4. November 2018 Studie · Mobile Sicherheit · 34 Min. Lesezeit

Ähnlichkeit von Quellcode in Anwendungen

Archiviert. Dieser Inhalt wird nicht mehr aktualisiert. Er bleibt zur Information verfügbar.

Stand 2018-10-23
Quelle https://technology2.a-sit.at/veroeffentlichungen/aehnlichkeit-von-quellcode-in-anwendungen/

Zur Gewinnung eines abstrakteres Verständnisses vom Verhalten von Progrmamcode, werden in diesem Projekt unter Anwendung aktueller wissenschaftlicher Ansätze Muster in Code identifiziert und klassifiziert.

Für die sicherheitsorientierte Analyse von Applikationen ist es seit jeher essentiell, sämtliche in den Programmen vorkommenden Komponenten und ihre designierte Aufgabe zu identifizieren. Mit dem dadurch gewonnen Verständnis über die Funktionsweise lässt sich schließlich erheben, ob Anwendungen gegen etablierte Sicherheitsprinzipien verstoßen bzw. davon abgesehen überhaupt die Funktionalität umsetzen, die zu erwarten wäre. Die in den letzten Jahren stark zugenommene Komplexität und der Umfang einzelner Anwendungen führt jedoch dazu, dass Abläufe und Zusammenhänge bei Code-Reviews nur mehr schwer nachvollziehbar sind.

Das Ziel des Projekts bestand darin, durch die Anwendung aktueller wissenschaftlicher Ansätze Muster in Code zu identifizieren, um darauf aufbauend, ein abstrakteres Verständnis über das Verhalten von Quellcode abzuleiten. Hierfür essentiell ist es, geeignete Methoden zu finden, um auch sehr umfangreiche Mengen an Code so zu verarbeiten, dass ein semantisches Verständnis und eine Klassifikation von Mustern möglich sind. Die aus diesem Projekt gewonnenen Erkenntnisse sollen eine Lösung aufzeigen, um semantisch ähnliche Codefragmente identifizieren zu können.

1. Einleitung

Bei einer Programmanalyse nach sicherheitsorientierten Gesichtspunkten wird das Verhalten und die Implementierung von Anwendungen den Anforderungen in einem sicherheitsaffinen Verwendungsumfeld gegenübergestellt. Die Beurteilung, ob gewisse Teile eines Programms problematisch bzw. „unsicher" sind, basiert für gewöhnlich auf Prämissen, die den Rahmen der Untersuchung abstecken. Das bedeutet im Umkehrschluss, dass eine zuverlässige Bewertung nur für jene Aspekte erfolgen kann, nach denen explizit gesucht wurde. In der Praxis zeigt sich dieser Umstand sehr eindringlich in der Funktionsweise von Anti-Viren-Lösungen, bei denen Programmteile klassischerweise manuell überprüft und Code mithilfe manuell

erstellter Signaturen („Hashes“) erkannt wird. Die Identifikation von Codefragmenten, die als schädlich betrachtet werden, findet dabei typischerweise auf der Binärebene statt. Sobald Schadcode mutiert, führt eine monolithische Prüfung jedoch nicht mehr zum gewünschten Resultat und es muss auf eine heuristische Verhaltensanalyse ausgewichen werden.

Dass die manuelle Sichtprüfung nicht mehr ausreicht, zeigt sich vor allem in der Dynamik, die sich bei Anwendungen für Mobilbetriebssysteme wie Android oder iOS beobachten lässt. Zunehmende Rechenleistung und stark gestiegenes Speichervermögen von Mobilgeräten wie Smartphones und Tablets, führen dazu, dass Anwendungen für die Mobilbetriebssysteme Android und iOS nicht nur ressourcenhungriger werden, sondern auch die Komplexität bei der Entwicklung und Wartung von Apps drastisch zunimmt.

Zur Vereinfachung dieses Prozesses und um Funktionalität zu kapseln, greifen Android- und iOS-Anwendungen typischerweise auf Programmbibliotheken von Drittherstellern („Third-Party-Libraries“) zurück. Die Hauptanwendung kann diese Bibliotheken anhand der verfügbaren Schnittstellen aufrufen und nahtlos einsetzen, um beispielsweise Social Media-, Werbung- oder „Single Sign On“-Funktionen zu ergänzen. Das Prüfen einer Implementierung auf ihre Sicherheitsaspekte hin, wird dadurch jedoch signifikant erschwert, da Programmabläufe zunehmend weit verschachtelt und nicht mehr vollumfänglich nachvollziehbar sind. Eingebundene Bibliotheken unterscheiden sich zudem technisch nicht vom Code der Hauptanwendung und sind daher nicht als solche erkennbar. Der Ansatz von statischem „Code Fingerprinting“ zur Identifikation von Programmbibliotheken schafft in gewissem Maße Abhilfe, ist aber mit ähnlichen Einschränkungen verbunden, wie die eingangs erwähnte manuelle Erstellung von Signaturen für Programmcode. In der Praxis heißt das, dass nur Libraries gesucht und gefunden werden, bei denen die hinterlegte Signatur übereinstimmt. Wird jedoch z.B. eine neuere Version derselben Programmbibliothek eingesetzt, für die es noch keine Signatur gibt, würde sie nicht gefunden werden.

Anwendungen und Programmbibliotheken für Android und iOS ist eigen, dass sie vergleichsweise oft ein Update erhalten und die dabei vorgenommenen Änderungen der zugrundeliegenden Implementierungen zumeist nur klein sind. Der weit verbreitete Einsatz von Techniken zur Code-Verschleierung („Obfuscation“) verhindert jedoch, dass sich die Unterschiede einzelner Versionen mit nur geringem Aufwand feststellen lassen. Es lässt sich somit nicht ohne Weiteres herausfinden, ob Code hinzugefügt wurde, der im sicherheitskritischen Kontext problematisch ist oder ob sich plötzlich Schadcode in einer Anwendung befindet. Durch Code-Obfuscation werden sämtliche nicht unbedingt notwendigen Debug-Symbole und Identifier aus dem Code entfernt oder umbenannt. Mithilfe von ProGuard [1] etwa werden die Namen von Methoden, Klassen und Packages auf abgekürzte oder nichtssagende Bezeichner geändert. In der Praxis wird so z.B. aus dem Package-Bezeichner „ *com.google* “ die Kurzfassung „ *c.a* “. Analog dazu werden auch „Shrinking“ oder „Optimizations“ verbreitet eingesetzt. Hierbei werden nicht genutzte Codeteile beim Kompilieren entfernt und Codefragmente auf effiziente Ausführung hin optimiert. Für die Identifikation von Codefragmenten ist der Vorgang hinderlich, da dadurch relevante Indikatoren verloren gehen.

Zusammenfassend lässt sich festhalten, dass die manuelle und exakte Identifikation von Codefragmenten in Anbetracht der benötigten Ressourcen und der inhärenten Unvollständigkeit zunehmend an ihre Grenzen stößt. Um diesen Umständen zu begegnen, wurden in diesem Projekt 1 https://www.guardsquare.com/en/proguard Ansätze gesucht, um die **Ähnlichkeit von Codefragmenten** anhand ihrer semantischen Zusammensetzung festzustellen und eine abstraktere Ebene für Code-Klassifikation zu schaffen.

Angesichts der eingangs beschriebenen Rahmenbedingungen und Besonderheiten bei Mobilplattformen werden die in den weiteren Abschnitten erarbeiteten Techniken auf Android-Anwendungen angewandt. Im Gegensatz zu Desktopanwendungen ist es bei Android möglich, aus einer gegebenen Anwendung eine approximative Fassung des originalen Quellcodes zu extrahieren. Dies ermöglicht eine experimentelle Analyse mit realen Anwendungen, wobei automatisch gefundene Ergebnisse mithilfe des vorliegenden Quelltextes verifiziert werden können.

