Zentrum für sichere Informationstechnologie – Austria
Secure Information Technology Center – Austria

A-SIT

Seidlgasse 22 / 9, 1030 Wien
Tel.: (+43 1) 503 19 63–0
Fax: (+43 1) 503 19 63–66

Inffeldgasse 16a, 8010 Graz
Tel.: (+43 316) 873-5514
Fax: (+43 316) 873-5520

http://www.a-sit.at
E-Mail: office@a-sit.at

DVR: 1035461
ZVR: 948166612
UID: ATU60778947

TOPOLOGISCHE ANALYSE VON PROGRAMMCODE

Version 1.0 vom 13.06.2019
Johannes Feichtner – johannes.feichtner@a-sit.at

Bei der sicherheitsorientierten Analyse von Applikationen tritt zunehmend der Umstand auf, dass die eigentlich zu untersuchenden Eigenschaften von Programmen im Code immer schwerer auffindbar und nachvollziehbar werden. Zunehmende Komplexität und umfangreicherer Funktionsumfang von Programmen tragen dazu bei, dass sich manuelle Überprüfungen der Implementierung nur auf gewisse Aspekte konzentrieren können (z.B. Code mit kryptographischen Funktionsroutinen), andere, potentiell ebenfalls sicherheitsrelevante Aspekte aber außer Acht gelassen werden oder nicht in allen vorkommenden Verwendungsszenarien analysiert werden können.

Im Zuge dieses Projekts wurde ein Ansatz gesucht, um Programmcode so aufzubereiten, dass eine topologische Suche ermöglicht wird. Angesichts der gegebenen Herausforderungen lag dabei die graphische Darstellung von Code als AST (Abstract Syntax Tree) für die Suche von zusammengehörigen Codefragmenten nahe. Das Ziel war es, die Analyse großer Anwendungen zu vereinfachen und effizient feststellen zu können, wo sich sicherheitskritische Programmteile befinden.

Inhaltsverzeichnis

Inhaltsverzeichnis		1
1.	Einleitung	2
2.	Verwandte Arbeiten	4
3.	Abbildung von Strukturen in Abstract Syntax Trees	6
4.	AST-Pfade mit kontextuellen Informationen	7
4.1.	**Praktische Anwendung**	**9**
5.	Beschreibung von Variablen über AST-Pfade	10
6.	Fazit	12
Literaturverzeichnis		12

© A-SIT, Partner für Sicherheitsfragen

1. Einleitung

Umfangreiche Leistungsressourcen moderner Geräte im Desktop- und Mobilbereich ermöglichen die Ausführung zunehmend komplexerer und inhaltlich umfangreicherer Anwendungen. Damit einhergehend steigt auch der nötige Aufwand, Programme zu analysieren, gegebenenfalls Schwächen in der Implementierung aufzudecken und die korrekte Umsetzung sicherheitsrelevanter Funktionalität zu überprüfen. Anwendungen mit vergleichsweise großen Kontroll- und Datenfluss-graphen, sowie einer Vielzahl möglicher Verzweigungen (Branches) in der Ausführung, erschweren fundierte Rückschlüsse auf die Sicherheit von Implementierungen. Neben der Identifikation von Code-Zusammenhängen über Programmteile hinweg, stellt auch das eigentliche Auffinden sicherheitsrelevanter Funktionalität in der Praxis eine wesentliche Herausforderung dar. Setzt sich Funktionalität durch einen Zusammenschluss verschiedener Code-Elemente zusammen, z.B. ein Teil aus dem Hauptprogramm und ein weiterer aus einer Drittbibliothek, erhöht dies den Aufwand und das Risiko, dass bei Analysen sicherheitsrelevante Aspekte übersehen werden.

Um diesem Umstand beizukommen, ist es daher essentiell (1) Zusammenhänge in Programmcode, die eine gewisse Funktionalität abbilden, identifizieren zu können und (2) diese Zusammenhänge so isolieren bzw. extrahieren zu können, um basierend darauf ähnliche Codefragmente identifizieren zu können.

Zum Feststellen von Ähnlichkeiten in großen Zusammenhängen wurden in den letzten Jahren erhebliche Fortschritte im Bereich des maschinellen Lernens erzielt. Die grundsätzliche Basis für die dabei verwendeten Modelle sind Vektoren, die in Korrelation zueinanderstehen und deren Distanz durch ein Maß wie etwa den Euklidischen Abstand oder die Kosinus-Distanz ausgedrückt werden. Wenngleich Techniken aus dem Machine Learning-Bereich konzeptionell geeignet sind, in großen Datenmengen ähnliche Codeteile zu finden, ist es aus der Sicht aktueller Forschung noch ungelöst, wie Code bestmöglich für die Anwendung mit maschinellem Lernen repräsentiert werden könnte. Die wesentliche Herausforderung dabei ist es, den Informationsgehalt von Syntax, Semantik und die Zusammenhänge von Codeteilen so abzubilden, dass sie als „Features“ für Machine Learning eingesetzt werden können. Anknüpfend an diese „strukturelle“ Fragestellung folgt zwangsläufig die Suche nach einer Lösung, die gewählten Features zu diskretisieren, sie nach ihrer „Wertigkeit“ zu gewichten und eine vektorielle Repräsentation zu finden.

Mit der zunehmenden Rechenleistung von Grafikkarten wurden in den Bereich des maschinellen Lernens neue Technologien für semantisches Textverständnis eingebracht (word2vec [1], GloVe [2]), die konzeptionell eine sehr ähnliche Problemstellung wie bei Programmcode in Angriff nehmen. Intuitiv liegt eine Analogie zwischen Text mit Sätzen, Absätzen und Kapitel und Methoden, Klassen und Packages vor. Trotz der ähnlichen Problemstellung gibt es aufgrund der syntaktischen und semantischen Eigenheiten von Code keine hinreichende Äquivalenz, die es ermöglichen würde, Methoden aus dem Bereich des Textverständnisses auf Programmcode anzuwenden. Die Tatsache, dass in Programmen Methoden in beliebigen Klassen und Packages vorkommen können, gilt nicht für textuelle Absätze in Relation zu Kapiteln in einem regulären Text. Obwohl technisch möglich, wären diese Ansätze bei Anwendungen mit Codefragmenten daher nicht in der Lage, Syntax und Semantik zu berücksichtigen, sowie Relationen zwischen Codeteilen abzubilden.

