Institut für Programmstrukturen und Datenorganisation (IPD)

Lehrstuhl Programmierparadigmen
Prof. Dr.-Ing. Snelting

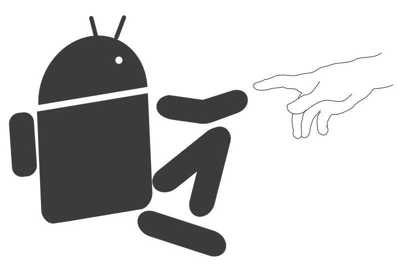
Automatische Modellierung des Lebenszyklus von Android-Anwendungen

Und die Verwendung dieses Modells im Rahmen einer IFC-Analyse

Diplomarbeit von

Tobias Blaschke

an der Fakultät für Informatik



Gutachter	Prof. Dr.-Ing. Gregor Snelting
Betreuender Mitarbeiter:	Dipl.-Inf. Univ. Jürgen Graf
Bearbeitungszeit:	02.10.2013 – 01.04.2013

Hiermit erkläre ich, Tobias Blaschke, die vorliegende Diplomarbeit selbständig und nur mit den genannten Hilfsmitteln verfasst zu haben.

Ort, Datum

Unterschrift

Inhaltsverzeichnis

1 Motivation 1
1.1 Ziel der Arbeit 2
1.2 Umfang der Arbeit 2
1.3 Struktur der Arbeit 3

2 Grundlagen 5
2.1 Repräsentation von Programmen 5
2.1.1 Kontrollfluss 5
2.1.2 Grundblöcke 6
2.1.3 Kontrollabhängigkeit 7
2.1.4 Datenabhängigkeit 8
2.1.5 Programmabhängigkeitsgraphen (PDG) 8
2.1.6 Program Slicing mit dem PDG 9
2.1.7 Definition von Systemabhängigkeitsgraphen (SDG) 10
2.1.8 Program Slicing mit dem SDG 13
2.2 Weitere verwendete Begriffe 14
2.2.1 Aufrufgraphen 14
2.2.2 Points-To-Analyse 14
2.2.3 Typinferenz 18
2.2.4 Static Single Assignment (SSA) 18
2.3 Struktur einer Android-Applikation 19
2.3.1 Ausführungsverhalten von Android-Applikationen 19
2.3.2 Die Bestandteile des Namensraums android.app 20
2.3.3 Einsprungspunkte 20
2.3.4 Lebenszyklus der Einsprungsklassen 21
2.3.5 Intents 25
2.3.6 Kommunikation in Androidanwendungen 26
2.3.7 Dalvik Bytecode 28
2.3.8 APK-Containerformat 30
2.4 T.J. Watson Libraries for Analysis (WALA) 31
2.4.1 Programmrepräsentation mittels WALA 31
2.4.2 Stubs 34

2.4.3 Einlesen von Android-Apps 35
2.5 Joana, Java Object-sensitive Analysis 35
2.5.1 Benutzerschnittstellen Joanas 36
2.5.2 Joanas SDG-Format 38

3 Aufbau des Modells **39**
3.1 Auffinden der Einsprungspunkte 40
3.1.1 Weitere zu berücksichtigende Funktionen 40
3.1.2 Heuristische Selektion von Einsprungspunkten 40
3.2 Instanziierungsverhalten für Parameter 41
3.3 Sortierung der Einsprungspunkte 42
3.4 Grundstruktur des Modells 43
3.5 Nachbearbeitung durch den Anwender 44
3.6 Synthese des Modells zu einem Einsprungspunkt 44
3.7 Anpassung des Starts von Intents 44
3.7.1 Funktion für externe Ziele 45

