Erkennung von Android-Malware mit maschinellem Lernen

Michael Stahlberger[1,2] · Tobias Straub[1]

[1]Duale Hochschule Baden-Württemberg
Center for Advanced Studies
tobias.straub@cas.dhbw.de

[2]Fiducia & GAD IT AG
Karlsruhe
michael.stahlberger@fiduciagad.de

Zusammenfassung

Schadsoftware ist mittlerweile auf mobilen Geräten ein ebenso großes Problem wie auf PCs. Durch dynamische Analyse des Verhaltens von Apps wird daher versucht, die Erkennungsraten zu verbessern. Ein vielversprechender Ansatz ist dabei maschinelles Lernen. Dieser Beitrag führt kurz in die relevanten Methoden ein und erläutert ihre Anwendung für die Malware-Erkennung. In unserem Versuchsaufbau werden Android-Apps auf einem Smartphone unter Simulation von Benutzerinteraktionen ausgeführt. Die dabei protokollierten Aufrufe des Linux-Kernels dienen als charakteristisches Merkmal für das Training bzw. die spätere Anwendung der Klassifikatoren. Die Auswertung zeigt, dass bereits mit diesen einfachen Mitteln eine relativ hohe Genauigkeit bei der Klassifikation möglich ist. Bei geeigneter Parameterwahl konnte per Support Vector Machine und Logistischer Regression ein Accuracy Score von über 95% erreicht werden.

1 Einleitung

Der Antiviren-Hersteller GDATA zählte einer aktuellen Untersuchung[1] zufolge im Jahr 2017 über 3 Millionen Android-Schaddateien. Die Zahl liegt nur geringfügig unter dem Vorjahreswert, so dass weiterhin von einer kritischen Bedrohungslage auszugehen ist. Aufgrund der zeitlich begrenzten Unterstützung der Hersteller wird ein signifikanter Anteil von Android-Geräten heute nicht mehr mit Updates versorgt und damit ohne ausreichenden Schutz betrieben.

In Googles *Play Store* sind aktuell 3,66 Millionen Apps verfügbar[2], allein im vierten Quartal 2017 wurden 19 Milliarden Apps heruntergeladen.[3] Im Laufe des Vorjahres wurden 700.000 Apps aus dem Play Store wegen Verstößen gegen Googles Richtlinien entfernt. Bei einem überwiegenden Teil davon dürfte es sich um *Potentially Harmful Applications* (PHA) handeln.[4]

[1] https://www.gdata.de/blog/2018/02/30489-jede-stunde-rund-343-neue-android-schadprogramme-in-2017 (Abruf am 29.03.2018)

[2] https://de.statista.com/statistik/daten/studie/74368/umfrage/anzahl-der-verfuegbaren-apps-im-google-play-store/ (Abruf am 20.06.2018)

[3] https://www.androidpolice.com/2018/01/26/play-store-hit-19-billion-downloads-q4-2017-increasing-lead-apples-app-store-145/ (Abruf am 29.03.2018)

[4] Die genaue Verteilung ist in https://android-developers.googleblog.com/2018/01/how-we-fought-bad-apps-

Bereits bei PCs zeigt sich seit geraumer Zeit, dass eine rein signaturbasierte Erkennung von Schadsoftware an ihre Grenzen stößt. Dies gilt auch für mobile Geräte. Bei Apps wird daher zusätzlich auf eine dynamische Analyse gesetzt [BZNT11, ITBV13] und die Vorteile beider Ansätze in der hybriden Analyse kombiniert – nicht zuletzt auch durch Google selbst im Rahmen der Initiative *Play Protect*.[5]

Künstliche Intelligenz gewinnt mit hoher Geschwindigkeit an Bedeutung. Da Sicherheitsexperten typischerweise noch nicht mit dieser Domäne vertraut sein dürften, gibt die vorliegende Arbeit einen Erfahrungsbericht, wie Methoden des maschinellen Lernens verwendet werden können, um die prinzipielle Funktionsweise praktisch zu demonstrieren. Mit Hilfe von Machine Learning-Algorithmen wird versucht, Schadsoftware unter Android-Apps anhand von beobachtetem Verhalten aufzudecken. Aufbauend auf bestehenden Erkenntnissen wird die Auswirkung von verschiedenen Parametern auf die Erkennungsrate untersucht.

2 Bestehende Ansätze

Unter *statischer Analyse* (engl. static analysis) versteht man den Ansatz, in ausführbarem (oder z.T. auch Quell-)Code nach Anzeichen für verdächtiges Verhalten zu suchen. Dabei wird nach bestimmten Befehlssequenzen, so genannten *Signaturen* gesucht, die zu bekannter Schadsoftware gehören [CJ06]. Dieses Prinzip wurde jahrelang in Antiviren-Software umgesetzt und auch auf den Bereich mobiler Endgeräte übertragen. Autoren von Schadsoftware können jedoch mittels polymorphem Code, der Verschleierung per Obfuskation, der Verschlüsselung sowie dem dynamischen Nachladen von Codeteilen zur Laufzeit statische Analysen relativ leicht umgehen (siehe [CJ06] für eine Beschreibung der einzelnen Techniken).