Zu den Eigenheiten von Programmcode gehört beispielsweise, dass die Reihenfolge der Anordnung einzelner Methoden keine Rolle spielt, EntwicklerInnen Codeteile in beliebigen Methoden und Methoden in beliebigen Klassen gruppieren können. Obwohl es stilistisch naheliegt, Methoden, die zusammen gesehen eine gewisse Funktionalität implementieren, in zusammengehörenden Klassen zu implementieren, gibt es aus technischer Sicht keine Einschränkungen, die die Gestaltungsfreiheit bei der Entwicklung betreffen würden. Es ist daher (auch für das Vorhaben in diesem Projekt) davon auszugehen, dass der Programmcode in Methoden eine gewisse Funktionalität abbildet und auf verwandte Codeteile durch weitere Methodenaufrufe verweist. Eine weitere Herausforderung ist, dass Code nicht nur aus Wörtern besteht, die normalem Text entsprechen, sondern durch Operatoren, Symbole und weitere syntaktische Elemente eine individuelle Semantik umgesetzt wird, bei der Elemente unterschiedliche Aufgaben erfüllen. Der Operator =, welcher einer Variable einen Wert zuweist, hat daher einen anderen Informationsgehalt als der Name einer Variable, der im Idealfall ihren Verwendungszweck bzw. ihre Aufgabe beschreibt.

In der Praxis führen diese Unterschiede von Code-Elementen dazu, dass nicht alle Teile als „gleichrangig“ in einem Modell verarbeitet werden können. Wird für die Suche nach semantisch ähnlichen Codeteilen beispielsweise anhand von Variablennamen gesucht, könnte diese Suche nur dann zuverlässige Ergebnisse produzieren, wenn der Bezeichner von Variablen ihre Verwendung so präzise beschreibt, dass damit ein Vergleichsargument geschaffen wird. Ungeachtet dessen wurde dieser Ansatz in verwandten Arbeiten untersucht, indem Methoden aus dem NLP-Bereich (Natural Language Processing) auf Code angewandt und Text als lineare Sequenz einzelner Ausdrücke mithilfe von Machine Learning trainiert wurde [3], [4]. Wie aus diesen Arbeiten auch einschränkend hervorgeht, ist der Einsatzzweck vieler Variablen weniger im Namen definiert, als in den Operationen, die auf sie angewandt werden und daher den *Kontext* einer Verwendung darstellen. Im Gegensatz zur Suche semantisch verwandter Wörter in einem Text, gibt es zurzeit keine Lösungen, ähnliche Codeteile anhand von Variablen mitsamt Operationen zu finden bzw. den Kontext der Verwendung zu berücksichtigen. Würden Operatoren wie ! oder = beispielsweise als Text im gleichen Kontext gesehen, würde eine Suche kaum brauchbare Ergebnisse hervorbringen, da diese Operatoren in sehr vielen Kontexten eingesetzt werden und daher individuell wenig Aussagekraft besitzen. Die Erkennung, welche Rolle diese syntaktischen Elemente spielen und eine anschließende Gewichtung bzw. „Bewertung ihrer Relevanz“ bedingt jedoch die Kenntnis sprachspezifischer Eigenschaften. Ansätze müssten dazu also an gewisse Programmiersprachen angepasst werden.

Die zentralen Fragestellungen für die Verarbeitung von Code mithilfe von Techniken aus dem Bereich des maschinellen Lernens sind daher wie folgt:

- Wie kann Code „behandelt“ bzw. transformiert werden, damit er mithilfe von Machine Learning-Modellen verarbeitet werden kann? Die Kenntnis des in sich beschränkten Sets von Syntax und Semantik einer Programmiersprache helfen bei der Gewichtung einzelner Code-Elemente. Eine wesentliche Schwierigkeit stellt jedoch die Abbildung von Zusammenhängen zwischen einzelnen Codeteilen dar, die bislang nicht gelöst ist.

- Wenn Zusammenhänge zwischen Codefragmenten abgebildet werden: Wo beginnen diese und wo enden sie? Wie lange muss eine kontextuelle Relation mindestens und maximal sein, um so aussagekräftig zu sein, dass ähnliche Codefragmente gefunden werden können? Ist der gesuchte Zusammenhang inhaltlich „kurz“, steigt die Wahrscheinlichkeit, viele weitere kurze Fragmente zu finden, die zwar funktional verwandt sind, aber keinen Schluss auf größere Programmteile ermöglichen. Zu lange Zusammenhänge können wiederum zu „eindeutig“ bzw. individuell sein, um sie woanders wiederzufinden.

Mit der Möglichkeit, Code in geeigneter vektorieller Form für Machine Learning-Modelle aufzubereiten, wäre eine allgemein anwendbare Darstellung geschaffen, um unabhängig von einem bestimmten Modell Code mithilfe von maschinellem Lernen zu trainieren. Welches Modell für diese „Features“ folglich gewählt wird, hängt im Wesentlichen vom Ziel der Analyse ab. Um Implementierungen sicherheitsrelevanter Funktionalität in Programmen aufzufinden, sind neben der Suche semantisch ähnlicher Codefragmente unter anderem folgende Szenarien praxisrelevant:

- **Vorhersage von Methodennamen:** Davon ausgehend, dass der gesamte Code einer Methode eine gewisse Funktionalität implementiert, wäre die „Zusammengehörigkeit“ des Codes innerhalb der Methode eine Art Cluster, die die Methode beschreibt. Angesichts des Umstands, dass Methoden typischerweise so implementiert werden, dass sie kleine Sub-Tasks von Programmen erfüllen, ist diese Annahme durchaus naheliegend. Der Name einer Methode sollte daher im Idealfall zusammenfassend die inhaltliche implementierte Funktionalität beschreiben.

  Unter der Prämisse, semantische Zusammenhänge im Code einzelner Methoden abzubilden, könnte somit ein Machine Learning-Modell dahingehend trainiert werden, Methodennamen vorherzusagen. Mit Methodennamen, die die Funktionalität des beinhalteten Codes widerspiegeln, würde eine Analyse von Programmen wesentlich vereinfacht, da relevante Programmteile bereits auf Basis ihrer Bezeichnung identifiziert werden könnten.

```
function cloneObject(object) {
  if (!object)
    return object;
  let clone = {};
  for (let key in object) {
    clone[key] = object[key];
  }
  return clone;
}
```

Wie in diesem JavaScript-Codebeispiel exemplarisch dargestellt, sind alle Zeilen der Methode (`function`) im Kontext der Implementierung einer bestimmten Funktionalität relevant. Können die semantischen Informationen der einzelnen Codezeilen in einen Zusammenhang gebracht werden, könnten Literale in diesen Codezeilen als Labels für ein Machine Learning-Modell dienen, das die Funktionalität beschreibt.

