0.1],
    # 'max_depth': [3, 5],
    # 'min_samples_split': [2, 5, 10],
    # 'min_samples_leaf': [1, 2, 4],
    # 'subsample': [0.8, 0.9, 1.0],
    # 'max_features': ['auto', 'sqrt'],
    # 'loss': ['deviance', 'exponential']
}
# Uncomment the hyperparameter settings above as needed.

# Train and evaluate the GradientBoostingClassifier model with the specified parameters
gb_model, gb_model_evaluation = train_and_evaluate_model(reduced_features, labels, GradientBoostingClassifier(random_state=42, learning_rate=0.1,

# Train and evaluate the GradientBoostingClassifier model with default parameters
# gb_model, gb_model_evaluation = train_and_evaluate_model(reduced_features, labels, GradientBoostingClassifier(random_state=42))

# Save the trained GradientBoostingClassifier model
gb_save_path = save_model(gb_model, "gb_model")
```

```
GradientBoostingClassifier:
```

	Confusion Matrix	Accuracy	Precision (Macro)	Recall (Macro)	F1-Score (Macro)	ROC AUC	Cross-Val-Scores (Accuracy)
GradientBoostingClassifier	[[19121, 129], [87, 8273]]	0.9922	0.9901	0.9914	0.9907	0.9914	[0.9913074972835929, 0.9900398406374502, 0.990...




```
Accuracy: 0.9922
Precision Score: 0.9901
Recall Score: 0.9914
F1 Score: 0.9907
ROC AUC Score: 0.9914
Cross-Validation Scores:
Fold 1: 0.9913
Fold 2: 0.9900
Fold 3: 0.9904
Fold 4: 0.9909
Fold 5: 0.9891
Mean Cross-Validation Score: 0.9904
gb_model model was successfully saved at '.\gb_model.joblib'.
```

XGBClassifier

In this section, we train, evaluate, and save an XGBoost (XGBClassifier) model.

Hyperparameter Tuning (Optional)

Hyperparameter tuning for XGBoost can be performed by specifying hyperparameter settings in the `xgb_params` dictionary. Uncomment the desired hyperparameter settings as needed.

Train and Evaluate the Model

We train and evaluate the XGBoost model using the specified hyperparameters (if provided) and default parameters (commented out).

- The model is trained on `reduced_features` and `labels`.
- The evaluation results are stored in the `xgb_model_evaluation` DataFrame.

Save the Trained Model

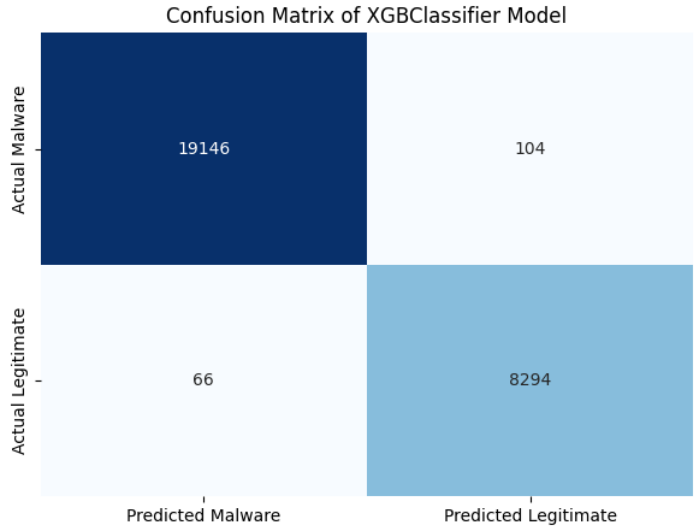
The trained XGBoost model is saved using the `save_model` function, and the file path is stored in `xgb_save_path`.

You can uncomment the hyperparameter settings, as needed, to perform hyperparameter tuning, or use the default parameters to train the model. The saved model can be used for future predictions.

In [ ]:
```python
# Example for training and evaluating an XGBoost model
print("XGBClassifier:")

xgb_params = {
    # Hyperparameter settings go here
    # 'n_estimators': [50, 100, 200],
    # 'learning_rate': [0.01, 0.1, 0.2],
    # 'max_depth': [3, 4, 5, 6],
    # 'min_child_weight': [1, 2, 3],
    # 'gamma': [0, 0.1, 0.2],
    # 'subsample': [0.8, 0.9, 1.0],
    # 'colsample_bytree': [0.8, 0.9, 1.0],
    # 'scale_pos_weight': [1, 2, 5]
}

# Train and evaluate the XGBoost model with the specified parameters
xgb_model, xgb_model_evaluation = train_and_evaluate_model(reduced_features, labels, XGBClassifier(random_state=42, objective='binary:logistic',

# Train and evaluate the XGBoost model with default parameters
# xgb_model, xgb_model_evaluation = train_and_evaluate_model(reduced_features, labels, XGBClassifier(random_state=42))

# Save the trained XGBoost model
xgb_save_path = save_model(xgb_model, "xgb_model")
```

```
XGBClassifier:
```

	Confusion Matrix	Accuracy	Precision (Macro)	Recall (Macro)	F1-Score (Macro)	ROC AUC	Cross-Val-Scores (Accuracy)
XGBClassifier	[[19146, 104], [66, 8294]]	0.9938	0.9921	0.9934	0.9927	0.9934	[0.9927562477363274, 0.9896776530242666, 0.991...