1.1. Problemstellung

Die diesem Projekt zugrundeliegende Problemstellung lässt sich in zwei wesentliche Aspekte unterteilen: 1) Die Extraktion von Codefragmenten, die für eine Charakterisierung im jeweiligen Kontext relevant sind und 2) die darauffolgende automatisierte Verarbeitung, aus der ein generalisiertes Verständnis über die Funktionalität resultieren soll. In den nachfolgenden beiden Abschnitten wird auf diese beiden Bereiche detaillierter eingegangen, um technisch zu eruieren, welche Anforderungen es an eine Lösung gibt und welche Merkmale beachtet werden müssen.

1.1.1. Semantische Ähnlichkeit von Code-Fragmenten

Die manuelle Suche und Identifikation problematischer Codefragmente ist angesichts des Umfangs mobiler Anwendungen kaum mehr praktikabel. Eine Möglichkeit, die folglich in der Praxis zur Anwendung kommt, ist die zielgerichtete Suche nach „Messpunkten" bzw. singulären Aspekten, die ein gewisses Codefragment charakterisieren. Wenn also ein Codefragment dahingehend geprüft werden soll, ob es schadhaft ist, wird nach einem oder mehreren Kennzeichen („Marker") gesucht, die die „Schadhaftigkeit" möglichst zweifelsfrei belegen.

Versucht beispielsweise eine Anwendung erhöhte Berechtigungen durch Aufruf der Datei „su" zu bekommen, wäre ein Messpunkt der Aufruf der Datei im Dateisystem und ein zweiter die Bezeichnung „su". Die manuelle Auswahl dieser „Marker" bringt jedoch die im vorigen Abschnitt erwähnten Konsequenzen mit sich. Die Definition ist inhärent unvollständig, da Systeme verschiedenste API-Methoden zur Verfügung stellen, um Dateien im Dateisystem aufzurufen. Heißt das entsprechende Programm zudem anders als „su", wäre ein weiterer Messpunkt festzulegen.

Diese Beschränkung auf einzelne Aspekte, wie spezifische Keywords oder gewisse Methodenaufrufe ist grundsätzlich ein legitimer Ansatz, der jedoch darunter leidet, dass keine *kontextuelle* Betrachtung von Codefragmenten bzw. einzelnen Instruktionen erfolgt. Es wird nur ermöglicht, Aussagen über einzelne Eigenschaften in Anwendungen zu treffen. Eine Projektion der Erkenntnisse auf das Verhalten der gesamten Anwendung ist heutzutage gängig; folglich von einer kompletten Anwendungsanalyse zu sprechen, wäre aber nur bedingt korrekt. Die fehlende Einbeziehung des Kontextes führt in der Praxis dazu, dass bei einer Anwendungsanalyse zwar auf gewisse Merkmale hin überprüft wird, dies jedoch keine Indikationen darüber zulässt, welche Funktionalität eine Anwendung *eigentlich implementiert* und in *welchem Kontext* gewisse Instruktionen verwendet werden. Inkludiert eine Anwendung beispielsweise Code um GPS-Positionsdaten an einen Webdienst zu senden, kann es sich dabei je nachdem um gewünschte (Navigation, Radarwarner) oder ungewünschte (Malware, Spionage) Funktionalität handeln. Analog dazu gibt es keine Möglichkeit herauszufinden, welche Funktionalität in einer Anwendung enthalten sollte, um den Charakter einer für den jeweiligen Bereich „typischen" Anwendung zu erfüllen. Eine Anwendung zur sicheren Ablegung von Passwörtern ist heutzutage z.B. in praktisch allen Fällen so

implementiert, dass beim Öffnen der Anwendung zunächst nach einem Passwort gefragt wird, das Zugang zu weiteren Passwörtern ermöglicht. Dahinter steht der Einsatz einer Funktion zur Schlüsselableitung („Key Derivation Function“). Ist diese absent, müsste entweder ein gleichwertiger oder besserer Ersatz implementiert sein oder es liegt ein Sicherheitsproblem nahe.

Die technische Herausforderung bei der Extraktion von Codefragmenten aus Anwendungen ist ebenfalls verbunden mit der zuvor bereits erwähnten „Obfuscation“-Thematik. Ein aus einem Codefragment abgeleitetes „Muster“ müsste so generisch sein, dass es unabhängig von „Obfuscation“ oder einer Neuerstellung der Anwendung die gleiche semantische Aussage beinhalten kann.

Praktisch bedeutet das, dass z.B. sämtliche durch „Obfuscation“ oder Neukompilieren der Anwendung änderbaren Eigenschaften nicht Bestandteil des extrahierten Musters sein können. Konkret betrifft dies z.B. Bezeichner von Variablen, Registern, Parametern, Objektbeschreibungen. Welche Auswirkungen aus diesen Eigenschaften tatsächlich entstehen, wird im nachfolgenden Verlauf dieses Dokuments anhand mehrerer Evaluierungsszenarien aufgezeigt.

Eine weitere Problemstellung, die einer fundierten Überlegung bedarf, ist die Notwendigkeit, extrahierte Codefragmente miteinander vergleichen zu können. Liegen beispielsweise mehrere „Muster“ vor, die für Malware typisch sind, muss eine gewisse Metrik darauf angewandt werden können, um die semantische Nähe zueinander bewerten zu können. Ein naiver Ansatz wäre z.B. die Repräsentation von Instruktionen als Vektoren bzw. deren Gesamtheit in einer Methode als Analogie auf anderer Ebene. Die Transformation von Instruktionen zu Vektoren kann beispielsweise über „One-Hot-Encoding“ erfolgen, wo jedem Wort ein Vektor zugewiesen wird. Die einfache Subtraktion der Vektoren zeigt folglich an, wie ähnlich die Vektoren zueinander sind. Die Schwächen und Grenzen dieses aus dem NLP-Bereich stammenden Ansatzes zeigen sich jedoch schnell an den Charakteristika von Programmcode:

1. Würde jede Instruktion individuell als Vektor dargestellt, würde die Gesamtheit der Vektoren einer Methode eine vergleichsweise dünne Matrix sein (*sparse matrix)*, da bei „One-Hot-Encoding“ die Vektoren so groß sind, wie das maximale Set an Instruktionen und die Wahrscheinlichkeit gering ist, dass die gleichen Zusammenhänge in einem Codefragment öfter auftreten.
2. Die meisten Instruktionen in Anwendungen können durch andere substituiert werden, um die gleiche Funktionalität zu erhalten. Ähnlich wie bei natürlicher Sprache können „gleiche Sachverhalte“ also mit unterschiedlichen Wörtern ausgedrückt werden. Dieser Aspekt tritt insbesondere durch „Obfuscation“ in den Vordergrund, da dabei regulärer Quelltext durch eine Fassung ersetzt wird, die die gleiche Funktionalität „in anderer Schreibweise“ umsetzt. Nicht zuletzt Schadcode bedient sich bewusst dieser Möglichkeit, indem der Code bewusst modifiziert wird, um eine Erkennung anhand fixer Signaturen zu vermeiden bzw. um die Erkennung durch geringfügige Modifikation zu erschweren.

1.1.2. Anforderungen an ein Machine Learning-Modell

Als Schlussfolgerung aus der im vorigem Abschnitt erläuterten Problemstellung lässt sich festhalten, dass zum Vergleich der Ähnlichkeit von Quellcode zunächst eine adäquate Repräsentation des Codes gefunden werden muss, die die Semantik des Programms abbildet. Dabei muss sowohl „Obfuscation“ von Programmcode durch entsprechende Erkennung bzw. Ausfiltern berücksichtigt werden, als auch die gemeinsame Abbildung zusammengehöriger Instruktionen.