Dynamische Analyse (engl. dynamic analysis, behavior-based detection) versucht, diese Nachteile zu vermeiden. Dabei wird die zu prüfende Software in einer kontrollierten Umgebung unter Beobachtung ausgeführt. Ziel ist es, Anomalien anhand der zur Laufzeit ausgeführten Funktionen und deren Parametern zu erkennen. Berücksichtigt werden Kernel- und API-Aufrufe sowie aufgebaute Netzwerkverbindungen [ESKK12]. [BZNT11] gibt einen Überblick über frühe Arbeiten zur anomaliebasierten Erkennung von Android-Malware. Eine umfassende Darstellung aktuellerer Methoden inklusive der Diskussion ihrer Wirksamkeit findet sich in [TFA+17]. Canzanese [CJ15, S. 19] gibt eine Zusammenstellung von Ansätzen zur Erkennung von PC-Malware anhand von Systemaufrufen. Eine umfangreiche Studie, die diese Technik nutzt, ist [CMK15].

Bereits [BBS+10] beschreibt eine Sandbox zur Extraktion von statischen und dynamischen Merkmalen aus Android-Apps. Alzayklaee [AYS17] zeigt jedoch verschiedene Techniken, mit denen Anwendungen trotz des Sandbox-Sicherheitsmodells in Android erkennen können, ob sie in einem Emulator ausgeführt werden. Malware kann sich daher im entscheidenden Moment bewusst harmlos verhalten, um der Entdeckung zu entgehen. Zur Vermeidung dieses Nachteils wurde daher für unseren Versuchsaufbau ein physisches Gerät verwendet.

Eine wesentliche Einschränkung des in [AYS16] beschriebenen Werkzeugs besteht darin, dass es nur API-, aber keine nativen Library-Aufrufe (etwa von zur Laufzeit nachgeladenem Code) erkennen kann. Daher werden vorliegend zusätzlich Linux-Kernelaufrufe (Syscalls) betrachtet, da API-Calls durch Android auf unterster Ebene in diese überführt werden. Hierbei wurde nicht nur, wie in [BZNT11, BBS+10], die Häufigkeit einzelner Aufrufe, sondern auch die Häufigkeit

and-malicious.html (Abruf am 29.03.2018) nicht genannt.

[5] https://source.android.com/security/reports/Google_Android_Security_2016_Report_Final.pdf (Abruf am 06.04.2018)

von Sequenzen der Länge 2 und 3 betrachtet. [CMK15] haben bereits gezeigt, dass sich dadurch bessere Erkennungsraten erzielen lassen.

Anstelle des unüberwachten k-means Clustering wie in [BZNT11] werden in diesem Beitrag überwachte Methoden zur Klassifizierung genutzt. Betrachtet werden Logistische Regression und Support Vector Machines. Vorhandene Android-Malware-Datensätze (siehe Abschnitt 4.1) beinhalten die Klassen des überwachten Lernens. Über Parameter lassen sich die Klassifikatoren jeweils auf die vorliegenden Verhaltensdaten einstellen, um so bessere Erkennungsraten zu erzielen. Auf Basis des identischen Datensatzes können die Ergebnisse beider Klassifikatoren miteinander verglichen werden.

3 Methoden des maschinellen Lernens

Maschinelles Lernen (ML) ist ein Teilgebiet der Informatik, das sich mit Systemen befasst, die Wissen selbstständig erlernen und dieses auf neue Probleme anwenden können. Hierbei werden die zugehörigen Algorithmen (Klassifikatoren) in *überwachtes und unüberwachtes Lernen* unterteilt, wobei ein Algorithmus beim unüberwachten Lernen eigene Modelle auf Basis so genannter *Features* bildet. Im vorliegenden Anwendungsfall handelt es sich bei den Features um die mittels Analyse der ausgeführten Apps gewonnenen Daten über Befehlssequenzen, repräsentiert als Feature- und Klasse-Vektoren.

Beim überwachten Lernen werden zusätzlich zu den gegebenen Features Informationen zur jeweiligen Klasse bereitgestellt, hier also die Information, ob eine Anwendung schädlich oder unschädlich ist. Mit diesen Daten *trainiert* der Algorithmus ein entsprechendes Modell und optimiert dessen Parameter, sodass eine möglichst korrekte Klassifizierung erfolgen kann. Ziel ist es, den Zusammenhang zwischen Trainingsdaten und deren *Label* (also die Zuordnung zu einer der beiden Klassen) zu verallgemeinern, sodass ein Klassifikator ebenfalls Aussagen über ihm bisher unbekannte Daten treffen kann (*Prediction*).