- **Vorhersage von Variablennamen:** Die Abbildung von Strukturen über Code-Elemente kann auch dazu eingesetzt werden, Bezeichner für Variablen zu finden, die in semantisch ähnlichen Zusammenhängen aufgetreten sind. In der Praxis kann so ein Modell trainiert werden, das die Bezeichner von Variablen in Abhängigkeit ihres Code-Kontextes vorhersagen kann.

```
def execute(c):
  o = Popen(c, stdout=PIPE,
      stderr=PIPE, shell=True)
  c, x = p.communicate()
  r = p.returncode
  if r:
    raise CalledProcessError(r,c)
  else:
    return c.rstrip()
```

```
def execute(cmd):
  process = Popen(cmd, stdout=PIPE,
                  stderr=PIPE, shell=True
  out, err = process.communicate()
  ret = process.returncode
  if ret:
    raise CalledProcessError(ret,out)
  else:
    return out.rstrip()
```

Wie in diesem Python-Codebeispiel veranschaulicht, kann auf Basis semantischer Informationen ein Rückschluss auf gute bzw. aussagekräftigere Variablennamen gelingen. Ein verwandter Ansatz, wo für die Vorhersage von Variablennamen ein Wahrscheinlichkeitsmodell mit sog. Conditional Random Fields (CRF) gearbeitet wurde, ist JSNice [3]. Das entwickelte Modell berücksichtigt die Syntax und Semantik von JavaScript und kann somit nicht mit Programmcode, der in anderen Sprachen verfasst ist, umgehen.

2. Verwandte Arbeiten

Verwandte Arbeiten fokussieren sich auf die Vorhersage einzelner Eigenschaften in Programmcode unter Einbeziehung von Syntax und Semantik gewisser Sprachen. Im Folgenden wird auf die Herangehensweise bestehender Arbeiten eingegangen, um darauf aufbauend einen eigenen Ansatz herzuleiten, der die im vorigen Abschnitt vorgelegten Fragestellungen berücksichtigen kann.

Für die **Vorhersage von Variablennamen** gibt es Forschungsarbeiten, die auf die Syntax und Semantik gewisser Programmiersprachen ausgerichtet sind. Mit Fokus auf die Programmiersprache Java wurde mit „DeGuard“ [3] ein statistisches Modell erarbeitet, das mithilfe von CRFs Zusammenhänge von Programmelementen abbildet. Die dabei trainierten Features berücksichtigen die bei Java-Anwendungen typischen Verbindungen von Elementen, wie etwa, dass Klassen in Packages organisiert sind oder Schleifen nur im Rahmen (Scope) von Methoden auftreten können. Für die praktische Anwendung benötigt das vorgeschlagene Modell Java-Programmcode, der semantisch möglichst zielgerichtet beschreibt, welche Aufgabe eine Variable repräsentiert. Das Ergebnis erinnert folglich an das im vorherigen Abschnitt präsentierte Umwandlungsbeispiel. Zur Vorhersage wird die bayessche MAP-Inferenz (Maximum-a-posteriori-Hypothesis) eingesetzt, die im Unterschied zur vollständiger Inferenz den Vorteil bietet, dass auch Informationen zur Vorhersage verwendet werden können, die nicht in den Trainingsdaten enthalten sind. Dadurch kann die Varianz des MAP-Punkteschätzwerts verringert werden, was jedoch gleichzeitig das Ergebnis verzerrt. In der Umsetzung erinnert das Verfahren an Maximum Likelihood Learning mit Weight Decay, das sich in ähnlicher Form mittlerweile bei vielen „Deep Learning“-Modellen wiederfindet.

Stark verwandt mit dem Ansatz von Bichsel et al. ist die Arbeit zu „Predicting Program Properties from „Big Code“ [4]. Ebenfalls über CRFs wird dabei auf Syntax und Semantik von Javascript-Code Bezug genommen. Für Python-Code haben Raychev et al. [6] einen alternativen Ansatz über Entscheidungsbäume („Decision Trees“) vorgeschlagen, der neben Variablennamen auch Methodennamen vorhersagen kann.

Für die **Vorhersage von Methodennamen** haben Allamanis et al. [3] einen Ansatz vorgeschlagen, der Anleihe nimmt an Ideen der „Text Summarization“. Wie im vorherigen Abschnitt bereits beleuchtet, wurde dabei eine NLP-Technik zur Textzusammenfassung so abgewandelt, dass sie auf Java-Code angewandt werden kann und die Ausgabe der Vorhersage ein Set potentieller Methodennamen ist. Dahinter steht ein sog. „Convolutional Neural Network“ (CNN), das Zusammenhänge von Code-Elementen (nicht aber die Elemente selbst!) als Vektoren abbildet und mit einem sog. „Attention Layer“ jene Vektoren hoch gewichtet, die für die Vorhersage semantisch sehr relevant erscheinen.

Einen alternativen Ansatz zur Prediction von Methodennamen haben Raychev et al. [6] verfolgt. Zur Analyse, welche Code-Elemente in Zusammenhang miteinander stehen, wird der „Abstract Syntax Tree“ (AST) von Methoden verwendet. Im Gegensatz zum NLP-abgewandelten Ansatz von Allamanis et al., werden Relationen somit auf Basis ihrer Abhängigkeiten im AST gebildet und nicht auf Grundlage ihrer Koexistenz im Programmcode. In der vorliegenden Arbeit haben die Autoren eine explizite Grammatik für JavaScript-Code definiert. Mit Kenntnis der semantischen Zusammenhänge von JavaScript-Code wurden Code-Features extrahiert, die Relationen zwischen Knoten des AST repräsentieren. Für die Anwendung des vorliegenden Ansatzes mit anderen Programmiersprachen müsste somit mindestens eine weitere Grammatik, sowie eine Kontroll- und Datenflussanalyse vorgenommen werden.

Ebenfalls auf Basis des AST von Methoden aber unter Einbeziehung weiterer kontextueller Relationen, haben Bielik et al. [7] einen Ansatz vorgeschlagen, bei dem der semantische Kontext einzelner Kosten des AST identifiziert werden soll. Vorhersagen über Variablennamen werden somit nicht über einzelne Knoten im AST selbst, sondern anhand des Kontextes abgeleitet, in dem die Knoten vorkommen. Mit Fokus auf die Semantik von JavaScript-Code gelingt es den Autoren so, Vorhersagen zu treffen, die nicht von der Präsenz einzelner AST-Knoten abhängig sind. Wie bereits in der Einleitung dieses Berichts angeführt, bringt der Einbezug des Kontextes jedoch auch Schwierigkeiten mit sich (z.B. wie groß das „Kontext-Fenster“ ist), die von Bielik et al. durch einen vergleichsweise pragmatischen Zugang und im speziellen Fall für JavaScript gelöst wurde.