Um schließlich mithilfe dieses Modells die Ähnlichkeit von Quellcode zu prüfen, gibt es, in Übernahme entsprechender Termini aus dem Bereich Machine Learning, zwei Herangehensweisen:

1. *Supervised Learning*: Dabei passiert eine Prüfung der Ähnlichkeit durch Zuweisung eines Codefragments zu einer vordefinierten Kategorie (Klassifikation). Diese vorausgewählte Kategorie kann beispielsweise zu einer Unterscheidung von harmlosen und schädlichem Code führen (Malware-Erkennung durch Binärklassifikation). Bei einem „Multi Class"-Ansatz könnte eine Kategorie genauso gut eine tatsächliche Programmkategorie sein (z.B. Games, Communication, Traveling, ...). Um eine dahingehende Unterscheidung zu ermöglichen, wird also ein Modell benötigt, bei dem Klassen mit den Trainingsdaten encodiert werden können. Anders ausgedrückt, muss beim maschinellen Lernen von Codefragmenten die zugehörige Klasse irgendwo spezifiziert werden. Sofern dies nicht nur ein Wort ist, wie z.B. die Kategorie, sondern etwa die Beschreibung einer Anwendung, braucht es einen Mechanismus, um diese Beschreibung zunächst als Klasse aufzubereiten. Bei natürlichem Text ist es naheliegend, hierbei Techniken aus dem NLP-Bereich heranzuziehen.

2. *Unsupervised Learning*: Der zweite Ansatz kommt ohne Klassen aus und soll Code direkt miteinander vergleichen. Traditionellerweise kommt dabei „Clustering" zum Einsatz, wobei die Zugehörigkeit von Features zu einzelnen Gruppen („Clustern") festgestellt wird. Der Vorteil dieses Ansatzes ist, dass sich dadurch Code vergleichen lässt, von dem nicht bekannt ist, welche Funktionalität er eigentlich implementiert. Wurde also ein Modell mit dem Code von Apps trainiert, bei denen man weiß, dass sie harmlos sind, hätte Schadcode im Idealfall eine andere semantische Struktur bzw. würde nicht harmlosem Code entsprechen und daher über den „Unsupervised"-Ansatz gefunden werden. Ein Nachteil dieses Ansatzes liegt darin, dass ein Algorithmus die Zugehörigkeit von Code-Fragmenten zu den Clustern selbst herausfindet und die Cluster daher nur eine „relative" Kategorisierung darstellen, bei der die Ähnlichkeit anhand der Distanz zu den umgebenden Clustern angegeben wird. Praktisch heißt das, dass ohne Weiteres nicht bekannt ist, warum etwas im gleichen Cluster vorkommt bzw. was die einzelnen Cluster darstellen.

Ein Grundproblem, das unabhängig vom Algorithmus für maschinelles Lernen zu lösen ist, liegt im typischerweise umfangreichen Konvolut an Quellcode selbst. Würde z.B. jede Instruktion des Quellcodes einer Android-Anwendung ein Feature in einem semantischen Modell darstellen, wäre die Abbildung in einem hochdimensionalen Raum unumgänglich. Neben der dramatischen Steigerung der Komplexität des Modells, müsste in weiterer Folge wieder eine Dimensionalitätsreduktion, z.B. mittels Clustering oder „Principal Component Analysis" stattfinden, um aus dem Modell etwas Sinnvolles auszulesen.

Die Selektion, welche Eigenschaften aus dem Quellcode einer Anwendung für die Klassifikation besonders relevant und welche möglicherweise ausgespart werden können, um die Komplexität des semantischen Modells zu reduzieren, wird im Zuge dieses Projekts nicht detaillierter erfasst. Die damit verbundene Fragestellung, welchen „Informationsverlust" das Ausklammern von Quellcode-Features, wie sog. „Junk Code" auf Klassifikation oder Clustering hat, ist ein separater Studienbereich, dessen Behandlung den Rahmen dieses Projekts sprengen würde. Der Einfachheit halber wird daher angenommen, dass alle Instruktionen, die im Quellcode vorkommen, gleichermaßen relevant sind.

1.2. Zielsetzung

Um den in der Einführung beschriebenen Einschränkungen bei „Code Fingerprinting“ und den Implikationen bei Einsatz von „Obfuscation“ zu begegnen, wird in diesem Projekt ein technischer Ansatz vorgeschlagen, um die Ähnlichkeit von Quellcode dennoch vergleichen zu können. Wie im vorherigen Abschnitt beschrieben, gibt es mehrere Varianten, wie dieser Vergleich stattfinden kann.

Im Bewusstsein der angeführten Herausforderungen verfolgt dieses Projekt folgende Zielsetzungen:

- Aufbereitung von Quellcode in einer Art und Weise, um mithilfe von Machine Learning ein semantisches Modell zu trainieren. Diese Verarbeitung muss in erster Linie „Obfuscation“ und kontextuelle Zusammenhänge in Code berücksichtigen.
- Es ist ein semantisches Modell zu wählen, das variable Methodengrößen und unterschiedliche Längen von Instruktionen so berücksichtigen kann, dass die Dimensionalität des Modells nicht „explodiert“.

Das trainierte Modell soll in weiterer Folge für die Behandlung konkreter Aufgaben eingesetzt werden. Beispiele für derartige Anwendungsszenarien sind etwa:

- Suche nach Android-Apps, deren Quelltext einer gewissen Beschreibung oder Kategorie im PlayStore entspricht.
- Clustering des Codes von Anwendungen basierend auf ihrer natürlichen Beschreibung, in der App enthaltenen Zeichenketten („Strings“) oder dem UI-Layout.
- Die Identifikation von Malware durch Auffinden von „ähnlichem Schadcode“.

In den nachfolgenden Abschnitten wird zunächst auf die besonderen Eigenschaften von Quellcode eingegangen, um darauf basierend einen Algorithmus zu finden, der diese Charakteristika abbilden zu können. In weiterer Folge wird ein semantisches Modell mithilfe dieses Mechanismus erstellt und zur Beantwortung ausgewählter Fragestellungen eingesetzt. Darin soll einerseits die Praxistauglichkeit des gewählten Ansatzes herausstellen und andererseits auf aktuelle Forschungsfragen eingegangen werden, die bislang mangels einer entsprechenden Lösung in dieser Form nicht beantwortet werden konnten.

2. Semantisches Verständnis von Programmcode

In den letzten Jahren wurden mehrere Techniken im Bereich „Machine Learning“ entwickelt, die über die reine Extraktion von Informationen aus Text hinausgehen. Das Erkennen von Zusammenhängen, automatisierte Gewichtung des Informationsgehalts und Abstraktion sind die Zielsetzungen neuer Algorithmen, um aus einem Konvolut Relevantes herauszufiltern.

Inspiriert von „Latent Semantic Analysis“ haben Teufl et al. [1] einen Ansatz vorgestellt, um semantische Abhängigkeiten in einem Modell abzubilden. Beliebige Eingabedaten dabei mithilfe von Clustering und Distanzmethoden in einen Vektorraum projiziert und in Abhängigkeit zueinander gebracht. Im Blick auf Text als Eingabe entstanden Ansätze wie word2vec [2] und GloVe [3]. Wörter werden dabei als Vektoren in einen hochdimensionalen Raum transformiert, der die semantischen Beziehungen zwischen ihnen repräsentiert. Vektoren, die nahe beisammen liegen, weisen dabei auf ähnliche semantische Bedeutungen hin. Diese als

„word embeddings“ oder „word vectors“ bekannten Wortbeschreibungen zeigen für einzelne Wörter auf, mit welchen sie verwandt sind, da sie gemeinsam oder in Nähe zueinander vorkommen. Ein Wortvektor ist also ein dichter Vektor für jedes Wort eines Satzes, der so gewählt wird, dass er ähnlich ist zu Vektoren von Wörtern, die in ähnlichen semantischen Zusammenhängen auftreten. Da gewisse sprachliche Konstrukte wie Präpositionen oder Artikel verhältnismäßig oft vorkommen, jedoch per se deshalb nicht mehr Informationen liefern, würde eine, sich rein auf Häufigkeit von Wörter beschränkende, Technik die Qualität der Klassifikation reduzieren. Neuere Methoden wie word2vec sind sich diesem Umstand bewusst und setzen ein sog. „Sub-sampling“ häufiger Wörter ein. Bei diesem auch als *tf-idf-Maß* bekannten Bewertungsschema werden vorkommende Wörter werden dabei verhältnismäßig geringer gewichtet oder gar entfernt, seltene dafür umso höher priorisiert.