Eine Grundannahme unserer Untersuchung besteht darin, dass schädliche Anwendungen typische Verhaltensmuster aufweisen und damit als solche erkannt werden können.

Preprocessing

In der Praxis ist es üblich, die Features in verschiedenen Vorbereitungsschritten (*Preprocessing*) so anzupassen, dass ein Klassifikationsalgorithmus möglichst gute Erkennungsraten erzielt. Typische Schritte umfassen dabei das Entfernen von *statistisch irrelevanten* Features. Zum Beispiel kommt ein `sleep`-Befehl in sämtlichen Aufrufsequenzen vor, so dass er als Unterscheidungsmerkmal ungeeignet ist. Dieses Aussortieren wird in dieser Arbeit zum einen über einen Schwellwert (*Treshold*) als untere Grenze für die Zahl von Aufrufen erreicht (damit werden z.B. Abstürze von Apps berücksichtigt). Zum anderen wird der KBest-Parameter verwendet, der über einen χ^2-Test die Abhängigkeit der Features untereinander prüft und ähnliche aussortiert. Die Angabe erfolgt prozentual zu den aufgetretenen Features. Die Wahl KBest=90% sortiert dabei 10% der Features aus, die sich am ähnlichsten sind.

Skalierung

Ein weiterer Vorbereitungsschritt ist die *Skalierung* der Feature-Räume. Die verwendeten Implementierungen der ML- und Preprocessing-Algorithmen stammen aus der *scikit*-Bibliothek.[6] Exemplarisch wurden hierfür folgende Skalierungsmethoden verglichen: Der *StandardScaler* bereitet die Daten so auf, dass sie ähnlich der Standard-Normalverteilung mit einem Mittelwert

[6] http://scikit-learn.org/ (Abruf am 06.04.2018)

von 0 und einer Varianz von 1 vorliegen. Der *MinMaxScaler* skaliert die Features linear auf einen Wertebereich von $[0; 1]$ um.

Klassifikatoren

Für Klassifikatoren existieren eine Vielzahl an Modellen und Algorithmen. Gemeinsam haben die Modelle, dass sie das beobachtete Verhalten abbilden und während des Lernvorgangs ihre Parameter optimieren (*Training*). Nach diesem Vorgang sollte idealerweise eine klare Trennung der Klassen auf Basis der Trainingsdaten vorliegen. Somit kann untrainiertes Verhalten ebenfalls in das Modell überführt werden und eine Klassifizierung stattfinden. Je besser die Trainingsdaten das der Klasse zugehörige Verhalten abbilden, desto höher ist die Erkennungsrate.

Für diesen Beitrag wurden als Modelle *Logistische Regression* (LR) und *Support Vector Machine* (SVM, Stützvektormethode) betrachtet, wobei letztere nachfolgend kurz erläutert wird. Detaillierte Beschreibungen der Ansätze finden sich in der einschlägigen Literatur [BH07, RN16].

Die Grundidee bei SVM besteht darin, die Objekte, die durch Feature-Vektoren repräsentiert sind, entsprechend ihres Labels möglichst gut durch eine Hyperebene in zwei Klassen zu trennen. Der *Normalen-* oder *Stützvektor* der Hyperebene ist die Lösung des SVM-Algorithmus. Ziel der Trainingsphase ist es, eine Hyperebene zu finden, die einen maximalen Abstand zu den am nächsten liegenden Objekten aufweist. Der kleinste Abstand zwischen zwei Objekten, die zu unterschiedlichen Klassen gehören, wird als Margin bezeichnet (dargestellt durch die gestrichelte Linie zwischen einem hellen und einem dunklen Punkt in Abbildung 1).

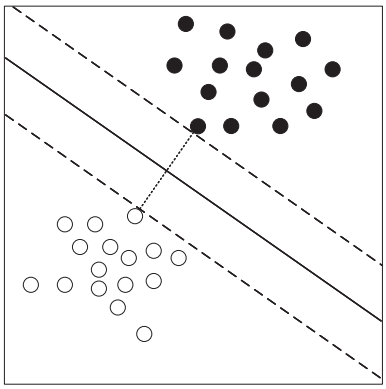


Abb. 1: Lineare Trennung von Objekten mittels SVM

Über den Parameter C wird gesteuert, inwieweit im Training Fehlklassifizierungen zugelassen werden, um den Effekt von Ausreißern, die nicht der tatsächlichen statistischen Verteilung entsprechen, abzumildern. Ist schließlich ein solches Modell bestimmt, können Vorhersagen (*Predictions*) getroffen werden. Der Abstand eines zu klassifizierenden Objekts zur Hyperebene ist gleichzeitig ein Maß für die Zuverlässigkeit der Zuordnung.