4 Umsetzung des Modells **47**
4.1 Auffinden der Einsprungspunkte 47
4.1.1 Heuristische Selektion aus der Klassenhierarchie 50
4.2 Instanziierungsverhalten für Parameter 50
4.2.1 Instanziierung der Parameter 51
4.2.2 Verwaltung der einzelnen Instanzen 52
4.3 Sortierung der Einsprungspunkte 52
4.4 Nachbearbeitung gefundener Einsprungspunkte 53
4.4.1 Nachbearbeitung ohne JoDroid 54
4.4.2 Nachbearbeitung mit JoDroid 54
4.5 Synthese des Modells zu einem Einsprungspunkt 56
4.5.1 Erstellung der Umgebung 57
4.5.2 Instanziierung der REUSE-Parameter 57
4.5.3 Instanziierung der CREATE-Parameter 58
4.5.4 Hinzufügen der Einsprungspunkte 58
4.5.5 Einfügen von Sonderbehandlungen 59
4.5.6 Abschließende Arbeiten 61
4.6 Auflösen der Ziele von Intents 62
4.6.1 Kontextfreie Variante 63
4.6.2 Variante mit spezialisiertem Kontext 64
4.6.3 Die Intent-Ziel-Tabelle 65
4.6.4 Generierung eingeschränkter Modelle 66
4.6.5 Generierung von Wrapper-Funktionen für das Modell 67

5 Softwareanpassungen **69**
5.1 Toolbox für Synthetische Methoden in WALA 69
5.1.1 Kapselung von SSA-Variablen 71
5.1.2 Die typsichere Instruction-Factory 72
5.1.3 Zugriff auf Parameter synthetischer Funktionen 73
5.1.4 Verwaltung von SSA-Variablen 73
5.1.5 Reihenfolgeänderung von Instruktionen 74
5.1.6 Variablennamen in Synthetischen Methoden 75
5.1.7 Prüfung der Zuweisbarkeit von Primitiven 75
5.1.8 Instanziierung von Variablen 75
5.2 Weitere Anpassungen an WALA 76
5.2.1 GoTo-Befehl für Synthetische Methoden 77
5.2.2 Einlesen Androids Manifest-Datei 77
5.2.3 AndroidEntryPoint 77
5.2.4 Aufrufkontext für Start-Funktionen 78
5.3 Anpassungen an Joana 79
5.3.1 Aufbereitung der Fortschrittsinformationen 79
5.3.2 Umbenennung anonymer Variablen 79
5.4 Hilfsprogramme 79
5.4.1 Generierung erweiterter Stubs 80
5.4.2 Auffinden nicht aufgelöster Aufrufe 80
5.4.3 Konvertierung des Formates SuSis zu Joana 80

6 Analyse **83**
6.1 Analyseeinstellungen 83
6.1.1 Einstellungen WALAs 83
6.1.2 Konfiguration der Lebenszyklusmodellierung 85
6.1.3 Einstellungen bezüglich des Systemabhängigkeitsgraphen . . 86
6.1.4 IFC-Einstellungen 87
6.2 Vorgehen bei den Analysen 88
6.3 Vergleich mit alternativen Programmen 88
6.4 Analyse der Anwendungen 89
6.4.1 DroidBench 89
6.4.2 AllComponentsTest 94
6.4.3 Insecure Bank 97

7 Probleme **103**
7.1 Probleme während der Implementierung 103
7.1.1 Verwendung von FakeRoot-Methoden 103
7.1.2 Instanziierung von Variablen 104
7.1.3 Zyklische Abhängigkeiten von Typen 104
7.1.4 Erkennung der Ursache von Exceptions 104
7.1.5 Extraktion von AndroidManifest.xml 105

7.2 Generelle Fragestellungen zur Analysierbarkeit von Android Applikationen . . . 105
7.2.1 Berücksichtigung pseudo-externer Abhängigkeiten . . . 106
7.2.2 Behandlung eingehender Intents . . . 106
7.2.3 Direkte Speicherzugriffe . . . 107
7.2.4 Geschlossene Analysierbarkeit einer einzelnen Applikation . . 107

8 Offene Arbeiten 109
8.1 Ideen zur Verbesserung der Genauigkeit . . . 109
8.1.1 Verwendung von Komponenten externer Applikationen . . . 109
8.1.2 Umgang mit impliziten Intents . . . 110
8.1.3 Verfeinerung von Instanziierungsverhalten . . . 110
8.1.4 Genaueres Abbilden weiterer Lebenszyklen . . . 111
8.1.5 Auswertung Androids Kontextinformationen . . . 111
8.1.6 Auswertung von Elementen der Benutzerschnittstelle . . . 111
8.2 Unbehandelte Aufgaben . . . 112
8.2.1 Berücksichtigung des Originals bei Überschreibungen . . . 112
8.2.2 Verbesserung der Auflösung von Intents . . . 112
8.2.3 Erweiterung des Scanvorgangs . . . 113
8.2.4 Einlesen weiterer Analyseparameter . . . 113