Programmcode in Hochsprachen ähnelt normalem Text in vielerlei Gesichtspunkten, wenngleich die zugrundeliegende syntaktische, strukturelle und grammatikalische Form unterschiedlich ist. Wie bei Text müssen sämtliche Instruktionen und Argumente *valide* und aus einem definierten Alphabet abgebildet werden. Die Abfolge der Wörter hat in beiden Fällen den grammatikalischen Regeln der jeweiligen Sprache zu entsprechen. Der Aufbau ist bei Programmcode hierarchisch, z.B. Methode in Klasse, welche von Package gekapselt wird, wobei diese Hierarchie etwas über die semantische Zugehörigkeit von Instruktionen aussagt. Aufgrund dieser *Ähnlichkeit der Eingabedaten* liegt die Annahme nahe, dass ein NLP-basierter Ansatz für normalen Text sich auch für die Verwendung mit Programmcode adaptieren ließe.

Word2vec wurde entworfen, um einzelne Wörtern anderen in einem hochdimensionalen Raum gegenüber stellen zu können. Zunächst wird jedes im Text vorkommende Wort in einem fixen Vokabular aufgenommen und durch einen Vektor repräsentiert. Beim wortweisen Verarbeiten eines Textes wird dann die bedingte Wahrscheinlichkeit maximiert, dass ein gewisses Wort *t* im Kontext mit einem anderen Wort *c* gemeinsam auftritt. Da diese Operation für das gesamte Vokabular vorgenommen wird, werden alle Wortvektoren in Zusammenhang zueinander gebracht. Word2vec unterstützt hierfür zwei Modellarchitekturen: *Skip-gram* und *Continuous Bag of Words (CBOW).* Bei ersterem Algorithmus werden auf Basis von einem gegebenen Wort *t* Kontextwörter *c* positionsunabhängig vorhergesagt. Mithilfe von CBOW ist das umgekehrte, also die Vorhersage eines Wortes *t* auf Basis von mehreren Kontextwörtern *c* möglich.

Im Hinblick auf Programmcode ergeben sich dabei mehrere interessante Einsatzszenarien für diese Algorithmen. Auf Basis eines gewissen Codefragments (Sammlung von Instruktionen) könnte einer Instruktion gesucht werden, die semantisch ähnlich ist (CBOW). Praktisch heißt das, dass so etwa Instruktionen gefunden werden müssten, die immer gemeinsam auftreten. Wird als Input z.B. ein Fragment gegeben, bei dem Dateioperationen (Öffnen, Schreiben) abgebildet sind, müssten andere Dateioperationen (Lesen, Schließen) semantisch sehr ähnlich sein. Umgekehrt könnte bei Skipgram etwa nach Gabe einer Instruktion geprüft werden, in welchem Kontext sie üblicherweise verwendet wird. Wird also beispielsweise als Input der Aufruf einer API-Methode verwendet (z.B. Kamerazugriff bei Android-Apps), müsste als Kontext vorhergesagt werden können, in welchem Rahmen diese Methode üblicherweise verwendet wird. Im Idealfall zeigen sich in der Vektorrepräsentation somit Zusammenhänge, wie Instruktionen typischerweise verwendet werden. Je nach Anwendungsfall könnte dies also auch umgedreht interessant sein, um Anomalien zu finden.

Eine wesentliche Einschränkung von word2vec ist die Tatsache, dass es grundsätzlich nur auf Basis einzelner Wörter operieren kann. Es kennt also keine Kompositionen mehrerer Wörter, Sätze, Absätze oder ganzer Dokumente. Behelfsmäßig ist es denkbar, z.B. jeden Wortvektor eines Satzes zu normieren im

Hinblick auf alle anderen bestehenden Wortvektoren. Dies geht jedoch mit einem Informationsverlust einher, da Wortvektoren nur mehr in Zusammenhang eines Satzes mit anderen vergleichbar sind und nicht mehr einzeln. Quoc Le et al. [4] haben folglich „Paragraph Vectors“ (auch bekannt als doc2vec) als Erweiterung zu word2vec vorgeschlagen. Dabei werden auf Basis von Wortvektoren weitere Vektoren auf übergeordneter Ebene, wie z.B. Absätzen oder ganzen Dokumenten gelernt, sodass schließlich ein Vergleich der semantischen Ähnlichkeit auf dieser Ebene ermöglicht wird. Anstelle jedes Wort mit jedem anderen in einem Dokument gegenüberzustellen, sieht doc2vec die Verwendung eines Fensters vor. Wohingegen es bei word2vec sein kann, dass Matrizen recht dünn besiedelt sind, wenn z.B. alle Wortkombinationen nur selten oder einmalig gemeinsam auftreten, beschränkt die Verwendung eines Fensters die Größe der Vektoren und reduziert damit auch die Anzahl der benötigten Dimensionen. Um mitunter kompliziert darstellbare Zusammenhänge von Zusammenhängen in Sätzen abzubilden, werden weitere Matrizen verwendet. Der Speicherbedarf von doc2vec ist somit vergleichsweise hoch und benötigt entsprechend viele Eingabedaten. Angesichts der eingangs in diesem Dokument angesprochenen codemäßig sehr umfangreichen Android-Anwendungen, sollte doc2vec demnach prädestiniert sein.

2.1. Ein Programm ist (nicht) Text

Die im vorherigen Abschnitt angeführten Verfahren wurden entwickelt, um semantische Zusammenhänge in Text natürlicher Sprache zu erkennen. Instruktionen können praktisch durchaus analog mit natürlichen Wörtern gesehen werden. Darüber hinaus besteht Quellcode jedoch nicht nur aus Wörtern im klassischen Verständnis. Neben wortähnlichen Ausdrücken wie Strings und Bezeichner, besteht die Semantik von Programmcode aus Keywords (*float, const, else, ...)*, Konstanten, Symbolen und Operatoren. Würde man eine Zeile des Programmcodes folglich als Satz sehen und „=“ als Wort, würde dieses folglich in verhältnismäßig vielen Zusammenhängen auftreten. Das in word2vec verwendete *tf-id-Maß* würde die individuelle Aussagekraft dieses Wortes reduzieren und falls die Wahrscheinlichkeit der Worthäufigkeit unter einen vordefinierten Schwellwert (Threshold) fällt, das Wort aus dem Korpus entfernen.

Die Konsequenz daraus ist, dass es nicht mehr möglich ist, die Komposition von Instruktionen in Wortvektoren abzubilden. Es kann folglich Instruktionen geben, bei denen Operatoren oder Keywords fehlen. Gleichermaßen kann dies auch auf Bezeichner oder Identifier von Variablen zutreffen, wenn diese mithilfe von Obfuscation ersetzt wurden. Da die meisten Schemata für Obfuscation deterministisch vorgehen und Klassen, Methoden und Packages oft durch simple Buchstaben wie „a, b, c“ ersetzt werden, ist zu erwarten, dass dies auch Auswirkungen auf die Klassifikation haben wird, da word2vec alle Wörter nur einmal im Korpus führt. Konkret heißt das, dass das Wort „a“ in Methode 1 eine andere semantische Bedeutung haben kann als in Methode 2. Code-Obfuscation trägt somit dabei, Idiome von Wörtern zu schaffen, die de facto keine sind. Im Hinblick auf die geplanten Anwendungsszenarien könnte das beispielsweise bedeuten, dass die Wörter in verschleiertem Code ggf. sehr gering gewichtet oder ausgefiltert werden. Um diesen Effekt zu untersuchen, wird in einem nachfolgenden Abschnitt ein Experiment mit dem Code von Schadprogrammen vorgenommen, der typischerweise stark verschleiert ist.

Wird Programmcode als Text mithilfe von NLP-Techniken wie word2vec gelernt, ist die zugrunde gelegte Ausgangsbasis die Sprachdefinition von natürlichem Text. Der Ansatz eignet sich somit nicht, um syntaktische Regeln einer gewissen Programmiersprache zu lernen. Dieser Effekt hat praktisch ambivalente Auswirkungen. Einerseits lässt sich Programmcode dadurch generisch (sprachagnostisch) lernen, sodass keine vorherige Definition der jeweiligen Grammatik hinterlegt werden muss. Andererseits verhindert diese mangelnde Sprachkenntnis zugleich, dass das resultierende Modell die Regeln der Sprachsemantik

berücksichtigen könnte. Zusammenhänge im Kontrollflussgraph wie „Basic Blocks“, Methodendefinitionen und Klassen somit nicht als solche erkannt. Praktisch heißt das, dass die komplette hierarchische Struktur verloren geht, nach der Quellcode verfasst ist und Codefragmente nur als Ansammlung von Wörtern (*Bag of Words*) gesehen werden können und nicht als sprachspezifisches Konstrukt.