Auch nicht linear trennbare Objekte können per SVM klassifiziert werden, indem sie zunächst mit Hilfe von nichtlinearen Funktionen in höher dimensionale Vektorräume transformiert werden. Neben der linearen SVM werden dabei die auf einer solchen Transformation beruhenden Varianten *sigmoid*, *rbf* (*radial basis function*) und *poly* unterschieden.

Gütemaß

Zur Bestimmung der Güte von Vorhersagen werden folgende Größen betrachtet:

TP: Software ist schädlich und wurde auch so klassifiziert (*True Positive*)

TN: Software ist unschädlich und wurde auch so klassifiziert (*True Negative*)

FN: Software ist schädlich, wurde jedoch als unschädlich klassifiziert (*False Negative*)

FP: Software ist unschädlich, wurde jedoch als schädlich klassifiziert (*False Positive*)

Damit lassen sich verschiedene Vergleichsmetriken definieren (siehe etwa [BH07]). Für diesen Beitrag wurde der Wert

$$\text{Accuracy} := \frac{(TP + TN)}{(TP + TN + FP + FN)}$$

gewählt, welcher den Anteil korrekter Klassifizierung misst und im Idealfall 1 beträgt.

Um eine allgemeingültige Aussage über die Eignung des Modells zu treffen, wird der Ausgangsdatensatz in Trainings- und Testdaten aufgeteilt und im Anschluss der Accuracy-Score, basierend auf den Testdaten, ermittelt. Dieser Vorgang wird bei einer *Cross-Validation* (CV) mehrfach mit verschieden Partitionen der Ausgangsdaten (*Splits*) wiederholt. Um den Score nicht zu verfälschen, wurden die Samples pro App gruppiert, sodass nicht Teile einer bereits zu testenden App trainiert werden. Im Anschluss wurde der Mittelwert der Scores gebildet, um eine Gesamtbewertung des Modells zu erhalten.

4 Lösungsarchitektur und Vorgehensweise

4.1 Vorbereitungen und Testdaten

Für die Untersuchungen wurde ein Smartphone vom Typ Galaxy S3 neo in Kombination mit der Firmware *CyanogenMod 12.1* verwendet.[7] Dies erlaubt es, die Android Debug Bridge[8] unter Root-Rechten auszuführen, was u.a. deshalb notwendig ist, um später uneingeschränkten Zugriff auf die zu analysierenden Prozesse zu erhalten. Ferner wurde das GoogleApps-Paket[9] eingespielt und zudem Funktionen deaktiviert, die es normalerweise verhindern, dass von Google bereits erkannte Schadsoftware installiert werden kann.

Zum Erstellen eines vollständigen Systemabbilds (*Clean Image*) des vorkonfigurierten Geräts wurde das Werkzeug *TWRP*[10] verwendet. Damit Messungen nicht durch eventuelle Rückstände vorher ausgeführter Software verfälscht werden, wurde jeweils zunächst ein Recovery des Clean Images durchgeführt, um wieder einen definierten neutralen Ausgangszustand herzustellen.

Zum Training der Klassifikatoren wurden Kerneltraces von insgesamt 50 verschiedenen Apps verwendet. Davon waren jeweils 25 als PHA bzw. unschädlich eingestuft. Die Schadsoftware stammt aus gelabelten Datensätzen von *Drebin* [ASH+14] und *Virusshare*[11], wovon jeweils fünf einer gemeinsamen Schadklasse angehören.[12] Die konkret gewählten schädlichen Anwendungen enthielten u.a. Funktionen für Privilege Escalation, Remote Control und dem Ausspähen persönlicher Informationen.

[7] Vom Nachfolgeprojekt *LineageOS* standen zum Untersuchungszeitpunkt nur inoffizielle und unvollständige OS-Versionen zur Verfügung.

[8] eine Software-Schnittstelle, über die per USB mit dem Gerät kommuniziert werden kann

[9] http://opengapps.org/ (Abruf am 08.04.2018), enthält die Standard-Android Apps wie Kalender, Play Store etc.

[10] https://twrp.me/ (Abruf am 08.04.2018)

[11] https://virusshare.com/ (Abruf am 30.03.2018)

[12] Für eine Typisierung mobiler Schadsoftware auf Basis der Infektionswege, Aktivierungsmechanismen und Payloads wird auf [ZJ12] verwiesen.

Die unschädlichen Apps waren zufällig ausgewählte Anwendungen des *Google Play Stores*. Es wird dabei angenommen, dass diese durch Google Play Protect überprüft wurden und tatsächlich unschädlich sind.