9 Fazit 115

A Liste der registrierten Einsprungspunkte 117
A Einsprungspunkte der Grundkomponenten . . . 117
A.1 android.app.Application . . . 117
A.2 android.content.ContentProvider . . . 118
A.3 android.app.Activity . . . 118
A.4 Fragment . . . 123
A.5 android.app.Service und BroadcastReceiver . . . 125

B Weitere Klassen für Einsprungspunkte 131

C Standartwerte des Instanziierungsverhaltens 133

D Funktionen zum Start von Intents 135

E JoDroids Einsprungspunktdatei 139

F Einstellmöglichkeiten des Modells 147

G Analyseeinstellungen von Insecure Bank 151

KAPITEL 1

Motivation

Die meisten Programme verfügen mit ihrer *main-Methode* über einen einzigen Einsprungspunkt[1]. Für die Analyse des Verhaltens eines solchen Programmes ist es somit hinreichend die Effekte der Ausführung dieser Methode zu betrachten.

Anders als die vorgenannten Programme weisen Anwendungen für Android eine besondere innere Struktur auf: Einzelne Fenster und auch Dienste dieser Anwendungen sind derart hinterlegt, dass sie unabhängig voneinander gestartet, pausiert und beendet werden können. Dies geschieht durch die Spezifikation mehrerer Einsprungspunkte, die im Lebenszyklus[2] eines Programmes in einer gewissen Reihenfolge durch das Betriebssystem angesprungen werden können.

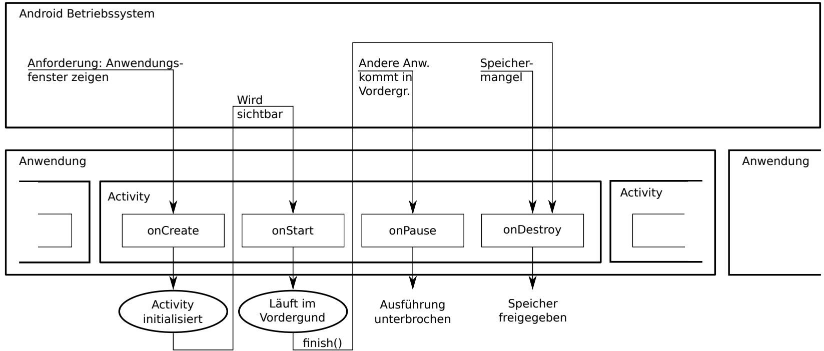


Abbildung 1.1: *Auszug aus dem Lebenszyklus einer Activity einer Android-Applikation*

Bereits für ein einzelnes Fenster eines Programmes existieren in Android mehrere Einsprungspunkte: Über sie ist es möglich beispielsweise abhängig von dem Status

[1] Als Einsprungspunkt bezeichnet man eine Stelle innerhalb eines Programmes, an der dessen Ausführung begonnen wird.

[2] Der Lebeszyklus einer Anwendung umfasst Vorkehrungen zum Start des Programmes, die Ausführung selbst und endet mit dem Entfernen des Programmes aus dem Speicher.

der Anzeige dieses Fensters unterschiedliche Aktionen in dem Programm auszuführen. Abbildung 1.1 zeigt einen Auszug aus Einsprungspunkten eines Fensters; in Unterabschnitt 2.3.4 wird genauer darauf eingegangen, welche Einsprungspunkte bei Android verfügbar sind.

Das Problem der Analyse von Android-Anwendungen liegt nun darin, dass viele Werkzeuge auf das Vorhandensein genau eines Einsprungspunktes ausgelegt sind. Das Ergebnis der Analyse der gesamten Anwendung umfasst in der Regel aber mehr Informationen als die Summe der Analysen ihrer einzelnen Einsprungspunkte[3].