3. Anwendung mit realem Programmcode

Eine vergleichsweise einfach zu verwendende Implementierung des Algorithmus von Mikolov et al. wird in einem quelloffenen Python-Framework namens gensim [2] bereitgestellt. Für die nachfolgend durchgeführten Szenarien wird hierauf zurückgriffen.

3.1. Datenaufbereitung

Für die nachfolgenden Experimente wurden drei Datenquellen verwendet:

1. Für einen ersten Versuch mithilfe von Java-Quelltext wurden mithilfe der „Google Big Query“Schnittstelle und eines kostenlosen Kontingents mehr als 3 Mio. Java-Quelltexte abgerufen.
2. Da ein systematischer Download von Android-Anwendungen aus Googles „PlayStore“ nicht mehr ohne weiteres möglich ist, wird eine Breitenstudie über viele Android-Anwendungen signifikant erschwert. Im Rahmen einer wissenschaftlichen Publikation namens „PlayDrone“

[5] wurde dennoch eine Methodik entwickelt, um programmatisch eine Vielzahl an Anwendungen abzurufen. Dieses Archiv wird dankenswerterweise öffentlich bereitgestellt [3] und wird für die hier vorgenommenen Experimente verwendet. In der Sammlung befinden sich 1,1 Mio. Android-Anwendungen. Aus Speichergründen wählen wir für die nachfolgende Untersuchung aus jeder der 42 Kategorien zu gleichen Teilen insgesamt 2.000 Anwendungen aus. 3. Ein Set aus ca. 2.000 Android-Anwendungen, die sog. Botnets implementieren. Der Datensatz [4] wurde nach Anfrage vom „Canadian Institute for Cybersecurity“ bereitgestellt und umfasst 14 Familien von Schadcode mit jeweils ca. 100-300 Derivaten pro Kategorie.

Für die Verarbeitung von Quellcode mithilfe von doc2vec muss der zu lernende Programmcode zunächst entsprechend aufbereitet werden. Instruktionen werden dafür „tokenisiert“ d.h. in Wörter unterteilt und sequentiell als Inputdaten übergeben. Nach ersten Experimenten mit den vorliegenden Daten hat sich gezeigt, dass es sinnvoll wäre, Informationen wie Operatoren oder Anführungszeichen gleich im Vorfeld der Verarbeitung aus dem Datensatz zu entfernen. Im Wissen, dass diese Informationen von doc2vec ansonsten ohnehin ausgefiltert werden würden, verringert sich zudem das verwendete Vokabular, was wiederum positiv zur Performance beiträgt. Darüber hinaus wurden auch Symbole, Punktationen und alle anderen Zeichen entfernt, die weder Zahl noch Buchstabe waren. Als letzten Prozess dieses „Feature Engineerings“ wurden auch Kommentare entfernt, die entweder manuell bei der Entwicklung (in Java-Code) oder beim Dekompilieren von Android-Anwendungen vom Werkzeug baksmali [5] eingefügt wurden.

Das Resultat dieser Aufbereitung ist ein sog. „Token Stream“, der die Instruktionen des Quelltexts sequentiell wie natürlichen Text abbildet. Wie im vorherigen Abschnitt dieses Dokuments erläutert, ist durch diesen Schritt die hierarchische Struktur verloren gegangen. Durch das (pragmatische) Entfernen von Operatoren und Symbolen, wurde dazu beigetragen, dass doc2vec diese Informationen nicht von selbst ausfiltern muss. Auf der anderen Seite wurden jedoch Zusammenhänge in Code entfernt, da der Ausdruck „a = b + 4“ nun gleich aussieht wie „a / b & 4“.

3.2. Geeignete Hyperparameter

Bei der Anwendung des doc2vec-Algorithmus sind mehrere Hyperparameter zu spezifizieren, die den Trainingsprozess und die Qualität des resultierenden Modells maßgeblich beeinflussen. Die Auswirkungen der wesentlichsten werden nachfolgend kurz veranschaulicht:

- *„window size“:* Die Größe des Fensters, das für doc2vec definiert, wie groß der umliegende Kontext eines Wortes sein soll. Diese Zahl zeigt somit an, wie viele Wörter vor und nach einem gewissen Wort als Kontext gesehen werden. Praktisch hat die „korrekte“ Wahl der Fenstergröße große Relevanz:

o Wird diese Fenstergröße zu klein gewählt, ergeben sich potentiell nur vage Zusammenhänge zwischen Codefragmenten.

2 https://radimrehurek.com/gensim/index.html 3 https://archive.org/details/android_apps&tab=about 4 https://www.unb.ca/cic/datasets/android-botnet.html 5 https://github.com/JesusFreke/smali o Wird sie hingegen zu groß gewählt, kann es z.B. sein, dass der Kontext einzelner Instruktionen über die Grenzen der beinhaltenden Methode hinausgehend gesehen wird. Das ist insofern problematisch, da die Reihenfolge, in der Methoden vorkommen, keinerlei Relevanz hat und sie in ihrer Anordnung austauschbar sind. Potentiell wird mit somit ein „verfälschter“ Kontext trainiert. Ein naiver Ansatz, um diesem Problem zu begegnen, ist „Shuffling“ von Methoden. Im Bewusstsein, dass die Anordnung von Methoden keine Relevanz hat, werden zusätzlich permutierte Anordnungen von Methoden trainiert, um den angeführten Effekt zu egalisieren. In Vorwegnahme der Ergebnisse der nachfolgend vorgestellten Anwendungsszenarien hatte „Shuffling“ zu keinen Verbesserungen bei der Klassifikation geführt. Eine mögliche Ursache wäre die teilweise Redundanz in den Trainingsdaten, die durch Subsampling in word2vec zu geringerer Gewichtung entsprechender Vektoren führt.

Analog dazu kommt es zu Problemen, wenn auf eine Instruktion keine weiteren folgen und das kontextuelle Fenster damit nicht ausgefüllt und mit Nullwerten gefüllt werden muss. Die Kontextinformation betroffener Instruktionen ist so nur zur Hälfte gegeben und in ihrer „Aussagekraft“ entsprechend limitiert.

- *Skip-gram* oder *DBOW:* Die Wortvektoren für das gesuchte semantische Modell können sowohl mit Skip-gram oder „Distributed Bag of Words“ (DBOW) trainiert werden. In Vorwegnahme der Ergebnisse hat sich gezeigt, dass DBOW zu einem weniger komplexen Modell führt, das schneller trainiert. Diesen Effekt bestätigen auch von Forscher von IBM in eigenen Experimenten mit Text [6].

- *„Sub-sampling threshold“:* Eine Einstellung, ab wann Instruktionen ausgefiltert werden, da sie zu oft vorkommen. Ist dieser Schwellwert zu gering gewählt, kann es passieren, dass zu häufig vorkommende Instruktionen zu oft als „ähnlich“ aufgefunden werden. In der Praxis finden sich somit fast immer die gleichen „Ähnlichkeiten“ in Instruktionen – fast nie jedoch selten auftretende Zusammenhänge. Ist der Wert zu hoch gesetzt, werden hingegen zu viele Instruktionen ausgefiltert und es dünnen sich im Extremfall „dichte“ Kontextvektoren aus, da man die häufiger vorkommenden Verweise entfernt hat. Je „dünner“ (sparse matrix) die verbleibenden Vektoren jedoch sind, desto geringer die semantische Ähnlichkeit zu anderen Vektoren und die Wahrscheinlichkeit ähnliche Vektoren zu finden.

Alle Hyperparameter sind abhängig von den Trainingsdaten und können nur experimentell herausgefunden werden. Dieser Prozess gestaltet sich in der Praxis mühsam und zeitaufwändig, da wiederholt Modelle mit unterschiedlichen Parametern trainiert und getestet werden müssen.