4.2 Datenerfassung und Preprocessing

Die Steuerung des Erfassungsprozesses erfolgte von einem mit dem Smartphone verbundenen PC aus. Die zugehörige Anwendung wurde in Python entwickelt, da sich diese Sprache gut als Bindeglied zwischen den unterschiedlichen Komponenten eignet. Durch Python-Skripte wurden das Tracing der Kernelaufrufe per Linux-Kommando `strace` angestoßen, Apps gestartet und die Simulation zufälliger Benutzereingaben initiiert. Für letztere Aufgabe wurde das Test-Werkzeug *UI/Application Exerciser Monkey*[13] verwendet. Monkey ist in der Lage, zufällige Benutzereingaben zu erzeugen, diese an einen Emulator oder ein angeschlossenes Gerät zu senden und das Systemverhalten auszuwerten. Ausnahmen oder Abstürze einer App werden dabei erkannt.

Tab. 1: Befehlsargumente der Traceanwendung

Argument	Beschreibung	Beispiel
-a <Paketname>	Paketname der Anwendung	“com.pcgprh.dfmdomom“
-l <Tracelogpfad>	Ausgabepfad der Tracefiles	“TRACELOGS“
-r <Runden>	Anzahl der Durchgänge	“30“
-t <Zeit>	Dauer pro Durchgang in Sekunden	“30“
-m <Monkeybefehl>	Parameter für Monkey	“–throttle 100 –pct-syskeys 0 800“
[-s <Schadklasse>]	Name der Schadklasse	“Android.Riskware.Agent.gHHUC“

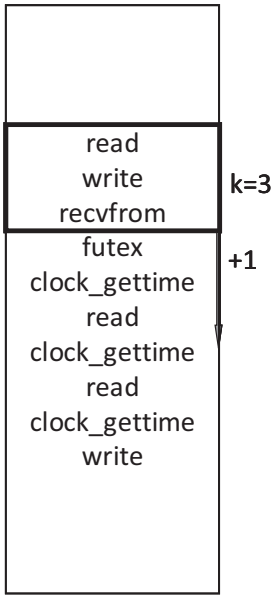
Tabelle 1 zeigt die Syntax eines Aufrufes mit einem Beispiel, in welchem Monkey insgesamt 800 Befehle jeweils im Abstand einer Zehntelsekunde ausführt, wobei Systemkommandos (wie z.B. der Home-Button) ausgenommen sind, da sie die App unterbrechen würden. Zur Vergleichbarkeit wurden für jede App zweimal 30 Durchgänge mit einer Dauer von je 30 Sekunden durchlaufen. Während eines Durchgangs wurden die Linux-Kernelaufrufe mitprotokolliert und zur späteren Weiterverarbeitung gespeichert. Im Mittel wurden für die Apps ca. 33.500 Aufrufe pro Durchgang registriert, wobei die Standardabweichung mit 19.600 sehr hoch ist. Nach verschiedenen Preprocessing-Schritten (siehe unten) wurde der Datensatz dem Klassifikator zum Training übergeben.

Am PC erfolgte die Weiterverarbeitung der gewonnenen Daten mittels *scikit*. Mit Hilfe eines *Sliding Window*-Algorithmus wird hierfür das Datenmodell mit den Sequenzlängen 1 bis 3 erzeugt. Für alle im Trace vorkommenden Sequenzen wird deren Häufigkeit gezählt, um den Feature-Vektor zu bilden. Abbildung 2 verdeutlicht diesen Prozess: Links sind beispielhaft die Aufrufe zu sehen, die zu den rechts angegebenen Häufigkeitsverteilungen der auftretenden Sequenzen verschiedener Länge führen. Für jede Sequenzlänge lagen insgesamt je 1.500 Vektoren schädlicher sowie unschädlicher Apps vor.

4.3 Algorithmen- und Parameterwahl

Freiheitsgrade hinsichtlich der Wahl der verwendeten Algorithmen und Parameter bestehen an zahlreichen Stellen. Zur Vereinfachung wurde die Heuristik verfolgt, sukzessive das jeweilige lokale Optimum für jeden Freiheitsgrad zu ermitteln, wobei für die noch nicht betrachteten Freiheitsgrade zunächst Standardwerte vorbelegt wurden. Um das bestmögliche Ergebnis zu

[13] https://developer.android.com/studio/test/monkey (Abruf am 17.06.2018)



k=1

Syscall-Sequenz	Anzahl
(read)	3
(write)	2
(recvfrom)	1
(futex)	1
(clock_gettime)	3

k=2

Syscall-Sequenz	Anzahl
(read, write)	1
(write, recvfrom)	1
(recvfrom, futex)	1
(futex, clock_gettime)	1
(clock_gettime, read)	2
(read, clock_gettime)	2
(clock_gettime, write)	1

k=3

Syscall-Sequenz	Anzahl
(read, write, recvfrom)	1
(write, recvfrom, futex)	1
(recvfrom, futex, clock_gettime)	1
(futex, clock_gettime, read)	1
(clock_gettime, read, clock_gettime)	2
(read, clock_gettime, read)	1
(read, clock_gettime, write)	1

Abb. 2: Sliding Window Algorithmus

erzielen, wäre an dieser Stelle ein aufwändiger *GridSearch*-Algorithmus durchzuführen. Dieser würde den Accuracy Score für jede mögliche Parameterkombination (siehe Tabelle 2) ermitteln und so das globale Optimum bestimmen.