```
Accuracy: 0.9938
Precision Score: 0.9921
Recall Score: 0.9934
F1 Score: 0.9927
ROC AUC Score: 0.9934
Cross-Validation Scores:
Fold 1: 0.9928
Fold 2: 0.9897
Fold 3: 0.9915
Fold 4: 0.9900
Fold 5: 0.9888
Mean Cross-Validation Score: 0.9905
xgb_model model was successfully saved at '.\xgb_model.joblib'.
```

Model Comparison and Visualization

In this section, we combine the evaluation DataFrames for all models (RandomForestClassifier, GradientBoostingClassifier, XGBClassifier) and visualize the metrics as bar charts for comparison.

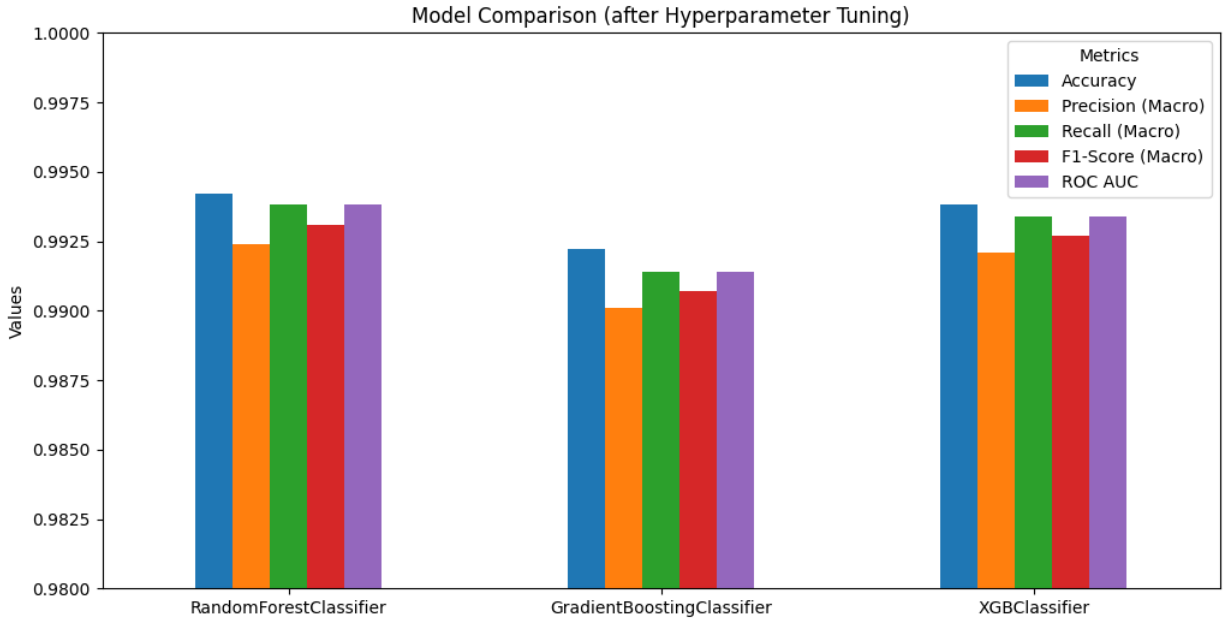
Combine Evaluation DataFrames

We combine the evaluation DataFrames for all models (`rf_model_evaluation` , `gb_model_evaluation` , `xgb_model_evaluation`) into a single DataFrame named `combined_df` .

Metric Visualization

We visualize the metrics as bar charts. This visualization allows us to compare the performance of different models based on the evaluation metrics.

In [ ]:
```python
# Combine the evaluation DataFrames for all models (rf_model_evaluation, gb_model_evaluation, xgb_model_evaluation)
combined_df = pd.concat([rf_model_evaluation, gb_model_evaluation, xgb_model_evaluation])

# Visualize the metrics as bar charts and limit the y-axis to the range from 0.95 to 1
ax = combined_df.plot(kind='bar', figsize=(12, 6), ylim=[0.98, 1.0])
plt.title('Model Comparison')
plt.ylabel('Values')
plt.xticks(rotation=0)
plt.legend(title='Metrics')
plt.show()
```



Funktionen zur Analyse von PE-Dateien

In diesem Codeabschnitt werden verschiedene Funktionen vorgestellt, die zur Analyse von PE-Dateien (Portable Executable) verwendet werden können. Diese Funktionen sind hilfreich, um wichtige Informationen aus einer PE-Datei zu extrahieren und Merkmale für die spätere Verwendung zu generieren.

calculate_entropy(data): Diese Funktion berechnet die Entropie eines gegebenen Datenstroms. Die Entropie ist ein Maß für die Unvorhersehbarkeit der Daten. Sie wird verwendet, um die "Unordnung" oder "Komplexität" eines Datenstroms zu quantifizieren.

process_resource_entry(entry): Diese Funktion verarbeitet die Ressourceneinträge einer PE-Datei. Sie durchläuft die Verzeichniseinträge und extrahiert Informationen über die Ressourcen, die in der PE-Datei enthalten sind. Dies kann nützlich sein, um Informationen über eingebettete Dateien oder andere Ressourcen in der PE-Datei zu extrahieren.

get_pefile_info(file_path, selected_feature_indices): Diese Funktion analysiert eine PE-Datei, extrahiert ausgewählte Merkmale und gibt sie zurück. Die ausgewählten Merkmale werden durch die `selected_feature_indices` festgelegt, und die Funktion extrahiert Informationen wie Maschinentyp, Größe der optionalen Header, Abschnittsgrößen und viele andere relevante Informationen aus der PE-Datei.

Die Funktion ist robust und behandelt auch Fälle, in denen bestimmte Attribute in der PE-Datei nicht vorhanden sind. Sie bietet eine umfassende Analysemöglichkeit für PE-Dateien und kann zur Feature-Extraktion in Malware-Erkennungsanwendungen verwendet werden.

Entropy Calculation

In this section, we have a function `calculate_entropy(data)` that calculates the entropy of a given dataset.

Code Explanation

The function takes the following steps to calculate entropy:

1. **Total Number of Bytes**: It calculates the total number of bytes in the input data using `len(data)`.
2. **Byte Count**: It counts the occurrences of each byte in the data using the `Counter` function.
3. **Entropy Initialization**: Entropy is initialized to 0.
4. **Entropy Calculation**: It calculates entropy.
5. **Entropy Value**: The calculated entropy value is returned.

You can use this function to compute the entropy of a dataset, which is a measure of the randomness or uncertainty in the data.

In [ ]:
```python
def calculate_entropy(data):
    # Calculate the total number of bytes in the data
    total_bytes = len(data)

    # Count the occurrences of each byte in the data
    byte_count = Counter(data)

    # Initialize entropy to 0
    entropy = 0

    # Calculate entropy using the formula: -Σ(p_i * log2(p_i))
    for count in byte_count.values():
        probability = count / total_bytes  # Calculate the probability of each byte
        entropy -= (probability * math.log2(probability))

    return entropy
```

Processing Resource Entries

In this section, we have a function `process_resource_entry(entry)` that processes resource entries in a structured way. Resource entries can represent various types of data or directories within a resource.

Code Explanation

The function takes a resource entry as input and performs the following actions:

1. **Initialize an Entries List**: It initializes an empty list called `entries` to store processed resource entries.
2. **Iterate Over Entry's Entities**: The function iterates over the entities within the input resource entry. Entities can represent directories, data, or other attributes.
3. **Handling Directories**: If an entity is a directory (`entity.directory` is True), the function recursively processes its entries by calling itself (`process_resource_entry(entity.directory)`). The results of the recursive call are extended into the `entries` list.
4. **Handling Data**: If an entity is not a directory and has data (`entity.data` is True), the function extracts information about the data, specifically the offset to data and its size, and adds this information as a tuple to the `entries` list.
5. **Handling Other Entities**: If an entity is neither data nor a directory, the function prints the attributes of the entity for inspection. This is useful for understanding the structure of the resource entry.
6. **Return Extracted Information**: Finally, the function returns the `entries` list, which contains information about the resource entries and their data.

This function provides a structured way to process and extract information from resource entries, making it easier to work with structured resource data.

In [ ]:
```python
def process_resource_entry(entry):
    entries = []  # Initialize a list to store resource entries