Nach Studie verwandter Arbeiten können nachfolgende Schlussfolgerungen gezogen werden:

- „State-of-the-art“-Forschung fokussiert sich auf die Vorhersage von Variablen- und Methodennamen. Aktuelle Ansätze verwenden hierfür Conditional Random Fields (CRFs), Neuronale Netze bzw. Deep Learning, sowie statische Modelle mit MAP-Inferenz.

- Die Verarbeitung von Code-Elementen passiert entweder aktiv, indem Programmteile als Text oder gewisse Knoten im AST repräsentiert werden oder passiv, indem die Relationen zwischen einzelnen Elementen durch Vektoren abgebildet werden. Je nach Ansatz beschreiben die Vektoren die textuelle oder semantische Distanz zueinander.

- Wenngleich gezeigt wurde, dass die Behandlung von Codefragmenten als Text unter Abwandlung von Methoden aus dem NLP-Bereich passieren kann, setzen verwandte Arbeiten überwiegend auf Eigenschaften der Knoten des AST.

- Eine wesentliche Rolle spielen in bestehenden Arbeiten die Syntax, Semantik und Zusammenhänge, wie Code-Elemente abgebildet werden. ASTs werden typischerweise auf Methodenebenen gebildet und auch alle neuronalen oder statistischen Modelle, die von verwandten Arbeiten bislang vorgeschlagen wurden, zielen darauf ab, die Namen von Variablen oder Methoden auf dieser Basis vorherzusagen.

In verwandten Arbeiten finden sich bereits wesentliche Überlegungen, die mit den Zielsetzungen dieses Projekts übereinstimmen. In Anbetracht der eingangs aufgestellten Zielsetzungen fehlt jedoch bisher ein Ansatz, der unter anderem folgende Motive verwirklicht:

- Der Ansatz sollte nicht auf die Syntax und Semantik einzelner Programmiersprachen fokussiert, sondern unabhängig von bestimmten Grammatiken sein. Wenngleich bestehende Arbeiten für die jeweiligen Problemstellungen praxistaugliche Lösungen vorschlagen, sind diese teilweise stark angepasst für das jeweilige Umfeld und können nicht beliebige Code-Elemente oder Relationen zwischen ihnen beschreiben.

- Die Vorhersage von Namen für Variablen oder Methoden betrifft nur einen kleinen Ausschnitt der Tasks für die Methoden des maschinellen Lernens eingesetzt werden können. Wenn es also möglich wäre, Code-Elemente so abzubilden, dass sie nicht gebunden sind an eine gewisse Verarbeitungsform, wie z.B. CRFs oder CNNs, könnten diese Features neben der Prediction von Namen auch für weitere Aufgaben eingesetzt werden.

- Der gewünschte Ansatz sollte unabhängig von konkreten Machine Learning-Algorithmen funktionieren. Je nachdem welcher Task durch die Beschreibung von Code-Elementen gelöst werden soll, könnte somit auch jener ML-Algorithmus zum Einsatz kommen, der am besten zur Aufgabe passt und nicht der (einzige), der mit den Code-Features umgehen kann.

Für die Entwicklung eines eigenen Ansatzes lässt sich somit ableiten, dass der Fokus auf den AST sinnvoll erscheint, da dieser Zugang bei verwandten Arbeiten mit ähnlichen Zielsetzungen bereits erfolgreiche Ergebnisse zutage gefördert hat. Es wird jedoch vorgeschlagen, bestehende Ansätze dahingehend zu verbessern, dass (1) eine Repräsentation des ASTs erzeugt wird, die über beliebige Parser extrahierbar, (2) nicht gebunden an eine Programmiersprache und (3) so ausdrucksvoll ist, jede Eigenschaft von Code-Elementen zu erfassen, die syntaktisch vorhanden ist.

3. Abbildung von Strukturen in Abstract Syntax Trees

Um Programmcode in weiterer Folge mithilfe von Algorithmen für maschinelles Lernen trainieren zu können, suchen wir nach einer Repräsentation, die alle semantisch wertvollen Informationen in ASTs abbildet und dennoch nicht angepasst ist an eine gewisse Programmiersprache. Eine Möglichkeit, eine derartige Darstellung zu erhalten, ist das Zerlegen des AST in Einzelteile, die Code-Elemente beschreiben und sich auch in anderen Anwendungen wiederfinden. Eine mögliche Form dieser Dekomposition ist das Bilden von AST-Beziehungen zwischen den Knoten eines AST. Während die Knoten üblicherweise Terminale beschreiben, bilden die Verbindungen zwischen diesen Knoten die Relation ab, in der zwei oder mehrere Terminale zueinanderstehen.

Der Unterschied dieser Betrachtungsweise im Vergleich mit bestehenden Ansätzen liegt darin, dass sich Pfade nicht nur auf die Relation zwischen Elementen beschränken [7], sondern auch die Werte der Terminale dabei miteinbeziehen. Da Code-Elemente typischerweise in einer Hierarchie (je nach Programmiersprache z.B. durch Einrückungen oder {}-Klammern abgegrenzt) vorkommen, bildet auch der AST eine Hierarchie ab, die sich im AST widerspiegelt.

```
var asit = array[cnt];
```

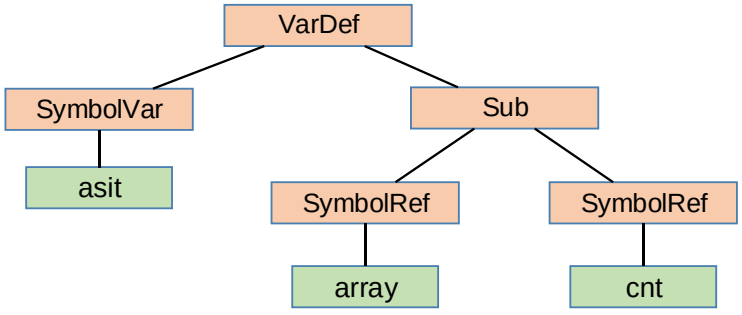


Abbildung 1. Code-Beispiel in JavaScript und der dazugehörige Ausschnitt aus dem AST.