1.1 Ziel der Arbeit

Das Verhalten von Android-Anwendungen soll derart gekapselt werden, dass ein einziger virtueller Einsprungspunkt generiert wird. Dazu muss zunächst ein Modell geschaffen werden, das derart auf die Applikation einwirkt, dass Android-Einsprungspunkte in geeigneter Form, Reihenfolge und Wiederholung aufgerufen werden, wie es einem typischen Lebenszyklus entspricht. Weiterhin sind dabei Abhängigkeiten zwischen den einzelnen Komponenten einer Applikation zu berücksichtigen.

Aus diesem Modell wird anschließend der virtuelle Einsprungspunkt generiert, der als Ausgangspunkt für eine Programmanalyse mit bestehenden Werkzeugen herangezogen werden kann. Dadurch ist es möglich mit nur geringen Anpassungen an den Werkzeugen selbst, eine Unterstützung für Android-Applikationen zu bieten.

1.2 Umfang der Arbeit

Möchte man Programme analysieren, so überführt man sie dazu zunächst meist in passendere Darstellungsformen. Diese sind weitgehend unabhängig von der Quellsprache, können teilweise sogar aus kompilierter Software generiert werden. Eine Bibliothek, die für solche Konvertierungen genutzt werden kann, ist *WALA* [11]. *WALA* bildet somit die Grundlage für mehrere Analysewerkzeuge.

Die Generierung vorgenannten Modells und die Synthese des virtuellen Einsprungspunktes erfolgt in *WALA*. Somit wird es *WALA*-basierten Werkzeugen ermöglicht statische Programmanalysen auf kompletten Android-Applikationen durchzuführen (*Whole Program Analyse*). Ein Werkzeug, welches von diesen Anpassungen profitiert findet sich in *Joana*.

Joana [18] ist ein aktuelles Forschungsprojekt des KIT. Ziel dieses Projektes ist es, durch genaue Abbildung der Informationspfade innerhalb eines Programmes, Aussagen bezüglich dessen Sicherheit treffen zu können. Eine auf Android angepasste Variante *Joanas* findet sich in *JoDroid*.

[3]Da zwischen den einzelnen Funktionen beispielsweise auch Daten auf dem Heap-Speicher ausgetauscht werden können, was bei dieser Vorgehensweise nicht berücksichtigt würde.

Das Verhalten des im Umfang dieser Arbeit entstandenen Modells wird abschließend anhand der Analyse von Programmen mittels *JoDroid* untersucht.

1.3 Struktur der Arbeit

Im ersten Kapitel wird zunächst auf Repräsentationsformen von Programmen eingegangen (siehe Abschnitt 2.1), welche als Grundlage für eine Analyse mittels *WALA* (siehe Abschnitt 2.4) und *Joana* (siehe Abschnitt 2.5) wichtig sind. Weiterhin wird dort detailliert der den Aufbau von Android-Applikationen beschrieben (siehe Abschnitt 2.3).

Die Erstellung des Lebenszyklusmodells wird in Kapitel 3 zunächst in allgemeingültiger Form behandelt, Kapitel 4 befasst sich anschließend mit der Umsetzung in *WALA*.

Weitere Anpassungen an *WALA* und *Joana* welche im Zuge der Umsetzung notwendig wurden werden in Kapitel 5 beschrieben.

Schließlich findet das erstellte Modell in Kapitel 6 Anwendung in der Untersuchung einiger ausgewählter Applikationen bezüglich ihrer Sicherheit.

Abschließend wird auf Probleme, welche während der Umsetzung auftraten, und weitere Verbesserungsmöglichkeiten eingegangen, um schließlich ein Fazit ziehen zu können.

KAPITEL 2

Grundlagen

In diesem Kapitel werden zunächst in Abschnitt 2.1 einige abstrakte Darstellungsformen von Programmen vorgestellt. Weitere im späteren verwendete Begrifflichkeiten werden in Abschnitt 2.2 definiert; anschließend wird in Abschnitt 2.3 ein kurzer Überblick über die Struktur von Android-Anwendungen gegeben. Schließlich werden in Abschnitt 2.4 und Abschnitt 2.5 die verwendeten Werkzeuge *WALA* und *Joana* vorgestellt.