    for entity in entry.entries:
        if hasattr(entity, "directory") and entity.directory:
            # If it's a directory, recursively process its entries
            entries.extend(process_resource_entry(entity.directory))
        elif hasattr(entity, "data") and entity.data:
            # If it's not a directory and has data, extract information about the data
            entries.append(
                (entity.data.struct.OffsetToData, entity.data.struct.Size)
            )  # Here you can process the desired data
        else:
            # If it's neither data nor a directory, print its attributes
            print("Attributes of 'non-data and non-directory entity':")
            for attr_name in dir(entity):
                if not attr_name.startswith("__"):
                    attr_value = getattr(entity, attr_name)
                    print(f"{attr_name}: {attr_value}")
            print("\n")

    return entries  # Return a list of extracted information about resource entries
```

Analyzing PE Files with `get_pefile_info` Function

In this notebook, we will explore the `get_pefile_info` function, which analyzes Portable Executable (PE) files using the pefile library. This function extracts various features from PE files based on selected feature indices.

Parameters

- `file_path` (str): The path to the PE file to be analyzed.
- `selected_feature_indices` (list): A list of indices specifying which features to extract from the PE file.

Function Description

The `get_pefile_info` function performs the analysis of PE files and extracts relevant features. It can be used to gather information about the structure and characteristics of PE files.

In [ ]:
```python
def get_pefile_info(file_path, selected_feature_indices):
    try:
        # Print the analysis message including the file name
        print("PE Analysis of file", os.path.basename(file_path), ":")

        # Load the PE file using the pefile library
        pe = pefile.PE(file_path)

        # Initialize a list to store selected features from the PE file
        file_selected_features = []

        # Extract entropy values for sections (example values for the first 2 sections)
        sections_entropy = [section.get_entropy() for section in pe.sections]

        # Extract sizes of raw data for sections
        sections_raw_size = [section.SizeOfRawData for section in pe.sections]

        # Extract virtual sizes of sections
        sections_virtual_size = [section.Misc_VirtualSize for section in pe.sections]

        # Check if the PE file has a DIRECTORY_ENTRY_RESOURCE
        if hasattr(pe, "DIRECTORY_ENTRY_RESOURCE"):
            # Process resource entries and calculate resource entropies and sizes
            resource_data = process_resource_entry(pe.DIRECTORY_ENTRY_RESOURCE)

            # Initialize variables to track minimum and maximum resource entropies
            resources_min_entropy = float("inf")
```

```
            resources_max_entropy = float("-inf")

            # Lists to store resource entropies and sizes
            resource_entropies = []
            resource_sizes = []

            # Iterate through resource data entries
            for offset, size in resource_data:
                # Calculate entropy for the resource data
                entropy = calculate_entropy(
                    pe.get_memory_mapped_image()[offset : offset + size]
                )
                resource_entropies.append(entropy)
                resource_sizes.append(size)

                # Update minimum and maximum resource entropies
                resources_min_entropy = min(resources_min_entropy, entropy)
                resources_max_entropy = max(resources_max_entropy, entropy)
        else:
            # If DIRECTORY_ENTRY_RESOURCE is not found, set default values
            resource_entropies = [0]
            resource_sizes = [0]
            print("The 'DIRECTORY_ENTRY_RESOURCE' attribute was not found.")
            print("Resource entropy and size are set to 0.")

        # Iterate through selected feature indices
        for selected_index in selected_feature_indices:
            # Use a switch-like structure to match the selected index
            match selected_index:

                case 0:
                    # Index 0: Extract and print the 'Machine' value from FILE_HEADER
                    machine_value = pe.FILE_HEADER.Machine
                    print("Machine:", machine_value)
                    file_selected_features.append(machine_value)

                case 1:
                    # Index 1: Extract and print the 'SizeOfHeaders' value from OPTIONAL_HEADER
                    optional_header_size = pe.OPTIONAL_HEADER.SizeOfHeaders
                    print("SizeOfHeaders:", optional_header_size)
                    file_selected_features.append(optional_header_size)

                case 2:
                    # Index 2: Extract and print the 'Characteristics' value from FILE_HEADER
                    characteristics_value = pe.FILE_HEADER.Characteristics
                    print("Characteristics:", characteristics_value)
                    file_selected_features.append(characteristics_value)

                case 3:
                    # Index 3: Extract and print the 'MajorLinkerVersion' value from OPTIONAL_HEADER
                    major_linker_version = pe.OPTIONAL_HEADER.MajorLinkerVersion
                    print("MajorLinkerVersion:", major_linker_version)
                    file_selected_features.append(major_linker_version)

                case 4:
                    # Index 4: Extract and print the 'MinorLinkerVersion' value from OPTIONAL_HEADER
                    minor_linker_version = pe.OPTIONAL_HEADER.MinorLinkerVersion
                    print("MinorLinkerVersion:", minor_linker_version)
                    file_selected_features.append(minor_linker_version)

                case 5:
                    # Index 5: Extract and print the 'SizeOfCode' value from OPTIONAL_HEADER
                    size_of_code = pe.OPTIONAL_HEADER.SizeOfCode
                    print("SizeOfCode:", size_of_code)
                    file_selected_features.append(size_of_code)

                case 6:
                    # Index 6: Extract and print the 'SizeOfInitializedData' value from OPTIONAL_HEADER
                    size_of_initialized_data = pe.OPTIONAL_HEADER.SizeOfInitializedData
                    print("SizeOfInitializedData:", size_of_initialized_data)
                    file_selected_features.append(size_of_initialized_data)

                case 7:
                    # Index 7: Extract and print the 'SizeOfUninitializedData' value from OPTIONAL_HEADER
                    size_of_uninitialized_data = pe.OPTIONAL_HEADER.SizeOfUninitializedData
                    print("SizeOfUninitializedData:", size_of_uninitialized_data)
                    file_selected_features.append(size_of_uninitialized_data)

                case 8:
                    # Index 8: Extract and print the 'AddressOfEntryPoint' value from OPTIONAL_HEADER
                    address_of_entry_point = pe.OPTIONAL_HEADER.AddressOfEntryPoint
                    print("AddressOfEntryPoint:", address_of_entry_point)
                    file_selected_features.append(address_of_entry_point)

                case 9:
                    # Index 9: Extract and print the 'BaseOfCode' value from OPTIONAL_HEADER
                    base_of_code = pe.OPTIONAL_HEADER.BaseOfCode
                    print("BaseOfCode:", base_of_code)
                    file_selected_features.append(base_of_code)

                case 10:
                    # Index 10: Calculate and print the 'BaseOfData' value based on 'BaseOfCode' and 'SizeOfCode'
                    base_of_data = base_of_code + size_of_code
                    print("BaseOfData:", base_of_data)
                    file_selected_features.append(base_of_data)

                case 11:
                    # Index 11: Extract and print the 'ImageBase' value from OPTIONAL_HEADER
                    image_base = pe.OPTIONAL_HEADER.ImageBase
                    print("ImageBase:", image_base)
```