Wie im Kurzbeispiel in Abbildung 1 illustriert, kann die linker Hand angeführte Codezeile in den rechts daneben dargestellten AST überführt werden. Die rot markierten Blöcke repräsentieren die Syntaxelemente in abstrakter Form. Die grünen Blöcke wiederum enthalten Terminale, die in Relation zueinanderstehen. Im konkreten Beispiel kommt das Symbol „asit“ in einer Variablendefinition vor (Überdefinition), die auch die Terminale „array“ und „cnt“ umfasst. Wie im AST erkennbar, gibt es in der vorliegenden Baumstruktur einen Weg bzw. Pfad, um von „asit“ zu „cnt“ oder umgekehrt zu kommen.

Die Abbildung eines Pfades im AST ist in aktueller Forschung nicht formalisiert. Verwandte Arbeiten fokussieren sich der Einfachheit halber entweder auf die Knoten mit Terminalen oder Metainformationen, wie z.B. die Hierarchietiefe oder relative Distanzen zwischen einzelnen Elementen. Der bei Bielik et al. verwendete Kontext beschränkt sich so beispielsweise auf Elemente, die in vergleichsweise kurzem Abstand zueinander im AST vorkommen. Wenngleich dieser Zugang in der Praxis funktioniert, verlieren distanzbasierte Metriken den Kontext, wenn Codeblöcke an andere Stellen in der Methode verschoben werden oder eine direkte Wertzuweisung (Definition), wie im zuvor angeführten Beispiel, durch eine indirekte ausgetauscht wird.

Um herauszufinden, welche AST-Elemente in welchen Zusammenhängen vorkommen können, griffen bestehende Ansätze auf die Grammatik der jeweiligen Programmiersprache zurück. Aus jener könnte beispielsweise hervorgehen, dass ein „SymbolVar“ nur innerhalb einer „VarDef“ vorkommen kann. Die in diesem Projekt verfolgte Absicht, einen generalisierenden Ansatz zu entwerfen, bedingt daher ein Auskommen ohne Grammatik, bei dem Hierarchieebenen dennoch abgebildet werden können.

Im nachfolgenden Abschnitt wird auf Basis dieser Überlegungen ein Ansatz vorgestellt, der die Schwächen bestehender Ansätze ausgleichen soll.

4. AST-Pfade mit kontextuellen Informationen

Um Zusammenhänge zwischen Endknoten (Terminalen) im AST abzubilden, wird ein neuer Ansatz vorgeschlagen, der die Hierarchieebenen des AST durch Pfade abbildet. Die Grundidee ist es, den Kontext des Vorkommens von Terminalen bzw. Knoten des ASTs durch die Relation zu anderen AST-Knoten zu beschreiben. Praktisch wird also der AST „durchwandert“ und der Weg zu anderen Terminalen als „Pfad“ abgebildet. Anstelle einer losen Verbindung von Knoten fügen wir dem Pfad noch weitere Symbole hinzu, die angeben, wie der Weg im AST aussieht, um zu einem anderen Terminal zu gelangen. Basierend darauf könnte ein (Teil)-AST auch rekonstruiert werden.

Um Variablen bzw. Terminale kontextuell zu beschreiben, gibt es folglich ein Set von Pfaden, das ihre Relation zu anderen Knoten im AST angibt. Da naturgemäß nicht jedes Terminal mit jedem anderen verbunden ist, sondern mit jenen, wo ein semantischer Zusammenhang besteht, gibt ein Set von AST-Pfaden folglich an, wie eine Terminale im AST verwendet wird.

Gegenüber bestehenden Ansätzen hat die Darstellung von AST-Relationen mithilfe von Pfaden mehrere Vorteile:

- Pfade können auf Basis eines ASTs automatisch generiert werden. Es ist nicht weiter notwendig, Features, die potentiell interessante Zusammenhänge zwischen Knoten beschreiben, manuell oder anders abzubilden.

- Pfade über AST-Knoten helfen dabei, Verbindungen zwischen Terminalen zu beschreiben, ohne, dass die konkrete Syntax der verwendeten Programmiersprache bekannt sein muss. Für die Extraktion der Pfade ist es unerheblich, wie der jeweilige Parser die Elemente des ASTs beschreibt (SymbolRef, UnaryPrefix, Call, etc.).

- Pfade können hierarchisch gesehen „flach“ ausgedrückt werden. Dieser Umstand ist vor allem deswegen besonders relevant, weil Algorithmen für Machine Learning im Normalfall mit Wörtern als Features trainiert werden, jedoch keine Hierarchie oder Baumstruktur lernen könnten. AST-Pfade sind somit universell einsetzbar und nicht gebunden an gewisse

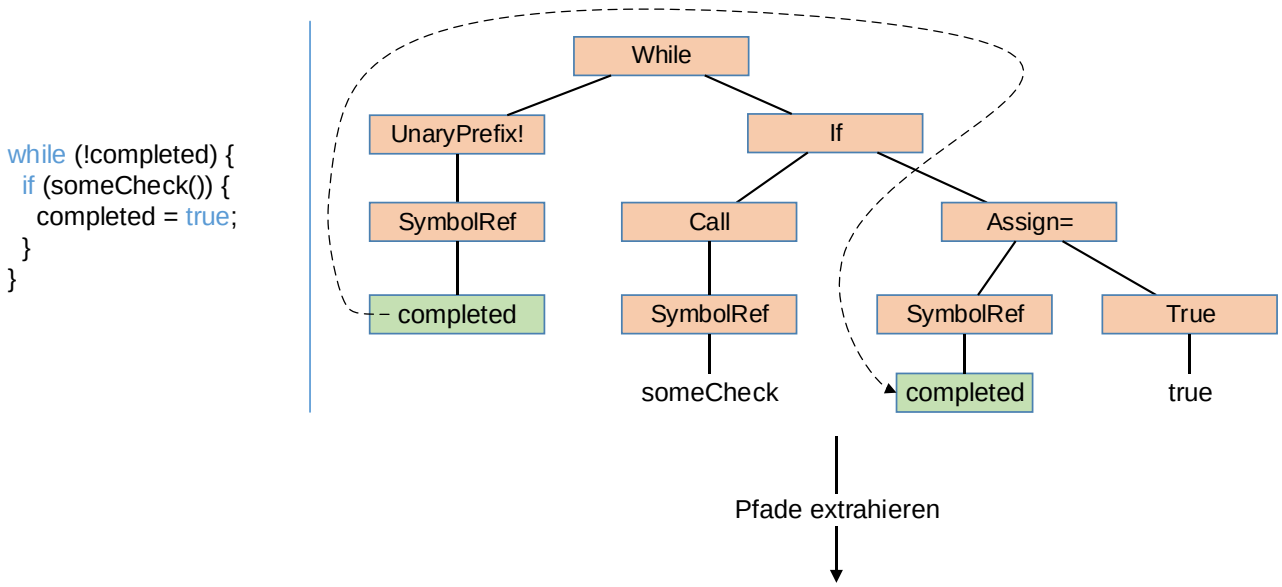
Lernalgorithmen. Sie bilden die Semantik von Programmcode ab und sind dennoch unabhängig von konkreten Programmiersprachen.