Während der Preprocessing-Phase wurden (in dieser Reihenfolge) der Schwellwert, die Auswahl der Skalierungsfunktion sowie der KBest-Parameter festgelegt. Bei der eigentlichen Klassifikation wurde der *Regularization Parameter*, der eine mögliche Überanpassung auf Trainingsdaten (Overfitting) verhindern soll, sowie der Klassifikator (jeweils noch mit verschiedenen Varianten) variiert. Tabelle 2 gibt eine Übersicht über das Vorgehen.

Tab. 2: Analysierte Parameter

Phase	Parameter	Wertemenge
Modellierung	Sequenzlänge k	1 | 2 | 3
Preprocessing	Threshold	0 | 100 | 500 | 1000 | 2000 | 3000 | 5000
	Scaler	MinMaxScaler | StandardScaler
	KBest	100% | 95% | 90% | 80% | 70% | 60%
Klassifikation	Regularization C	0.001 | 0.01 | 0.1 | 1 | 10 | 50 | 100
	Klassifikator	LR Solver: lbfgs | newton-cg | liblinear | sag | saga
		SVM Kernel: clinear | sigmoid | rbf | poly

Für jeden Preprocessing- und Klassifikations-Parameter wurden jeweils die CV-Accuracy-Scores für alle in der Tabelle angegebenen Elemente der Wertemenge ermittelt. Diejenige Parameter-Wahl, die jeweils den besten Score erzielte, wurde für die Bestimmung des nächsten Parameters vorbelegt. Einzelne Ergebnisse dieser Parameterbestimmung sind nachfolgend beschrieben.

5 Auswertung

Tabelle 3 zeigt exemplarisch die maximierten Scores unter der Parametervariation von KBest. Ziel ist es, einen Wert zu finden, der die Klassifizierung verbessert und ggf. die Lerndauer beschleunigt. Eine reduzierte Feature-Menge hat den Vorteil, dass der Lernalgorithmus weniger Dimensionen berücksichtigen muss und damit schneller wird. Das Ergebnis zeigt, dass eine Reduktion möglich ist, ohne das Ergebnis zu verschlechtern und es sogar minimal verbessert. Der SVM-Klassifikator erzielte in diesem Fall bessere Ergebnisse, wenn zuvor eine Bereinigung von 40% der gesamten Feature-Menge durchgeführt wurde. Der LR-Klassifikator dagegen zeigt nur Verbesserungen für reduzierte Feature-Mengen auf Basis von Zweier- und Dreiersequenzen.

Tab. 3: Accuracy für die jeweils beste KBest-Variation

Länge k	Klassifikator	KBest	∅Accuracy	Verbesserung[14]
1	SVM	60%	0,9214	0,0026
2	SVM	60%	0,9358	0,0045
3	SVM	60%	0,9080	0,0089
1	LR	95%	0,9143	0,0000
2	LR	70%	0,9419	0,0024
3	LR	90%	0,9445	0,0181

Die Untersuchungen zeigen, dass die durchschnittliche Erkennungsrate nahezu immer über 90% liegt. Es bestätigt sich die Annahme, dass die erfassten Features für die Erkennung von Schadsoftware gut geeignet sind.

Abbildung 3 zeigt nun den erzielten Score und die Lerndauer des LR-Klassifikators in Abhängigkeit vom Lernalgorithmus (Solver), dem Regularization-Parameter sowie der Sequenzlänge. Die Messungen ergaben, dass sich die Parameter deutlich auf den Score und die Trainingsdauer auswirken. Am stärksten ist der Einfluss der Sequenzlänge auf die Dimension des Feature-Raums und damit den Trainingsaufwand (erkennbar in Abbildung 3, untere Darstellungen). In dieser Konstellation wurde der beste Accuracy-Score mit einem Wert von über $0{,}95$ für den lbfgs-Algorithmus mit schwacher Regularisierung ($C = 10$) erzielt. Tabelle 4 enthält die als optimal bestimmten Parameterkombinationen für LR bzw. SVM.