2.1 Repräsentation von Programmen

Eine Darstellungsform von Programmen, welche unter anderem bei *Joana* als Zwischendarstellung Verwendung findet, bildet der sogenannte *Systemabhängigkeitsgraph* (SDG - System Dependence Graph). In diesem Kapitel sollen zunächst die Grundlagen zum Verständnis eines SDG geschaffen werden um schließlich dessen Funktionsweise selbst zu erklären.

Zur exemplarischen Verdeutlichung der dargestellten Konstrukte wird im Folgenden der in Listing 1 dargestellte Sourcecode zugrunde gelegt.

2.1.1 Kontrollfluss

Die Reihenfolge, in der Anweisungen eines Programms ausgeführt werden, nennt man *Kontrollfluss*. Dieser lässt sich ein einem Graphen darstellen: die Knoten des Graphen bilden die Anweisungen; die Kanten legen die Ausführungsreihenfolge fest [10]. Ein Knoten hat genau dann mehr als eine Ausgangskante, wenn es sich bei ihm um ein Prädikat mit Ziel der Kontrollflussänderung (beispielsweise eine *if*-Anweisung) handelt. Oft wird in den Graphen ein virtueller Start- und Endknoten eingefügt.

```
int i = 5;
int j = 3;
if ( i > j ) {
        i = j;
        j = 3;
} else {
        j = 2;
}
print ( j );
j = 1;
print ( j );
```

Listing 1: *Sourcecode der späteren Beispielgraphen*

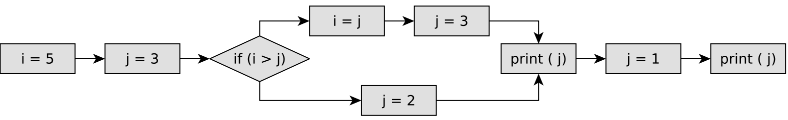


Abbildung 2.1: *Programmablaufplan mit Kontrollflusskanten*

2.1.2 Grundblöcke

Als *Grundblock* [3] bezeichnet man zusammenhängende Bereiche eines Programms, in denen keine Anweisungen in Verzweigungen liegen. Grundblöcke weisen einen festen Start- und Endpunkt auf. Dieser kann zwar als virtueller Knoten eingefügt werden, meist wird er aber mit einem vorhandenen Knoten zusammengelegt.

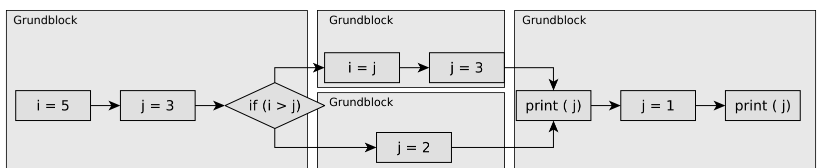


Abbildung 2.2: *Programmablaufplan mit eingezeichneten Grundblöcken*

Immer wenn im Verlauf des Programms der Startknoten ausgeführt wird, so wird jede Anweisung des selben *Grundblocks* (wenn auch nicht zwangsläufig direkt) genau einmal ausgeführt. Befindet sich unter diesen Anweisungen eine solche, die eine Verzweigung einleitet (beispielsweise ein *if*) so bedeutet das, dass je ein weiterer *Grundblock* für die Anweisungen auf den einzelnen Pfaden existiert. In einem

sogenannten *erweiterten Grundblock* kann dann an den Verzweigungsstellen in den *Grundblock* des Verzweigungspfades gesprungen werden.

Grundblöcke lassen sich aus dem *Kontrollflussgraph* durch Berechnung sogenannter *Dominatoren* [24] erstellen.

2.1.3 Kontrollabhängigkeit

Eine weitere Darstellung, welche durch die Berechnung mittels *Dominatoren* aus dem *Kontrollflussgraphen* generiert werden kann, bildet der *Kontrollabhängigkeitsgraph* [27] (Control Dependence Graph – CDG). An ihm lassen sich unter anderem direkt die *Grundblöcke* erkennen.