Zur besseren Veranschaulichung von AST-Pfaden zeigt Abbildung 2 exemplarisch den AST und die extrahierten AST-Pfade für ein Code-Beispiel mit mehreren Terminalen in JavaScript. Der grundsätzliche Ablauf ist wie folgt:

1. Extrahierung des AST durch Verwendung eines bestehenden Parsers, der die Syntax der jeweiligen Programmiersprache unterstützt. Im konkreten Beispiel wurde ein Parser der Compiler-Suite LLVM[1] verwendet, weshalb die Bezeichnungen im AST entsprechend der Konventionen dieses Parsers abgebildet werden. Bei Verwendung einer anderen kontextfreien Grammatik, wie z.B. von ANTLR[2] würden andere Bezeichnungen gewählt werden, was für die nachfolgende Extraktion von Pfaden nicht weiter relevant ist, da das vordergründige Ziel die möglichst eindeutige Abbildung der Relationen ist.

2. Mithilfe einer Tiefen- oder Breitensuche wird der Baum durchlaufen und es werden alle Pfade zwischen den Endknoten extrahiert. Wir gehen davon aus, dass ein Pfad eine Sequenz von AST-Knoten beschreibt und zusätzliche Verbindungsbezeichner („hoch“, „runter“) die Hierarchie des AST „flach“ abbilden. Das Symbol „≥“ steht sinngemäß für einen Weg nach „oben“, das Symbol „≤“ für einen Weg nach unten. Die konkreten Symbole wurden gewählt, weil sie so nicht als Terminale verwendet werden können, sondern z.B. zu „<“ und „=“ aufgespalten werden würden und es somit zu keinen Verwechslungen kommen kann.

3. Es ergibt sich ein Set an Pfaden, das die Verbindungen sämtlicher Terminale bzw. Endknoten im AST beschreibt. Ein Pfad besteht dabei aus einem Tupel, das nach folgendem Schema aufgebaut ist: **(Startknoten, Pfad, Endknoten)**. Wenngleich im nachfolgenden Beispiel nur Pfade zwischen Terminalen extrahiert werden, könnte das Konzept prinzipiell auf Nicht-Terminale angewendet werden, etwa um Sequenzen von Operationen abzubilden.



P_0 = (**completed**, SymbolRef ≥ UnaryPrefix! ≥ While ≤ If ≤ Call ≤ SymbolRef, **someCheck**)
P_1 = (**completed**, SymbolRef ≥ UnaryPrefix! ≥ While ≤ If ≤ Assign= ≤ SymbolRef, **completed**)
P_2 = (**completed**, SymbolRef ≥ UnaryPrefix! ≥ While ≤ If ≤ Assign= ≤ True, **true**)
P_3 = (**someCheck**, SymbolRef ≥ Call ≥ If ≤ Assign= ≤ SymbolRef, **completed**)
P_4 = (**completed**, SymbolRef ≥ Assign= ≤ True, **true**)

Abbildung 2. Code-Beispiel mit AST und AST-Pfaden.

[1] http://llvm.org
[2] https://www.antlr.org

Das in Abbildung 2 dargestellte Beispiel zeigt im unteren Teil die aus dem AST extrahierten Pfade. Wie im Code-Beispiel linker Hand ersichtlich, kommt das Terminal „completed“ zweifach im Code vor. Das erste Vorkommen als Bedingung der while-Schleife ist negiert, was im AST durch das Statement „UnaryPrefix!“ vermerkt ist. Die Relation zum zweiten Vorkommen des Terminals (strichliert gekennzeichnet) ergibt sich über die Knoten „SymbolRef, UnaryPrefix!, While If, Assign=, SymbolRef“. Mit Abbildung der Hierarchie des ASTs ergibt sich somit folgender Pfad, wobei das Terminal „completed“ sowohl Start-, als auch Endtoken ist:

P_1 = (**completed**, SymbolRef ≥ UnaryPrefix! ≥ While ≤ If ≤ Assign= ≤ SymbolRef, **completed**)

Abbildung 3. Exemplarischer Pfad zwischen gleichem Terminal.

Dieser Pfad drückt die Tatsache aus, dass „completed“ mit einer Negation als Haltebedingung für eine while-Schleife verwendet wird und zeigt, dass der Variable in einer if-Bedingung ein neuer Wert zugewiesen wird, sofern die Bedingung erfüllt / wahr ist. Da Start- und Endtoken des Pfades das gleiche Terminal betreffen und ein Wert zugewiesen wird, zeigt der Pfad außerdem, dass die Änderung des Wertes von „completed“ die Haltebedingung für die Schleife ist.

Für das Terminal „completed“ gibt es darüber hinaus weitere Pfade, wie etwa zum Wert „true“:

P_4 = (**completed**, SymbolRef ≥ Assign= ≤ True, **true**)

Abbildung 4. Exemplarischer Pfad zwischen unterschiedlichen Terminalen.

Dieser AST-Pfad drückt aus, dass die Zuweisung des Wertes „true“ den Wert von „completed“ ändert. Dies bestätigt wiederum, dass das Terminal „completed“ tatsächlich als Haltebedingung für die while-Schleife fungiert.

4.1. Praktische Anwendung

Mit Verweis auf das in Abbildung 2 vorgestellte Beispiel, können AST-Pfade für Vorhersagen von Variablennamen verwendet werden. Aus den Pfaden in Abbildung 3 und Abbildung 4 lässt sich erahnen, dass die Bezeichnung „completed“ semantisch die Haltebedingung beschreibt.

In der Praxis treten Pfade wie die zwei exemplarisch herausgestellten häufig auf. Anstelle der Bezeichnung „completed“ könnte genauso gut „running“, „done“, „stop“, etc. stehen. In allen diesen Fällen wären die AST-Pfade dennoch gleich. Im Hinblick auf Machine Learning wäre es somit möglich, Pfade mitsamt Terminalen zu trainieren und das resultierende Modell für die Vorhersage von Variablennamen zu verwenden. Dadurch gelingt eine Zuweisung bzw. Vorhersage wie im Python-Codebeispiel auf Seite 4 veranschaulicht. Da ein AST kontextuell nahe beisammen liegende Codeteile auch „hierarchisch nahe“ zusammenfasst, fließt in diese Vorhersage prinzipiell auch der Kontext ein, indem die Variablen verwendet werden. Konkret bedeutet das beispielsweise, dass Modelle unterscheiden können zwischen regulären Zuweisungen von Variablenwerten und jenen, die nur dann passieren, wenn zuvor eine if-Bedingung erfüllt wurde. Dieses „kontextuelle Wissen“ erhöht somit implizit die Genauigkeit der Vorhersagen potentieller Variablennamen.