```
            file_selected_features.append(image_base)

        case 12:
            # Index 12: Extract and print the 'SectionAlignment' value from OPTIONAL_HEADER
            section_alignment = pe.OPTIONAL_HEADER.SectionAlignment
            print("SectionAlignment:", section_alignment)
            file_selected_features.append(section_alignment)

        case 13:
            # Index 13: Extract and print the 'FileAlignment' value from OPTIONAL_HEADER
            file_alignment = pe.OPTIONAL_HEADER.FileAlignment
            print("FileAlignment:", file_alignment)
            file_selected_features.append(file_alignment)

        case 14:
            # Index 14: Extract and print the 'MajorOperatingSystemVersion' value from OPTIONAL_HEADER
            major_os_version = pe.OPTIONAL_HEADER.MajorOperatingSystemVersion
            print("MinorOperatingSystemVersion:", major_os_version)
            file_selected_features.append(major_os_version)

        case 15:
            # Index 15: Extract and print the 'MinorOperatingSystemVersion' value from OPTIONAL_HEADER
            minor_os_version = pe.OPTIONAL_HEADER.MinorOperatingSystemVersion
            print("MinorOperatingSystemVersion:", minor_os_version)
            file_selected_features.append(minor_os_version)

        case 16:
            # Index 16: Extract and print the 'MajorImageVersion' value from OPTIONAL_HEADER
            major_image_version = pe.OPTIONAL_HEADER.MajorImageVersion
            print("MajorImageVersion:", major_image_version)
            file_selected_features.append(major_image_version)

        case 17:
            # Index 17: Extract and print the 'MinorImageVersion' value from OPTIONAL_HEADER
            minor_image_version = pe.OPTIONAL_HEADER.MinorImageVersion
            print("MinorImageVersion:", minor_image_version)
            file_selected_features.append(minor_image_version)

        case 18:
            # Index 18: Extract and print the 'MajorSubsystemVersion' value from OPTIONAL_HEADER
            major_subsystem_version = pe.OPTIONAL_HEADER.MajorSubsystemVersion
            print("MajorSubsystemVersion:", major_subsystem_version)
            file_selected_features.append(major_subsystem_version)

        case 19:
            # Index 19: Extract and print the 'MinorSubsystemVersion' value from OPTIONAL_HEADER
            minor_subsystem_version = pe.OPTIONAL_HEADER.MinorSubsystemVersion
            print("MinorSubsystemVersion:", minor_subsystem_version)
            file_selected_features.append(minor_subsystem_version)

        case 20:
            # Index 20: Extract and print the 'SizeOfImage' value from OPTIONAL_HEADER
            size_of_image = pe.OPTIONAL_HEADER.SizeOfImage
            print("SizeOfImage:", size_of_image)
            file_selected_features.append(size_of_image)

        case 21:
            # Index 21: Extract and print the 'SizeOfHeaders' value from OPTIONAL_HEADER
            size_of_headers = pe.OPTIONAL_HEADER.SizeOfHeaders
            print("SizeOfHeaders:", size_of_headers)
            file_selected_features.append(size_of_headers)

        case 22:
            # Index 22: Extract and print the 'CheckSum' value from OPTIONAL_HEADER
            check_sum = pe.OPTIONAL_HEADER.CheckSum
            print("CheckSum:", check_sum)
            file_selected_features.append(check_sum)

        case 23:
            # Index 23: Extract and print the 'Subsystem' value from OPTIONAL_HEADER
            subsystem = pe.OPTIONAL_HEADER.Subsystem
            print("Subsystem:", subsystem)
            file_selected_features.append(subsystem)

        case 24:
            # Index 24: Extract and print the 'DllCharacteristics' value from OPTIONAL_HEADER
            dll_characteristics = pe.OPTIONAL_HEADER.DllCharacteristics
            print("DllCharacteristics:", dll_characteristics)
            file_selected_features.append(dll_characteristics)

        case 25:
            # Index 25: Extract and print the 'SizeOfStackReserve' value from OPTIONAL_HEADER
            size_of_stack_reserve = pe.OPTIONAL_HEADER.SizeOfStackReserve
            print("SizeOfStackReserve:", size_of_stack_reserve)
            file_selected_features.append(size_of_stack_reserve)

        case 26:
            # Index 26: Extract and print the 'SizeOfStackCommit' value from OPTIONAL_HEADER
            size_of_stack_commit = pe.OPTIONAL_HEADER.SizeOfStackCommit
            print("SizeOfStackCommit:", size_of_stack_commit)
            file_selected_features.append(size_of_stack_commit)

        case 27:
            # Index 27: Extract and print the 'SizeOfHeapReserve' value from OPTIONAL_HEADER
            size_of_heap_reserve = pe.OPTIONAL_HEADER.SizeOfHeapReserve
            print("SizeOfHeapReserve:", size_of_heap_reserve)
            file_selected_features.append(size_of_heap_reserve)

        case 28:
            # Index 28: Extract and print the 'SizeOfHeapCommit' value from OPTIONAL_HEADER
```