3.3. Szenario: Semantische Korrelation in Java-Code

Um semantische Ähnlichkeiten in Quelltext aufzufinden und herauszufinden, wie der dekompilierte Code für Android-Anwendungen aufbereitet werden muss, wurde als erster Versuch der gecrawlte Java-Programmcode zu einem „Token Stream" umgewandelt und anschließend mithilfe von gensim ein Modell für den Skip-gram- und DBOW-Algorithmus trainiert. Auf einem CPU-Cluster mit 2x Xeon E5-2690V4 mit 44 CPU-Kernen / 88 Threads und 512GB DDR4-RAM hat das Trainieren von 3 Mio. Java-Quelltexten ca. 50 Stunden in Anspruch genommen.

Nach Experimenten mit unterschiedlichen Hyperparametern wurden an das vermeintlich beste semantische Modell Anfragen gestellt. Das DBOW-Modell war unabhängig von den Hyperparametern immer eindeutiger. Die Ergebnisse zeigen neben den Treffern jeweils die Wahrscheinlichkeit der Ähnlichkeit bzw. ein Maß für die semantische Korrelation.

DBOW: Suche nach Ausdruck ‚mkdir':

```
[('mkdirs', 0.9018791215485157),
('delete', 0.8468794554512447),
('length', 0.8379912014003256),
('listFiles', 0.7949878517815778),
('exists', 0.7191047789872173),
('toPath', 0.6892378784884848),
```

```
('directory', 0.618748118416117),
('IOException', 0.5897101137512974),
('hashCode', 0.3815624351804243),
('pathSeparator', 0.3487075510871312)]
```

Wie in der Aufstellung erkennbar ist, findet die Suche nach dem Term „mkdir" mit relativ hoher Sicherheit semantisch verwandte Ausdrücke. Dass „mkdirs" in einem semantisch ähnlichen Kontext wie „mkdir" verwendet wird, ist naheliegend. Auffallend ist aber, dass praktisch alle gefundenen Ausdrücke in irgendeiner Form mit der *java.io.File* -API in Zusammenhang stehen. Das Gros der gefundenen Ausdrücke sind gleich wie „mkdir" Methoden, die diese API bereitstellt. Es ist offensichtlich, dass die Verwendung von „mkdir" somit in semantischer Hinsicht einer ähnlichen Verwendung wie der Methoden „delete", „length", etc. entspricht.

DBOW: Suche nach Ausdruck ‚Cipher' + ‚AES':

```
[('getInstance', 0.9618015115012138),
('update', 0.9491455872433064),
('doFinal', 0.9090165635657992),
('DES', 0.7574617771411558),
('Base64', 0.6194844137134579),
('String', 0.5648848615135364),
('Key', 0.5550910121312865),
('byte', 0.5525835101801441),
('ENCRYPT_MODE', 0.4744502598452352),
('BC', 0.3562923667148217)]
```

Die Suche nach „Cipher“ + „AES“ zielt darauf ab, Verwendungen der Java API *javax.crypto.Cipher* in Verbindung mit AES als Verschlüsselungsverfahren zu finden. Es zeigt sich in der Aufstellung, dass die Suchergebnisse ohne Ausnahme auf Zeichenketten oder Methoden verweisen, die typischerweise auch vorkommen, wenn Javas Cipher-API eingesetzt wird. Eine Verwendung des Wortes „Cipher“ in Kombination mit „AES“ ist im gegebenen Trainingsset scheinbar nur im kryptographischen Kontext aufgetreten. Dass „getInstance“ eine überaus hohe Korrelation aufweist, ist insofern naheliegend, da die API üblicherweise mit „Cipher.getInstance(„AES“)“ instanziiert wird. Das Auffinden der Methoden „update“ und „doFinal“ veranschaulicht die gefundene kontextuelle Nähe, da beide Methoden oft gemeinsam mit „getInstance“ verwendet werden. Die Tatsache, dass „DES“ ebenfalls gefunden wird, lässt darauf schließen, dass doc2vec die semantische Stellung von „AES“ mit hoher Wahrscheinlichkeit korrekt erfasst hat.

In Anbetracht der gewonnenen Ergebnisse kann davon ausgegangen werden, dass doc2vec grundsätzlich in der Lage ist, Wortvektoren sinnvoll aus Quelltext zu extrahieren. Inwieweit die Häufigkeit der Verwendung in den Trainingsdaten oder die im Vorfeld vorgenommenen Substitutionen einen Einfluss auf das Modell haben, lässt sich empirisch nur schwer abschätzen.

3.4. Szenario: Android Malware-Klassifikation

In Anlehnung an die in Abschnitt 1.2 exemplarisch angeführten Zielsetzungen wurden Experimente mit dem in Abschnitt 3.1 vorgestellten Datensatz mit Android-Malware durchgeführt. Die zentralen Fragestellungen dieser Untersuchung lauteten:

1. Ist Schadcode so „individuell“, dass sich daraus auch einzelne Familien bzw. „Klassen“ von Malware mit doc2vec erkennen lassen?
2. Kann mithilfe von doc2vec eine Prüfung von Codefragmenten vorgenommen werden, hinsichtlich ihrer Ähnlichkeit mit Schadcode? Anders gefragt: „Wie stark ähnelt ein Codefragment Malware?“

Anstelle für diese Klassifikation von Malware statische Signaturen, die auf Basis einzelner Features (Zeichenketten, Hashsumme eines Segments von Bytes) eine gewisse Konfidenz ableiten, zugrunde zu legen, soll jene in dieser Analyse mithilfe der semantischen Ähnlichkeit von Quelltext vorgenommen werden. Potentieller Schadcode wird daher nicht nur an singulären Eigenschaften, sondern dem semantischen Kontext erkannt, in dem er verwendet wird.

3.4.1. Klassifikation zu Malware-Familien

Pro Klasse / „Familie“ von Malware wurde zunächst jede verfügbare Android-Anwendung mithilfe von baksmali dekompiliert und der erhaltene Quellcode „tokensiert“ (siehe Abschnitt 3.1). Konkret wurden dafür analog zu Java-Code Wörter und Zahlen als „Token Stream“ in einem fiktiven „Dokument“ repräsentiert, mit dem doc2vec trainiert werden kann. Für den Quelltext jeder Applikation wurde ein derartiges „Dokument“ erstellt. In weiterer Folge wurden die aufbereiteten Features mit der Bezeichnung der Malware-Klasse (als sog. „Tag“) als Vektoren an doc2vec übergeben. Als Resultat ergab sich ein semantisches Modell, das sich für folgende Arten von Anfragen eignet:

- Welche Tags sind auf Basis der gelernten Dokumente ähnlich zueinander bzw. semantisch miteinander verwandt? Praktisch lässt sich so prüfen, ob der Code von Malware einer Klasse jenem einer anderen bekannten Klasse ähnelt.
- Welcher Tag passt zu einem gegebenen Dokument? Von einem gegebenen Quelltext soll also darauf geschlossen werden, zu welcher Malware-Familie er am ehesten zugehörig ist.

DBOW: Suche nach Tag ‚Zitmo‘:

```
[('AnserverBot', 0.7888568681009560),
('Geinimi', 0.7212692728248119),
('DroidDream', 0.6207648197796104),
('Pletor', 0.6157988665847729),
('TigerBot', 0.4614560982145340),
('Bmaster', 0.4434960967623050),
('Rootsmart', 0.4047252924414560),
('Sandroid', 0.4006684900021963),
('Nickyspy', 0.3454003959317934),
('MisoSMS', 0.3350836396960659)]
```

Die exemplarische Suche nach der Botnet-Familie „Zitmo“ als Tag zeigt Tags, deren zugehörige Dokumente ähnlich zu jenen der „Zitmo“-Klasse sind. Dass eine gewisse Korrelation zwischen Anwendungen der einzelnen Klassen vorliegt, zeigen die in den Aufstellungen ersichtlichen Wahrscheinlichkeiten. Bedauerlicherweise zeigen diese Ergebnisse nicht, worin diese Übereinstimmungen begründet sind. Dass es z.B. eine semantische Ähnlichkeit von 33,5% von „Zitmo“ und „MisoSMS“ gibt, muss nicht notwendigerweise am Malware-Code selbst liegen, sondern könnte mit der gleichen Applikationsstruktur der Android-Anwendung zu tun haben. Durch Subsampling und dem Ausfiltern wiederkehrender Muster sollte dieser Effekt zwar reduziert sein, wie groß die Auswirkung aber wirklich ist, lässt sich schwer bemessen.