Das mit AST-Pfaden trainierte Modell kann an die jeweilige Problemstellung angepasst werden. Sowohl die bei verwandten Arbeiten verwendeten CRFs, MAP-Inferenz, als auch ein Ansatz über word2vec/doc2vec sind denkbar, da AST-Pfade das von den Ansätzen geforderte Inputformat erfüllt. Würde im Vergleich dazu Programmcode einfach nur als normaler Text trainiert werden (siehe [3]) wäre es nicht möglich, syntaktische und semantische Zusammenhänge so abzubilden, wie sie mit den Symbolen „≥“ und „≤“ im Pfad abgebildet werden. Anstelle einfacher Sequenzen, wie etwa „**completed** -> SymbolRef -> Assign= -> True -> **true**“ kann somit die Hierarchie abgebildet werden, die im AST ersichtlich ist. Diese kann, wie jedoch schon zuvor erläutert, nicht unmittelbar für das Training von Machine Learning Algorithmen verwendet werden. In diesem Sinne bildet das aus AST-Pfaden generierte Modell Code somit möglichst realistisch ab.

5. Beschreibung von Variablen über AST-Pfade

Wie im vorherigen Abschnitt angeführt, eignen sich für die Vorhersage von Variablennamen über AST-Pfade unter anderem die bereits bei verwandten Arbeiten zum Einsatz gekommenen CRFs oder word2vec. Durch Trainieren sog. „Word Embedddings“ gelingt schließlich eine vektorielle Abbildung von AST-Pfaden, die wiederum für die Machine Learning-Algorithmen geeignet ist.

Um AST-Pfade im Hinblick auf die eingangs angeführten Zielsetzungen auf ihre Praxistauglichkeit hin zu überprüfen, wurde der Fokus für Testzwecke auf Android-Anwendungen gelegt. Ein Gutteil der EntwicklerInnen von Mobilanwendungen wendet vor der Veröffentlichung von Apps Techniken wie Code Transformations (Shrinking, Inlining) und Obfuscation an. Bei Letzterem werden Variablen- und Methodenbezeichner ausgetauscht durch semantisch nichtssagende Deskriptoren. Neben einer Reduktion der Codegröße erschwert dieser Schritt manuelle Code-Analysen, da Zusammenhänge und Bedeutungen von Variablen nicht mehr gut ersichtlich sind.

Das mithilfe von AST-Pfaden angestrebte Ziel ist es somit, bestmögliche passende Variablennamen vorzuschlagen und diesen Prozess somit „umzukehren“, also die Anwendung zu deobfuszieren. Das für diese Aufgabe zu trainierende Machine Learning-Modell benötigt in der Trainingsphase somit Anwendungen, die nicht obfusziert sind. Um einen derartigen Datensatz von Apps zu erhalten, wurden im ersten Schritt ca. 70.000 Android-Anwendungen[3] heruntergeladen.

	method	start	end	path
0	b	if-nez	TOKEN_REG	I_STATEMENT_FORMAT21t!I_ORDERED_METHOD_ITEMS!I...
1	a	Landroid/content/Context	0x0	I_METHOD_PROTOTYPE!I_STATEMENT_FORMAT3rc_METHO...
2	a	Landroid/content/Context	Z	I_METHOD_PROTOTYPE!I_STATEMENT_FORMAT3rc_METHO...
3	a	Ljava/lang/String	Lnet/comcast/ottlib /v2go/a/b	I_METHOD_PROTOTYPE!I_STATEMENT_FORMAT35c_METHO...
4	b	C	getMessage	I_METHOD_PROTOTYPE!I_STATEMENT_FORMAT35c_METHO...
5	a	TOKEN_REG	v2g.sl.stack_error	I_REGISTER_LIST!I_STATEMENT_FORMAT35c_METHOD!I...
6	a	getAction	Lnet/comcast/ottlib /sms/api/pojo/k	I_STATEMENT_FORMAT35c_METHOD!I_ORDERED_METHOD_...
7	b	cond_15	Ljava/lang /CharSequence	I_LABEL!I_ORDERED_METHOD_ITEMS!I_METHOD?I_ORDE...
8	b	C	Landroid/support /v4/content/o	I_METHOD_PROTOTYPE!I_STATEMENT_FORMAT35c_METHO...
9	a	Landroid/content/Context	Landroid/content/Intent	I_METHOD_PROTOTYPE!I_STATEMENT_FORMAT3rc_METHO...
10	a	const-string	[	I_STATEMENT_FORMAT21c_STRING!I_ORDERED_METHOD_...

Abbildung 5. AST-Pfade mit Start- und Endtoken, generiert aus dem Smali-Code einer Android-App.

Die weiteren Verarbeitungsschritte gestalteten sich wie folgt:

1. Aus allen Anwendungen wurde mithilfe des Werkzeugs apktool[4] der enthaltene Dalvik Bytecode in die sog. Smali-Repräsentation[5] überführt. Der eigentliche Programmcode wird dabei nicht verändert (kein richtiges Dekompilieren), weswegen die in weiterer Folge aus der App gewonnenen AST-Pfade den tatsächlich enthaltenen Programmcode widerspiegeln.

[3] https://archive.org/details/playdrone-apks
[4] https://ibotpeaches.github.io/Apktool/
[5] https://github.com/JesusFreke/smali

2. Mithilfe der in apktool bzw. Smali enthaltenen Grammatik wurde über den Parser ANTLR ein AST für jede Methode jeder Anwendung generiert.

3. In jedem AST wurden daraufhin die Terminale identifiziert und über eine Breitensuche Pfade zu anderen Terminalen gesucht. Diese Pfade erinnern schematisch an jene, die in Abbildung 2 dargestellt sind und werden exemplarisch für den Smali-Code bzw. AST einer Anwendung in Abbildung 5 veranschaulicht. Die vorkommenden Variablennamen repräsentieren dabei die Start- und Endtoken der AST-Pfade. Die Pfadelemente selbst repräsentieren dabei den Weg im AST, der die Terminale miteinander verbindet.