```
                size_of_heap_commit = pe.OPTIONAL_HEADER.SizeOfHeapCommit
                print("SizeOfHeapCommit:", size_of_heap_commit)
                file_selected_features.append(size_of_heap_commit)

            case 29:
                # Index 29: Extract and print the 'LoaderFlags' value from OPTIONAL_HEADER
                loader_flags = pe.OPTIONAL_HEADER.LoaderFlags
                print("LoaderFlags:", loader_flags)
                file_selected_features.append(loader_flags)

            case 30:
                # Index 30: Extract and print the 'NumberOfRvaAndSizes' value from OPTIONAL_HEADER
                number_of_rva_and_sizes = pe.OPTIONAL_HEADER.NumberOfRvaAndSizes
                print("NumberOfRvaAndSizes:", number_of_rva_and_sizes)
                file_selected_features.append(number_of_rva_and_sizes)

            case 31:
                # Index 31: Extract and print the number of sections (SectionsNb)
                sections_nb = len(pe.sections)
                print("SectionsNb:", sections_nb)
                file_selected_features.append(sections_nb)

            case 32:
                # Index 32: Extract and print the mean entropy of sections (SectionsMeanEntropy)
                sections_mean_entropy = np.mean(sections_entropy)
                print("SectionsMeanEntropy:", sections_mean_entropy)
                file_selected_features.append(sections_mean_entropy)

            case 33:
                # Index 33: Extract and print the minimum entropy of sections (SectionsMinEntropy)
                section_min_entropy = np.min(sections_entropy)
                print("SectionsMinEntropy:", section_min_entropy)
                file_selected_features.append(section_min_entropy)

            case 34:
                # Index 34: Extract and print the maximum entropy of sections (SectionsMaxEntropy)
                sections_max_entropy = np.max(sections_entropy)
                print("SectionsMaxEntropy:", sections_max_entropy)
                file_selected_features.append(sections_max_entropy)

            case 35:
                # Index 35: Extract and print the mean raw size of sections (SectionsMeanRawsize)
                sections_mean_rawsize = np.mean(sections_raw_size)
                print("SectionsMeanRawsize:", sections_mean_rawsize)
                file_selected_features.append(sections_mean_rawsize)

            case 36:
                # Index 36: Extract and print the minimum raw size of sections (SectionsMinRawsize)
                sections_min_rawsize = np.min(sections_raw_size)
                print("SectionsMinRawsize:", sections_min_rawsize)
                file_selected_features.append(sections_min_rawsize)

            case 37:
                # Index 37: Extract and print the maximum raw size of sections (SectionsMaxRawsize)
                section_max_rawsize = np.max(sections_raw_size)
                print("SectionsMaxRawsize:", section_max_rawsize)
                file_selected_features.append(section_max_rawsize)

            case 38:
                # Index 38: Extract and print the mean virtual size of sections (SectionsMeanVirtualsize)
                sections_mean_virtualsize = np.mean(sections_virtual_size)
                print("SectionsMeanVirtualsize:", sections_mean_virtualsize)
                file_selected_features.append(sections_mean_virtualsize)

            case 39:
                # Index 39: Extract and print the minimum virtual size of sections (SectionsMinVirtualsize)
                sections_min_virtualsize = np.min(sections_virtual_size)
                print("SectionsMinVirtualsize:", sections_min_virtualsize)
                file_selected_features.append(sections_min_virtualsize)

            case 40:
                # Index 40: Extract and print the maximum virtual size of sections (SectionMaxVirtualsize)
                section_max_virtualsize = np.max(sections_virtual_size)
                print("SectionMaxVirtualsize:", section_max_virtualsize)
                file_selected_features.append(section_max_virtualsize)

            case 41:
                # Index 41: Extract and print the number of DLL imports (ImportsNbDLL)
                imports_nb_dll = len(pe.DIRECTORY_ENTRY_IMPORT)
                print("ImportsNbDLL:", imports_nb_dll)
                file_selected_features.append(imports_nb_dll)


            case 42:
                # Index 42: Extract and print the number of imports (imports_nb)
                if pe.DIRECTORY_ENTRY_IMPORT:
                    imports_nb = sum(
                        len(entry.imports) for entry in pe.DIRECTORY_ENTRY_IMPORT
                    )
                else:
                    imports_nb = 0
                print("ImportsNb:", imports_nb)
                file_selected_features.append(imports_nb)

            case 43:
                # Index 43: Extract and print the number of imports with ordinals (imports_nb_ordinal)
                if hasattr(pe, "DIRECTORY_ENTRY_IMPORT"):
                    imports_nb_ordinal = sum(
                        len(entry.imports)
                        for entry in pe.DIRECTORY_ENTRY_IMPORT
```

```
                        if entry.imports[0].ordinal
                    )
                else:
                    imports_nb_ordinal = 0
                print("ImportsNbOrdinal:", imports_nb_ordinal)
                file_selected_features.append(imports_nb_ordinal)

            case 44:
                # Index 44: Extract and print the number of exports (export_nb)
                if hasattr(pe, "DIRECTORY_ENTRY_EXPORT"):
                    export_nb = len(pe.DIRECTORY_ENTRY_EXPORT.symbols)
                else:
                    export_nb = 0
                print("ExportNb:", export_nb)
                file_selected_features.append(export_nb)

            case 45:
                # Index 45: Extract and print the number of resources (resources_nb)
                resources_nb = len(pe.DIRECTORY_ENTRY_RESOURCE.entries)
                print("ResourcesNb:", resources_nb)
                file_selected_features.append(resources_nb)

            case 46:
                # Index 46: Extract and print the mean entropy of resources (resources_mean_entropy)
                resources_mean_entropy = np.mean(resource_entropies)
                file_selected_features.append(resources_mean_entropy)

            case 47:
                # Index 47: Extract and print the minimum entropy of resources (resources_min_entropy)
                resources_min_entropy = np.min(resource_entropies)
                file_selected_features.append(resources_min_entropy)

            case 48:
                # Index 48: Extract and print the maximum entropy of resources (resources_max_entropy)
                resources_max_entropy = np.max(resource_entropies)
                file_selected_features.append(resources_max_entropy)

            case 49:
                # Index 49: Extract and print the mean size of resources (resources_mean_size)
                resources_mean_size = np.mean(resource_sizes)
                file_selected_features.append(resources_mean_size)

            case 50:
                # Index 50: Extract and print the minimum size of resources (resources_min_size)
                resources_min_size = np.min(resource_sizes)
                file_selected_features.append(resources_min_size)

            case 51:
                # Index 51: Extract and print the maximum size of resources (resources_max_size)
                resources_max_size = np.max(resource_sizes)
                file_selected_features.append(resources_max_size)

            case 52:
                # Index 52: Extract and print the 'LoadConfigurationSize' value from OPTIONAL_HEADER
                load_configuration_size = pe.OPTIONAL_HEADER.DATA_DIRECTORY[10].Size
                print("LoadConfigurationSize:", load_configuration_size)
                file_selected_features.append(load_configuration_size)

            case 53:
                # Index 53: Extract and print the 'VersionInformationSize' value from VS_FIXEDFILEINFO
                if hasattr(pe, "DIRECTORY_ENTRY_EXPORT"):
                    version_info_size = pe.VS_FIXEDFILEINFO[0].FileVersionLS
                else:
                    version_info_size = 0
                print("VersionInformationSize:", version_info_size)
                file_selected_features.append(version_info_size)

            case _:
                # Handle invalid selected indices
                print(
                    selected_index,
                    "is an invalid value in Selected Feature Indices",
                )