Eine inhaltliche Erklärung weshalb das Botnet „Zitmo“ relativ gesehen so viel ähnlicher ist zu „AnserverBot“ als zu „MisoSMS“ könnte sein, dass „Zitmo“ gleich wie „AnserverBot“ und „Geinimi“ mit ihrem „Command & Control Server“ über HTTP kommuniziert und entsprechende Funktionalität hierfür implementiert. Im Gegensatz dazu erhalten „NickySpy“ und „MisoSMS“ Befehle beispielsweise via SMS. Dies kann selbstverständlich nur als Indikator gesehen und nicht als de facto Ursache für die Unterschiede herangezogen werden.

DBOW: Suche nach Dokument eines Derivats von ‚Zitmo':

```
[('2ce8c664bc8316232053a3f53ef80dc4.apk', 0.8507057336319708), # Zitmo
('6b7e27efaae0d0370f97c6be95f66eac.apk', 0.7488989483880678), # AnserverBot
('8e1b295ed2bac630161dd8da2e630891.apk', 0.7227419211169886), # Zitmo
('62bbaf8575d6a9d70beb37d5a1533f09.apk', 0.6917920069379007), # Geimini
('0ed7b7cef3d16458d459b7a939298fe6.apk', 0.4959984659992310), # Zitmo
('03f68e41d3e4e6f28e4c25c470bed025.apk', 0.4361644663091829), # AnserverBot
('3e43fec301add4b08ca161eb7e9f7369.apk', 0.4199831404806658), # Zitmo
('5ea0bc4a29da33790ca04dd23899a839.apk', 0.4111504749131647), # TigerBot
('9bd367a6317163610ba883eca4e6fb10.apk', 0.4046204977273468), # Wroba
('9f3269db5d0eabe6c5303ee016ddb8c3.apk', 0.3564623652935695)] # Bmaster
```

Bei der Prüfung, zu welcher Klasse der Quelltext eines (zuvor mittrainierten) Malware-Samples der Zitmo-Familie gehört, wurde nach Dokumenten gesucht, deren Quelltext semantisch ähnlich ist. Wie in der Aufstellung erkennbar, ist der erste Treffer der Aufstellung eine Datei bzw. ein Dokument namens „2ce8c664bc8316232053a3f53ef80dc4.apk", was dem gesuchten entspricht. Konkret heißt das, dass das Malware-Sample der Familie „Zitmo" mit einer Wahrscheinlichkeit von 85% wieder dieser Familie geordnet werden konnte. Diese Übereinstimmung spricht grundsätzlich dafür, dass die Malware-Samples dieser Familie hinreichend eindeutig und die Qualität des Modells hinreichend hoch sind. Es darf angenommen werden, dass mit mehr Samples pro Familie die Qualität nochmal gesteigert werden könnte.

3.4.2. Generische Malware-Klassifikation

Abgesehen von der Ähnlichkeit von Malware untereinander, könnte die Prüfung auf semantische Ähnlichkeit von Quelltext auch Aufschluss darüber geben, inwieweit der Code einer Anwendung jener von Malware ähnelt. Vergleichbare Ansätze zur Auswertung semantischer Informationen in Anwendungen [7, 8] zielen darauf ab, Applikationen zu finden, die in schädlicher Absicht modifiziert und neu gepackt wieder verteilt wurden. Wohingegen der Fokus dabei auf der Erkennung minimaler Unterschiede zwischen Anwendungen liegt, wird im Folgen beabsichtigt, eine binäre Klassifikation von Android-Anwendungen auf Basis des Quelltexts durchzuführen.

Für die Unterscheidung in zwei Klassen wurde der Quelltext sämtlicher Malware-Samples aus dem vorherigen Experiment mit dem Tag „Malware" trainiert. Analog dazu wurde im Anschluss das Modell erweitert um eine gleich große Anzahl von Anwendungen, die bekanntermaßen harmlos sind und mit einem entsprechenden Tag versehen wurden. Da sich der Datensatz des „PlayDrone"-Projektes in 42 Kategorien unterteilt, wurden aus jeder Kategorie zufällig 48 Anwendungen ausgewählt. Die Idee dieser Selektion ist es, zu verhindern, dass durch die Kategorie möglicherweise ein Bias hinsichtlich Codestruktur oder Funktionalität eingeführt werden würde.

Eine anschließende Prüfung der Genauigkeit (*accuracy)* des Modells mithilfe eines Validierungsdatensatzes hat aufgezeigt, dass trainierte Applikationen mit einer Genauigkeit von ca. 62% der richtigen Klasse zugeordnet werden können. Folglich wurde versucht, die Hyperparameter des Modells anzupassen, um dadurch die Genauigkeit zu erhöhen. Da dabei keine wesentlichen Veränderungen feststellbar waren,

wurden daraufhin weitere harmlose Applikationen trainiert. Im finalen Modell wurden insg. 10.000 Anwendungen trainiert, wobei 2.000 davon Malware waren. Durch die Erweiterung des Modells steig die Genauigkeit auf 69%.

Anschließend wurden stichprobenartig Suchen durchgeführt, um zu testen, inwieweit der Code bekannter Anwendungen Malware ähnelt.

DBOW: Suche nach Dokument ‚com.facebook.orca ‘:

```
[('com.facebook.katana.apk', 0.8447913955027601),
('com.facebook.lite.apk ', 0.7884876888562416),
('com.snapchat.android.apk ', 0.5009157540584981),
('jp.naver.line.android.apk ', 0.4888120860787510),
('com.tencent.mm.apk', 0.4782006415590805),
('com.kakao.talk.apk ', 0.4781787510511734),
('com.twitter.android.apk', 0.4743611116850410),
('com.icq.mobile.client.apk', 0.4722287883036332),
('messenger.chat.social.messenger.apk', 0.4676671157995251),
('com.vkontakte.android.apk ', 0.4617724914213935)]
```

Die exemplarische Suche nach Dokumenten, die ähnlich dem Paket der Messenger-Applikation von Facebook „com.facebook.orca“ sind, hat Anwendungen aufgezeigt, deren Quelltext semantisch ähnlich sein soll. Wie in der Aufstellung erkennbar, findet sich darunter keine Anwendung, die zuvor als Malware trainiert wurde. Vielmehr jedoch zeigen Ähnlichkeiten zu weiteren Applikationen, die als harmlos klassifiziert wurden. Insbesondere die gefundene Nähe zu weiteren Facebook-Anwendungen (com.facebook.katana, com.facebook.lite) ist naheliegend. Applikationen derselben Firma teilen oft Codefragmente untereinander oder sind sich funktional ähnlich. Im konkreten Fall konnte durch eine zusätzliche manuelle Analyse der angeführten Anwendungen herausgefunden werden, dass der Facebook-Messenger und die beiden weiteren Anwendungen Quelltext teilen. Der gleiche Quelltext wurde somit in mehreren Apps eingesetzt – was die semantische Korrelation sehr plausibel erscheinen lässt. Applikationen weiterer Hersteller teilen diese Codefragmente jedoch mutmaßlich nicht. Es ist insofern bemerkenswert, dass mit semantischer Ähnlichkeit zum Facebook-Messenger auch weitere Messenger bzw. verwandte Apps gefunden wurden. Eine mögliche Erklärung wäre, dass alle der angeführten Anwendungen ähnliche Berechtigungen verwenden und sich im Quelltext der Anwendungen Referenzen auf entsprechende API-Aufrufe befinden. Ob diese Implementierungsaspekte jedoch tatsächlich ausschlaggebend sind für eine semantische Ähnlichkeit, lässt sich nur sehr schwer nachvollziehen.