Tab. 4: Beste Parameter mit Accuracy Score und Lerndauer in Abhängigkeit von Sequenzlänge

k	Klassifikator	C	Threshold	Scaler	KBest	Accuracy	Lerndauer
1	LR mit lbfgs	100	2000	MinMaxScaler	100%	0,9193	0,64s
2	LR mit lbfgs	1	500	MinMaxScaler	70%	0,9521	3,04s
3	LR mit lbfgs	10	1000	StandardScaler	90%	0,9555	23,76s
1	SVM mit rbf	10	0	MinMaxScaler	60%	0,9297	0,59s
2	SVM mit sigmoid	50	0	MinMaxScaler	60%	0,9536	3,90s
3	SVM mit rbf	50	0	MinMaxScaler	60%	0,9458	20,47s

Der SVM-Klassifikator mit linearem Kernel weist die schnellste Lerndauer auf, wohingegen sich der Poly-Kernel für dieses Problem nicht eignet. Der LR-Klassifikator erzielte schnelle Lernzeiten mittels liblin-Solver, entsprechend ungeeignet ist hierbei der saga-Solver. Ein insgesamt gutes Tradeoff zwischen Einlerndauer und Score bietet hierbei der lbfgs-Solver des LR-Modells.

[14] Im Vergleich zu KBest=100%.

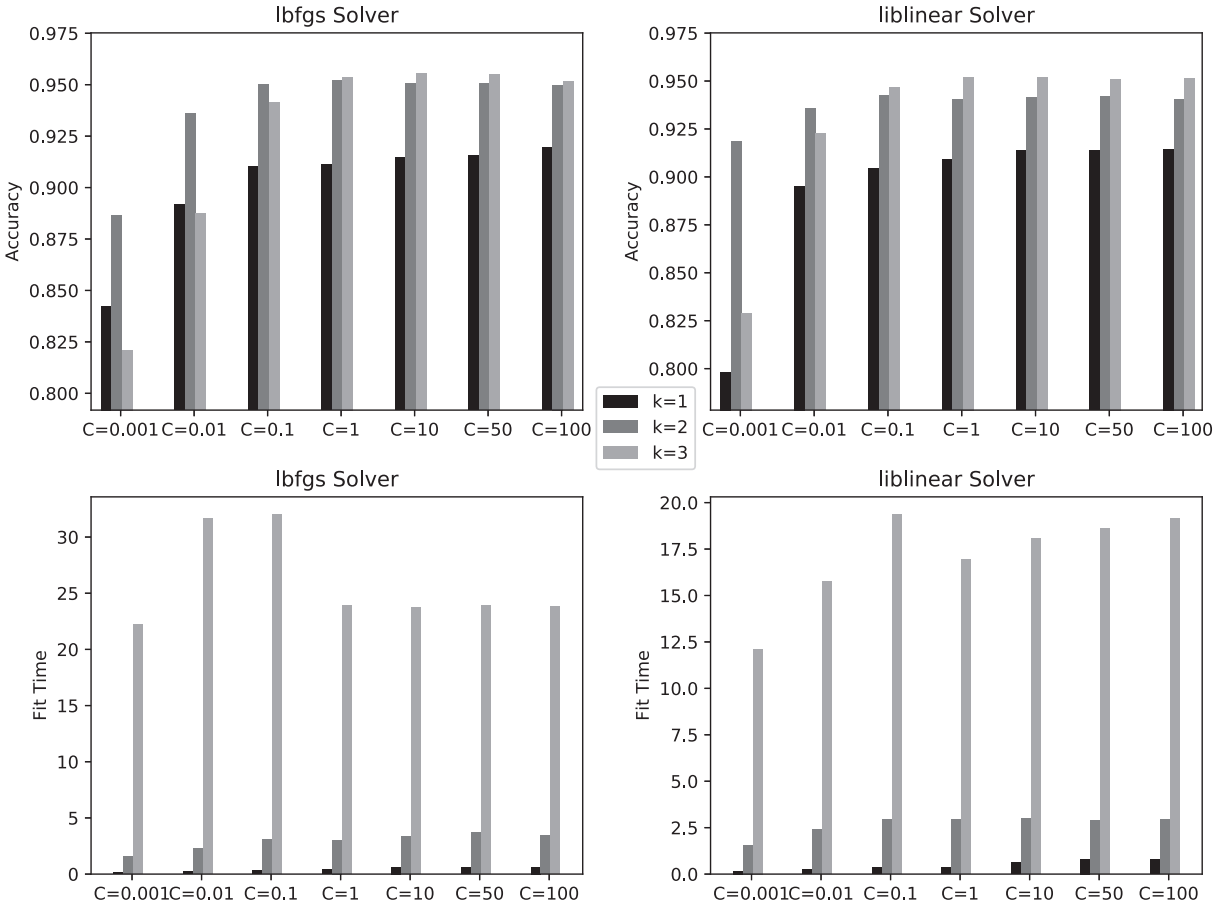


Abb. 3: Logistische Regression: Accuracy Score und Lerndauer in Abhängigkeit von Solver, C und k