        # Return the selected features as a NumPy array
        return np.array([file_selected_features])

    except Exception as e:
        # Handle exceptions while reading the file
        print(f"Error while reading the file: {str(e)}")
        return None
```

Analyzing PE Files with `get_pefile_info` Function

In this notebook, we will explore the `get_pefile_info` function, which analyzes Portable Executable (PE) files using the pefile library. This function extracts various features from PE files based on selected feature indices.

Parameters

- `file_path` (str): The path to the PE file to be analyzed.
- `selected_feature_indices` (list): A list of indices specifying which features to extract from the PE file.

Function Description

The `get_pefile_info` function performs the analysis of PE files and extracts relevant features. It can be used to gather information about the structure and characteristics of PE files.

Analyzing a PE File

To analyze a PE file, follow these steps:

1. Run the cell below to enter the path to the PE file you want to analyze.
2. The function will check if the entered file path exists.
3. If the file path is valid, it will analyze the PE file and extract the selected features.

Make sure to replace `selected_feature_indices` with your list of selected feature indices for analysis.

In [ ]:
```
# Ask the user for the file path
file_path = input("Enter the path to the PE file to analyze: ")

# Check if the entered file path exists
if not os.path.exists(file_path):
    print("The specified file path does not exist. Make sure you have provided a valid path.")
else:
    # Analyze the PE file and extract features
    pe_info = get_pefile_info(file_path, selected_feature_indices)
```

```
PE Analysis of file Install League of Legends euw.exe :
Machine: 332
SizeOfHeaders: 1024
Characteristics: 290
ImageBase: 4194304
MinorOperatingSystemVersion: 6
MajorSubsystemVersion: 6
Subsystem: 2
DllCharacteristics: 33088
SizeOfStackReserve: 1048576
SectionsMaxEntropy: 7.996438966157736
VersionInformationSize: 0
```

Load Models and Analyze File

In this section, we load pre-trained machine learning models (RandomForest, GradientBoosting, XGBoost), make predictions on a file using these models, and display the results.

Code Explanation

We start by loading the pre-trained machine learning models if they are not already loaded.

Next, we create a dictionary `results` to store the predictions and probabilities for each model.

We collect results for all models:

- Predictions are made using each model for a given input data (`pe_info`).
- Probabilities for malware and legitimate classifications are obtained.

Results are stored in the `results` dictionary.

We then display the results for all models:

- For each model, we print the malware and legitimate probabilities.
- We classify the file as either malware or legitimate based on predictions.

Additionally, we provide an example of displaying probabilities as a bar chart to visualize the model's confidence in the classification.

Displaying Results

For each model, we display:

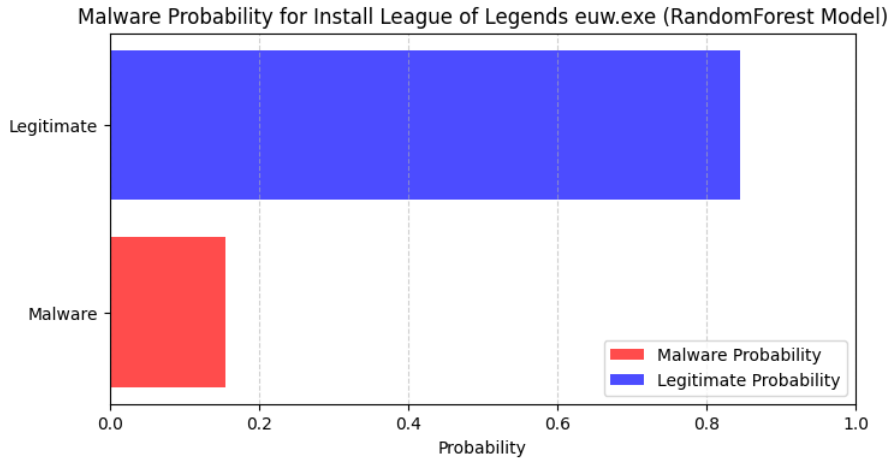
- Model name (e.g., RandomForest, GradientBoosting, XGBoost)
- Malware and legitimate probabilities
- The model's classification of the file (malware or legitimate)

We also visualize the probabilities as a bar chart to provide a visual representation of the model's confidence in its classification.

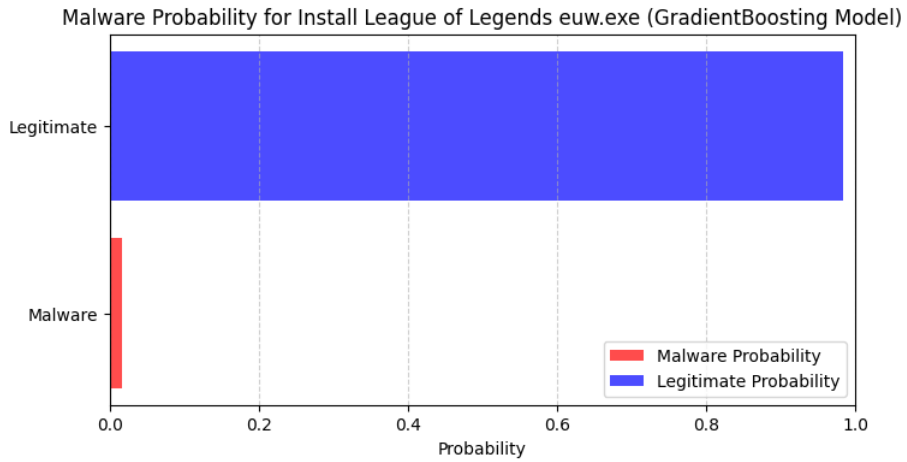
In [ ]:
```
# Load models (if not already loaded)
rf_model = load('./rf_model.joblib')
gb_model = load('./gb_model.joblib')
xgb_model = load('./xgb_model.joblib')

# Collect results for all models
results = {}

filename = os.path.basename(file_path)
print("Analyzing file ", filename)

# Predictions using RandomForestClassifier
rf_predictions = rf_model.predict(pe_info)
rf_probabilities = rf_model.predict_proba(pe_info)
results['RandomForest'] = {'Predictions': rf_predictions, 'Probabilities': rf_probabilities}

# Predictions using GradientBoostingClassifier
gb_predictions = gb_model.predict(pe_info)
gb_probabilities = gb_model.predict_proba(pe_info)
results['GradientBoosting'] = {'Predictions': gb_predictions, 'Probabilities': gb_probabilities}
```