Da das Generieren von ASTs und die Extraktion von AST-Pfaden aus einzelnen Methoden vergleichsweise ressourcenintensiv ist und dementsprechend Zeit beansprucht, wurde das App-Set schließlich auf ca. 2.000 Apps reduziert. Aus diesen Apps wurden etwas mehr als 1,4 Mio. Pfade extrahiert, wobei nur 22.900 verschiedene vorkommen. Das restliche Unterscheidungsmerkmal sind die Variablennamen bzw. Terminale. Rein rechnerisch gibt es also für jeden Pfad im Durchschnitt 61 Kompositionen, bei denen die Terminale unterschiedlich sind.

Nach dem Extrahieren der Pfade gibt es für die Weiterverarbeitung mehrere Freiheitsgrade:

- „Word embeddings“ können sowohl aus den gesamten Pfaden trainiert werden, als auch aus deren Teilelementen (die durch „≥“ und „≤“ voneinander getrennt sind). Welcher Ansatz effektiver ist, kann gegenwärtig nicht zweifelsfrei beantwortet werden.

- Viele Terminale sind selbst Wortkompositionen, wie z.B. „newToken“, „credit_card_number“ oder „OptionsItemSelected“. Anstatt Embeddings direkt für die gesamten Tokens zu lernen, wäre es möglich, sie zu „tokenizen“, also z.B. „newToken“ zu „new“ und „Token“ zu unterteilen, sodass ein zugehöriger Pfad schließlich sowohl mit „new“ als auch mit „Token“ trainiert wird. Eine Aufspaltung in „Subtokens“ könnte vor allem für Einsatzfälle interessant sein, wo es prinzipiell nur wenige Pfade gibt.

- Um aus den Pfaden „Word Embeddings“ zu erhalten, kann beispielsweise ein Training mithilfe von word2vec oder GloVe stattfinden. Der Einfachheit halber und weil für einen sehr großen Datensatz sehr viel Zeit in Anspruch nimmt, wird für das nachfolgende Experiment auf „One Hot Encoding“ zurückgegriffen – also jedem individuellen Token ein eindeutiger numerischer Wert zugewiesen.

Die Vektoren wurden in weiterer Folge mit einem vergleichsweise einfachen Netzwerk trainiert, das aus zwei Schichten besteht, die alle Neuronen miteinander verknüpfen („Dense Layer“). Mit einer allgemein üblichen[6] Auswahl an Hyperparametern wurde folglich ein neuronales Netz trainiert, das die AST-Pfade als „One Hot“-Vektoren trainiert.

An das resultierende Modell wurden in weiterer Folge Pfade übergeben für die eine Klassifikation, also der am besten passende Start- oder Endtoken gefunden werden. Im Testset hat diese Suche aus technischer Sicht funktioniert, aus praktischer jedoch keine verwertbaren Ergebnisse geliefert. Empirisch konnte also keine Verbindung aus Pfad und vorhergesagtem Terminal abgeleitet werden. Aus diesen Ergebnissen können folgende Schlussfolgerungen gezogen werden:

- Die technische Umsetzung und Repräsentation von Code mithilfe von AST-Pfaden funktioniert dahingehend, dass beliebige, selbst konstruierte neuronale Netze damit trainiert werden können.

- Ungeachtet der Praxistauglichkeit des Ansatzes hängt die Vorhersage von Variablennamen zu einem Gutteil von der Wahl und Optimierung von Hyperparametern bzw. dem nachfolgenden neuronalen Modell ab. Im Vergleich zu anderweitig hergeleiteten Vektoren als Input-Features oder der Betrachtung von Programmcode als Text, stellen AST-Pfade im Vergleich eine Form der Repräsentation dar, die die Realität am umfangreichsten abbildet.

[6] http://ruder.io/deep-learning-nlp-best-practices/i

6. Fazit

Im Zuge dieses Projekts wurde der Fokus auf die Verarbeitung von Programmcode durch Methoden aus dem Bereich des maschinellen Lernens gelegt. Das Ziel bestand darin, eine möglichst realitätsnahe Repräsentation von Code-Elementen zu erhalten, die alle wesentlichen syntaktischen und semantischen Eigenschaften abbilden kann.

Nach einer Studie verwandter Arbeiten wurde ein Ansatz entworfen, der sich in Bezug auf den Abstract Syntax Tree an bestehende Überlegungen anschließt. Zur Erreichung der eingangs gesetzten Zielsetzungen wurde jedoch ein alternativer Zugang gewählt, der eine Repräsentation von Codeteilen erlaubt, ohne, dass der Ansatz an Spezifika der dahinterstehenden Programmiersprache angepasst wäre. Die einzige Bindung besteht in der Verwendung eines Parsers, der aus Programmcode einen AST generiert. Die weiterfolgende Verarbeitung zu Pfaden passiert jedoch unabhängig davon. Ebenfalls flexibel gestaltet sich die Aufbereitung der Pfade zu Features für das nachfolgende Training mit Machine Learning-Algorithmen. Neben etablierten und von verwandten Arbeiten verwendeten Methoden wie word2vec oder CRFs kann auch ein „One Hot“-Encoding als einfachste Form verwendet werden.

Die Erkenntnisse aus diesem Projekt zeigen einen innovativen Ansatz, um Programmcode hierarchisch aufzubereiten und diese Hierarchie infolgedessen „flach“ wieder abzubilden. Das dabei verfolgte Konzept kann eingesetzt werden, um die Analyse großer Anwendungen durch Abstraktion zu vereinfachen und gewisse Implementierungsmuster in Programmen wieder zu finden.

Literaturverzeichnis

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

[2] J. Pennington, R. Socher und C. Manning, „Glove: Global Vectors for Word Representation,“ *EMNLP,* 2014.

[3] A. Miltiadis, P. Hao und S. C. A., „A Convolutional Attention Network for Extreme Summarization of Source Code,“ *ICML,* 2016.

[4] A. Miltiadis und C. B. C. A. S. Earl T. Barr, „Suggesting accurate method and class names,“ *ESEC/SIGSOFT FSE,* 2015.

[5] R. Veselin, V. M. T. und K. Andras, „Predicting Program Properties from "Big Code",“ *POPL,* 2015.

[6] B. Benjamin, R. Veselin, T. Petar und V. M. T., „Statistical Deobfuscation of Android Applications,“ *CCS,* 2016.

[7] R. Veselin, B. Pavol und V. M. T., „Probabilistic model for code with decision trees,“ *OOPSLA,* 2016.

[8] B. Pavol, R. Veselin und V. M. T., „PHOG: Probabilistic Model for Code,“ *ICML,* 2016.

[9] H. Abram und Z. S. M. G. P. T. D. Earl T. Barr, „On the naturalness of software,“ *ICSE,* 2012.