3.5. Diskussion der Ergebnisse

In den Aufstellungen einzelner Suchen in den jeweils trainierten Modellen wurden Anwendungen mit ähnlichem Quelltext oder „Tags“ in gewisser prozentualer Übereinstimmung gefunden. Die Tatsache, dass doc2vec bzw. word2vec eine Projektion sämtlicher Wörter in einen hochdimensionalen Raum vornehmen, verhindert jedoch, dass die gefundenen Ergebnisse faktisch untermauert werden können. Um herausfinden zu können, welche Features tatsächlich wesentlich sind für die Klassifikation, würde eine Möglichkeit benötigt, den Abstand einzelner Features im hochdimensionalen Raum auszuwerten. Durch die Reduktion der Projektionen entsteht ein gewisses Problem der Nachvollziehbarkeit.

Ein weiteres Problem, welches grundsätzlich mit allen Ansätzen modernder Machine Learning-Verfahren auftritt, ist die Wahl geeigneter Hyperparameter für den jeweiligen Datensatz. In den hier vorgenommenen Experimenten hat sich gezeigt, dass z.B. Skip-Gram und DBOW in Abhängigkeit von der Menge der gegebenen Trainingsdaten unterschiedlich gut klassifizieren.

Über die Gründe, weshalb eine Trennung von harmlosen und schadhaftem Code nur mit 72% gelingt, kann ohne weiteres nur spekuliert werden. Doc2vec liefert hierüber keine Auskünfte über Metriken oder Cluster. Die Ursache könnte demnach sowohl im gewählten Algorithmus liegen, als auch in den Trainingsdaten selbst. Die Tatsache etwa, dass der Code von Malware-Anwendungen nicht nur Malware enthält, sondern jene typischerweise nur ein Teilsegment ausmachen, könnte die Ergebnisse beeinflussen. Eine höhere Gewichtung von selten vorkommendem Code in word2vec hilft zwar, diesem Umstand zu begegnen. Es verbleibt dennoch die Frage, ob „individueller" (stark gewichteter) Code in einer Malware-Anwendung tatsächlich die Malware-Funktionalität bereitstellt oder eben nur vergleichsweise selten vorkommt, indem z.B. eine den Schadcode tarnende Anwendung ebenfalls enthalten ist und untypisch implementiert wurde.

Der Nachteil eines NLP-basierten Ansatzes zeigt sich in der Aufbereitung, die für word2vec notwendig war. Es geht nicht nur die Hierarchie verloren, in welche Code eingebettet ist, sondern auch die semantische Trennung zwischen einzelnen Elementen. Ein starres Fenster berücksichtigt somit nicht, wann eine Methode anfängt oder aufhört. Insofern kann eine Klassifikation, wie sie im vorliegenden Fall praktiziert wurde, auch einfach ausgetrickst werden, indem etwa ein Malware-Sample konstruiert wird, bei dem Instruktionen in neue oder andere Methoden verschoben wurden. Ist folglich die Fenstergröße zu groß gewählt, wird kein übereinstimmender Kontext mehr gefunden, woraufhin die gesamte die globale Wahrscheinlichkeit für eine semantische Korrelation sinkt.

Im Vergleich zu normalen Quelltext kommt bei Mobilanwendungen der Umstand hinzu, dass viele bei Anwendungen „Obfuscation" verwenden und Bezeichner in Quelltexten daher keine sinnvollen Rückschlüsse auf den Verwendungszweck ermöglichen. In Kombination mit der fehlenden Unterscheidung zwischen eigentlichen Instruktionen und Bezeichnern werden somit kontextuelle Zusammenhänge gelernt, die potentiell falsch sind. Ein pragmatischer Ansatz wäre, „Obfuscation" heuristisch festzustellen und entsprechend verschleierte Bezeichner durch eindeutige Platzhalter zu ersetzen. Die Eindeutigkeit dieser Platzdrücker geht jedoch wieder zulasten der gebildeten Wortvektoren, da so Zusammenhänge nicht erkannt werden, die eigentlich zusammengehören.

Analog dazu könnte helfen, sog. „Junk Code", der sich nicht unmittelbar auf die Funktionalität der Anwendung bezieht, im Vorfeld auszufiltern. In der Praxis ist die Unterscheidung, welcher Code zu einer Anwendung gehört und welcher von Drittherstellern kommt, jedoch leider nicht einfach zu bewerkstelligen. Bei Betrachtung des dekompilierten Codes einer Android-Anwendung gibt es hierbei keine Unterscheidungsmerkmale, die keine Segregation ermöglichen würden.

4. Fazit

Im Zuge dieses Projekts wurde ein Konzept erarbeitet, um semantische Ähnlichkeiten von Quelltext in Anwendungen zu analysieren. Nach einer Erörterung der Problemstellung und Eingrenzung der Zielsetzung, wurde ein Ansatz aus dem NLP-Bereich adaptiert, um Programmcode auf semantische Ähnlichkeiten hin untersuchen zu können. Es wurde vorgezeigt, wie gegebene Daten aufbereitet werden müssen, um von word2vec bzw. dessen Weiterentwicklung doc2vec verarbeitet zu werden.

Für ein besseres Verständnis der Funktionsweise und Auswahl von Hyperparametern bei doc2vec wurde zunächst Java-Code gelernt. Die gewonnenen Ergebnisse haben die grundsätzliche Tauglichkeit des gewählten Ansatzes untermauert. Daran anschließende Verwendungsszenarien mit harmlosen, als auch schädlichen Android-Anwendungen, haben veranschaulicht, wie vergleichsweise einfach Fragestellungen an trainierte Modelle Ergebnisse liefern können, die von traditionellen Ansätzen nicht erfasst sind. Die Experimente haben jedoch auch Grenzen bei der Anwendung von doc2vec aufgezeigt, die im Wesentlichen daran liegen, dass Quelltext nur zu einem gewissen Maß regulärem Text gleicht. Die Transformationsschritte, um daraus regulären Text zu formen, gehen mit einem Informationsverlust einher, der einer guten Klassifizierung abträglich ist.

Die Erkenntnisse aus diesem Projekt beschränken sich rein auf den gewählten Ansatz mittels doc2vec und einer vorangehenden Transformation hin zu einer Wortrepräsentation. Um die semantische Ähnlichkeit von Quelltext noch besser beurteilen zu können, wäre es also sinnvoll, eine Verarbeitung auch mithilfe anderer Algorithmen zu prüfen.

Referenzen

[1] P. Teufl, H. Leitold und R. Posch, „Semantic Pattern Transformation: Applying Knowledge Discovery Processes in Heterogeneous Domains," *13th International Conference on Knowledge Management and Knowledge Technologies, I-KNOW '13,* 2013.

[2] T. Mikolov, I. Sutskever und K. Chen, „Distributed Representations of Words and Phrases and their Compositionality," *Neural Information Processing Systems NIPS,* 2013.

[3] J. Pennington, R. Socher und C. Manning, „Glove: Global Vectors for Word Representation,"

Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP, 2014.

[4] L. Quoc und T. Mikolov, „Distributed Representations of Sentences and Documents,"

ICML'14 Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32, 2014.

[5] N. Viennot, E. Garcia und J. Nieh, „A Measurement Study of Google Play," *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2014),* 2014.

[6] L. Jey Han und T. Baldwin, „An Empirical Evaluation of doc2vec with Practical Insights into Document Embedding Generation".

[7] J. Crussell, C. Gibler und H. Chen, „Andarwin: Scalable detection of semanitcally similar Android Applications," *ESORICS,* 2013.

[8] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi und T. N. Nguyen, „Accurate and efficient structural characteristic feature extraction for clone detection," *Fundamental Approaches to Software Engineering,* 2009.

[9] L. Li, „AndroZoo++: Collecting Millions of Android Apps and Their Metadata for the Research Community," 2017.

[10] K. Weinberger, A. Dasgupta und J. Langford, „Feature hashing for large scale multitask learning,“ *Proceedings of the 26th Annual International Conference on Machine Learning (ICML ‘09),* 2009.