```python
# Predictions using XGBoost model
xgb_predictions = xgb_model.predict(pe_info)
xgb_probabilities = xgb_model.predict_proba(pe_info)
results['XGBoost'] = {'Predictions': xgb_predictions, 'Probabilities': xgb_probabilities}

# Display results for all models
for model_name, model_results in results.items():
    predictions = model_results['Predictions']
    probabilities = model_results['Probabilities']

    print("\n{} Model:".format(model_name))
    print("{} model suspects malware with {:.2f}% probability and legitimate with {:.2f}% probability".format(model_name, (1 - probabilities[0][0

    if predictions == 1:
        print("{} model classified the file {} as malware.".format(model_name, filename))
    else:
        print("{} model classified the file {} as legitimate software.".format(model_name, filename))


    # Example of displaying probabilities as a bar chart
    plt.figure(figsize=(8, 4))
    plt.barh(['Malware'], [1 - probabilities[0][0]], color='red', alpha=0.7, label='Malware Probability')
    plt.barh(['Legitimate'], [probabilities[0][0]], color='blue', alpha=0.7, label='Legitimate Probability')
    plt.xlabel('Probability')
    plt.xlim(0, 1)
    plt.title('Malware Probability for {} ({} Model)'.format(filename, model_name))
    plt.grid(True, axis='x', linestyle='--', alpha=0.6)
    plt.legend()
    plt.show()
```

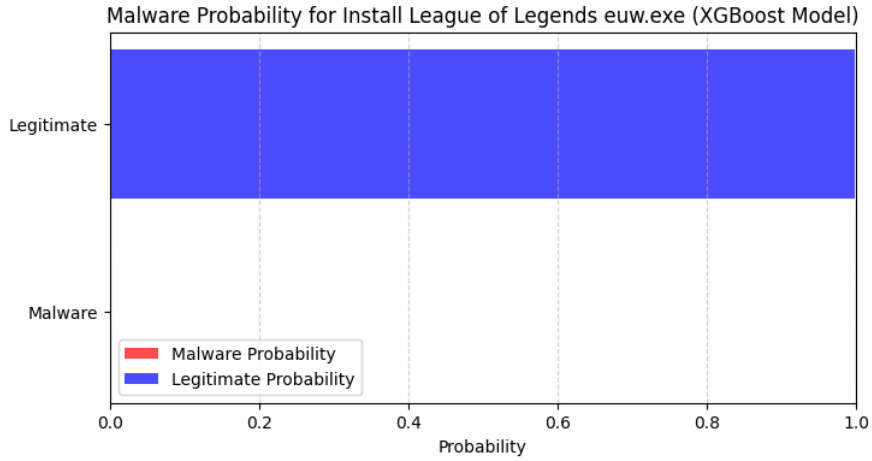
```
Analyzing file  Install League of Legends euw.exe

RandomForest Model:
RandomForest model suspects malware with 15.50% probability and legitimate with 84.50% probability
RandomForest model classified the file Install League of Legends euw.exe as legitimate software.
```




```
GradientBoosting Model:
GradientBoosting model suspects malware with 1.67% probability and legitimate with 98.33% probability
GradientBoosting model classified the file Install League of Legends euw.exe as legitimate software.
```