6 Fazit und Ausblick

Im Beitrag wurde gezeigt, dass eine Klassifikation von Android-Apps auf Basis ihres Verhaltens machbar ist, auch wenn dafür lediglich Aufrufe auf niedrigerer Betriebssystemebene ausgewertet werden. Zusammenfassend kann festgehalten werden, dass sich für die vorliegenden Trainings- und Testdaten sowohl LR- als auch SVM-Klassifikator mit durchschnittlichen Accuracy Scores von über $0{,}90$ gut für die Klassifizierung von Android-Apps auf Basis dynamischer Features eignen. Datenmodelle der Sequenzlängen 2 und 3 erzielten hierbei bessere Ergebnisse als das einfache Modell mit den Häufigkeiten einzelner Kernel-Aufrufe. Dabei ist jedoch zu beachten, dass die vorliegenden Erkenntnisse auf einer relativ kleinen Sample-Größe beruhen.

Zur Verarbeitung größerer Datensätze mit einer deutlich höheren Anzahl an Apps müsste der Ansatz skalierbar gemacht werden. Hierfür kommen etwa Techniken wie inkrementelles Lernen oder weitere Preprocessing-Schritte in Frage. Auch Hardware mit mehr CPU- und Hauptspeicher-Ressourcen sowie die Einbindung von GPUs könnten die Algorithmen beschleunigen. Dies würde eine Überprüfung ermöglichen, ob sich die hier als gut erwiesenen Parameter auch für größere Datenmengen bewähren. Ein trainiertes Modell könnte so zusammen mit anderen Merkmalen aus der statischen Analyse (welche in diesem Beitrag nicht berücksichtigt wurde) in ein Anti-Malwaresystem überführt werden.

Um während der Datenerfassung die Benutzerinteraktion mit der App zu simulieren, wurden

mittels Monkey künstlich Eingaben in zufälliger Weise erzeugt. Um eine realistischere Interaktion simulieren zu können, wäre es stattdessen denkbar, anonymisierte Echtdaten von Nutzern heranzuziehen, so dass mit der Zeit eine Abdeckung sämtlicher Programmzustände möglich wird. Für die Extraktion solcher Daten müsste es aber Schnittstellen geben. Momentan sind diese aufgrund der Sicherheitsarchitektur nur dem Android-System selbst zugänglich.

Literatur

[ASH+14] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, CERT Siemens: Drebin: Effective and explainable detection of android malware in your pocket. In *Network and Distributed System Security (NDSS)*, volume 14, pages 23–26, 2014.

[AYS16] M.K. Alzaylaee, S.Y. Yerima, S. Sezer: Dynalog: An automated dynamic analysis framework for characterizing android applications. In *Cyber Security And Protection Of Digital Services (Cyber Security), 2016 International Conference On*, pages 1–8. IEEE, 2016.

[AYS17] M.K. Alzaylaee, S.Y. Yerima, S. Sezer: Emulator vs real phone: Android malware detection using machine learning. In *Proceedings of the 3rd ACM on International Workshop on Security And Privacy Analytics*, IWSPA '17, pages 65–72, New York, NY, USA, 2017. ACM.

[BBS+10] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Camtepe, S. Albayrak: An android application sandbox system for suspicious software detection. In *Malicious and unwanted software (MALWARE), 2010 5th international conference on*, pages 55–62. IEEE, 2010.

[BH07] M. Berthold, D.J. Hand: *Intelligent data analysis: an introduction*. Springer, Berlin, 2. edition, 2007.

[BZNT11] I. Burguera, U. Zurutuza, S. Nadjm-Tehrani: Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[CJ06] M. Christodorescu, S. Jha: Static analysis of executables to detect malicious patterns. Technical report, University of Wisconsin-Madison, Dept. of Computer Sciences, 2006.

[CJ15] R.J. Canzanese Jr: *Detection and classification of malicious processes using system call analysis*. Drexel University, 2015.

[CMK15] R. Canzanese, S. Mancoridis, M. Kam: Run-time classification of malicious processes using system call analysis. In *Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on*, pages 21–28. IEEE, 2015.

[ESKK12] M. Egele, T. Scholte, E. Kirda, C. Kruegel: A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*, 44(2):6, 2012.

[ITBV13] R. Islam, R. Tian, L.M. Batten, S. Versteeg: Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications*, 36(2):646–656, 2013.

[RN16] S. Russell, P. Norvig: *Artificial intelligence: a modern approach*. Pearson, Boston, 3. edition, 2016.

[TFA+17] K. Tam, A. Feizollah, N.B. Anuar, R. Salleh, L. Cavallaro: The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4):76, 2017.

[ZJ12] Y. Zhou and X. Jiang: Dissecting android malware: characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.