```
XGBoost Model:
XGBoost model suspects malware with 0.16% probability and legitimate with 99.84% probability
XGBoost model classified the file Install League of Legends euw.exe as legitimate software.
```


Malware Probability for Install League of Legends euw.exe (XGBoost Model)
Legitimate
Malware
Malware Probability
Legitimate Probability
0.0
0.2
0.4
0.6
0.8
1.0
Probability

Literatur

[1] B. für Sicherheit in der Informationstechnik. (2023). „Die Lage der IT-Sicherheit in Deutschland 2023“ [Online]. Verfügbar: https://www.bsi.bund.de/DE/Service-Navi/Publikationen/Lagebericht/lagebericht_node.html

[2] S. Naz und D. K. Singh, „Review of Machine Learning Methods for Windows Malware Detection“, in *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, IEEE, Juli 2019, S. 1–6. DOI: 10.1109/ICCCNT45670.2019.8944796.

[3] M. Klatte, „Machine Learning: Fluch oder Segen für die IT-Security“, *Digitale Welt*, Jg. 5, Nr. 1, S. 22–25, Dez. 2020, ISSN: 2569-1996. DOI: 10.1007/s42354-020-0317-1.

[4] R. Benzmüller, „Machine Learning und Virenschutz: Nützlich — aber noch viel zu lernen“, *Datenschutz und Datensicherheit - DuD*, Jg. 42, Nr. 4, S. 224–230, März 2018, ISSN: 1862-2607. DOI: 10.1007/s11623-018-0912-6.

[5] Mayo. (Mai 2016). „Machine Learning Key Terms, Explained“ [Online]. Verfügbar: https://www.kdnuggets.com/2016/05/machine-learning-key-terms-explained.html

[6] Mayo. (März 2016). „The Data Science Process, Rediscovered“ [Online]. Verfügbar: https://www.kdnuggets.com/2016/03/data-science-process-rediscovered.html

[7] B. Hariharan, R. Siva, S. Sadagopan, V. Mishra und Y. Raghav, „Malware Detection Using XGBoost based Machine Learning Models - Review“, in *2023 2nd International Conference on Edge Computing and Applications (ICECAA)*, IEEE, Juli 2023, S. 964–970. DOI: 10.1109/ICECAA58104.2023.10212327.

[8] K. Thosar, P. Tiwari, R. Jyothula und D. Ambawade, „Effective Malware Detection using Gradient Boosting and Convolutional Neural Network“, in *2021 IEEE Bombay Section Signature Conference (IBSSC)*, IEEE, Nov. 2021, S. 1–4. DOI: 10.1109/IBSSC53889.2021.9673266.

[9] M. Lehner und E. Hermann, „Auffinden von verschleierter Malware: Einsatz der heuristischen Analyse“, *Datenschutz und Datensicherheit - DuD*, Jg. 30, Nr. 12, S. 768–772, Dez. 2006, ISSN: 1862-2607. DOI: 10.1007/s11623-006-0237-8.

[10] 1. [Hellmann Roland, Hrsg., *IT-Sicherheit: Methoden und Schutzmaßnahmen für sichere Cybersysteme*, Deutsch, 1 Online-Ressource (X, 244 Seiten), Berlin ; Boston, 2023.

[11] H. Holm, „Signature Based Intrusion Detection for Zero-Day Attacks: (Not) A Closed Chapter?“, in *2014 47th Hawaii International Conference on System Sciences*, IEEE, Jan. 2014. DOI: 10.1109/hicss.2014.600.

[12] S. Choudhary und A. Sharma, „Malware Detection amp; Classification using Machine Learning“, in *2020 International Conference on Emerging Trends in Communication, Control and Computing (ICONC3)*, IEEE, Feb. 2020. DOI: 10.1109/iconc345789.2020.9117547.

[13] F. Adkins, L. Jones, M. Carlisle und J. Upchurch, „Heuristic malware detection via basic block comparison“, in *2013 8th International Conference on Malicious and Unwanted Software: “The Americas” (MALWARE)*, IEEE, Okt. 2013. DOI: 10.1109/malware.2013.6703680.

[14] L. Bilge und T. Dumitraş, „Before we knew it: an empirical study of zero-day attacks in the real world“, in *Proceedings of the 2012 ACM conference on Computer and communications security*, Ser. CCS'12, ACM, Okt. 2012. DOI: 10.1145/2382196.2382284.

[15] I. Alsmadi, B. Al-Ahmad und M. Alsmadi, „Malware analysis and multi-label category detection issues: Ensemble-based approaches“, in *2022 International Conference on Intelligent Data Science Technologies and Applications (IDSTA)*, IEEE, Sep. 2022. DOI: 10.1109/idsta55301.2022.9923057.

[16] Sudhakar und S. Kumar, „An emerging threat Fileless malware: a survey and research challenges“, *Cybersecurity*, Jg. 3, Nr. 1, Jan. 2020, ISSN: 2523-3246. DOI: 10.1186/s42400-019-0043-x.

[17] N. Pohlmann, *Cyber-Sicherheit: Das Lehrbuch für Konzepte, Prinzipien, Mechanismen, Architekturen und Eigenschaften von Cyber-Sicherheitssystemen in der Digitalisierung*. Springer Fachmedien Wiesbaden, 2022, ISBN: 9783658362430. DOI: 10.1007/978-3-658-36243-0.

[18] M. Haim, „Maschinelles Lernen mit Goldstandard („überwachtes Lernen“)“, in *Studienbücher zur Kommunikations- und Medienwissenschaft*. Springer Fachmedien Wiesbaden, 2023, S. 221–255, ISBN: 9783658401719. DOI: 10.1007/978-3-658-40171-9_10.

[19] J. Frochte, *Maschinelles Lernen, Grundlagen und Algorithmen in Python* (Hanser eLibrary), 3., überarbeitete und erweiterte Auflage. München: Hanser, 2021, 1608 S., ISBN: 9783446463554.

[20] K. Choo, E. Greplova, M. H. Fischer und T. Neupert, „Unüberwachtes Lernen“, in *essentials*. Springer Fachmedien Wiesbaden, 2020, S. 47–58, ISBN: 9783658322687. DOI: 10.1007/978-3-658-32268-7_4.

[21] H. S. Anderson und P. Roth, „EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models“, *ArXiv e-prints*, Apr. 2018. arXiv: 1804.04637 [cs.CR].

[22] C. Chebbi, *Mastering Machine Learning for Penetration Testing, Develop an extensive skill set to break self-learning systems using Python*, 1. Birmingham: Packt Publishing, 2018, 1276 S., ISBN: 9781788993111. [Online]. Verfügbar: https://books.google.de/books?id=bXJiDwAAQBAJ.

[23] F. C. C. Garcia und F. P. M. I. au2, *Random Forest for Malware Classification*, 2016. arXiv: 1609.07770 [cs.CR].

[24] D. Wu, P. Guo und P. Wang, „Malware Detection based on Cascading XGBoost and Cost Sensitive“, in *2020 International Conference on Computer Communication and Network Security (CCNS)*, IEEE, Aug. 2020, S. 201–205. DOI: 10.1109/CCNS50731.2020.00051.

[25] E. Bartz, T. Bartz-Beielstein, M. Zaefferer und O. Mersmann, Hrsg., *Hyperparameter Tuning for Machine and Deep Learning with R: A Practical Guide, A Practical Guide*. Singapore: Springer Nature Singapore, 2023, 1323 S., ISBN: 9789811951701. DOI: 10.1007/978-981-19-5170-1.

[26] M. Stamp, M. Alazab und A. Shalaginov, Hrsg., *Malware Analysis Using Artificial Intelligence and Deep Learning*. Cham: Springer, 2021, 651 S., Literaturangaben, ISBN: 978-3-030-62582-5. DOI: 10.1007/978-3-030-62582-5.

[27] B. M. Khammas, A. Monemi, J. S. Bassi, I. Ismail, S. M. Nor und M. N. Marsono, „Feature selection and machine learning classification for malware detection“, *Jurnal Teknologi*, Jg. 77, Nr. 1, S. 243–250, 2015.

[28] D. Ucci, L. Aniello und R. Baldoni, „Survey of machine learning techniques for malware analysis“, *Computers Security*, Jg. 81, S. 123–147, 2019, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2018.11.001. [Online]. Verfügbar: https://www.sciencedirect.com/science/article/pii/S0167404818303808.

[29] A. Campagner, J. Lienen, E. Hüllermeier und D. Ciucci, „Scikit-Weak: A Python Library for Weakly Supervised Machine Learning“, in *Rough Sets*, J. Yao, H. Fujita, X. Yue, D. Miao, J. Grzymala-Busse und F. Li, Hrsg., Cham: Springer Nature Switzerland, 2022, S. 57–70, ISBN: 978-3-031-21244-4.

[30] Y. H. Choi, B. J. Han, B. C. Bae, H. G. Oh und K. W. Sohn, „Toward extracting malware features for classification using static and dynamic analysis“, in *2012 8th International Conference on Computing and Networking Technology (INC, ICCIS and ICMIC)*, Aug. 2012, S. 126–129.

[31] C. Chebbi, *Mastering-Machine-Learning-for-Penetration-Testing*, Github, 2018. [Online]. Verfügbar: https://github.com/packtpublishing/mastering-machine-learning-for-penetration-testing.

[32] F. O. Gbenga, A. A. Olusola und O. O. Elohor, „Towards Optimization of Malware Detection using Extra-Tree and Random Forest Feature Selections on Ensemble Classifiers“, *International Journal of Recent Technology and Engineering*, Jg. 9, S. 223–232, März 2021, ISSN: 2277-3878.

[33] O. Aslan und A. A. Yilmaz, „A New Malware Classification Framework Based on Deep Learning Algorithms“, *IEEE Access*, Jg. 9, S. 87 936–87 951, 2021, ISSN: 2169-3536. DOI: 10.1109/access.2021.3089586.