Kontrollabhängigkeiten fußen zunächst grundsätzlich im Startknoten des Graphen oder in einem Prädikat [22]. Eine Kontrollabhängigkeit liegt genau dann vor, wenn es auf dem *Kontrollflussgraph* ein Pfad ausgehend vom Startknoten zu dem in Frage stehenden Knoten existiert und beide Knoten dem selben *Grundblock* angehören.

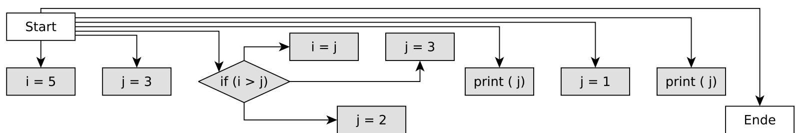


Abbildung 2.3: *Kontrollabhängigkeitsgraph (CDG) mit expliziten Start- und Endknoten*

Eine Kontrollabhängigkeit zwischen zwei Knoten zeigt an, dass die Ausführung des ersten Knotens die Ausführung des zweiten Knotens verhindern kann [4]. Weiterhin sagt sie aus, dass bei Ausführung eines Knotens – unter Berücksichtigung einer optionalen Bedingung – immer auch alle Vorgänger ausgeführt werden, wenn auch nicht zwangsläufig unmittelbar zuvor [22].

Anwendung finden *Kontrollabhängigkeiten* beispielsweise in der Optimierung. Zudem bilden sie die Voraussetzung für den *Programmabhängigkeitsgraph* (siehe Unterabschnitt 2.1.5) und damit den *Systemabhängigkeitsgraph.*

In Abbildung 2.3 wurden Start- und Endknoten explizit aufgeführt. Häufig werden diese virtuellen Knoten jedoch mit vorhandenen Knoten zusammengelegt. In obigem Beispiel würden der Startknoten mit dem Knoten "$i = 5$", der Endknoten mit dem rechten "*print(j)*" zusammenfallen.

In der Praxis lassen sich *Kontrollabhängigkeiten* oft nicht so leicht wie in obigem Beispiel ablesen: Durch das mögliche Auftreten von *Exceptions* werden implizit weitere *Prädikate* induziert. Ausgehend von diesen *Prädikaten* existieren dann zusätzliche Kanten zu einem eventuell vorhandenen *Catch-Block* oder zu einem alternativen Ausgang einer Funktion.

2.1.4 Datenabhängigkeit

Eine *Datenabhängigkeit* [27] entsteht, wenn eine Anweisung auf einen Wert zugreift, der einer anderen Anweisung generiert wurde. Greifen zwei Anweisungen nur lesend auf einen Wert zu, so existiert zwischen ihnen keine *Datenabhängigkeit*.

Voraussetzung für das Vorhandensein einer *Datenabhängigkeit* ist die Existenz eines Pfades auf dem *Kontrollflussgraph*.

Datenabhängigkeiten lassen sich wieder als Graph darstellen. Dieser Graph nennt sich *Datenabhängigkeitsgraph (Data Dependence Graph – DDG)*.

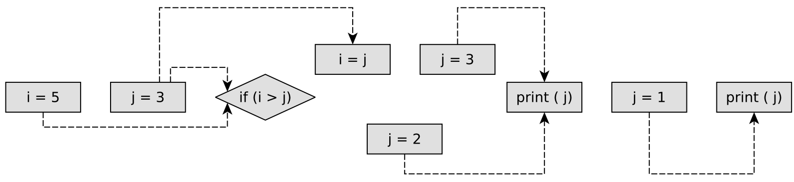


Abbildung 2.4: *Datenabhängigkeitsgraph (DDG)*

2.1.5 Programmabhängigkeitsgraphen (PDG)

Durch Kombinieren der beiden vorgenannten Graphen lässt sich der *Programmabhängigkeitsgraph (PDG)* [26] [13] erstellen.

Nach Definition in [22] sind in einem PDG zunächst nur skalare Variablen, Zuweisungen, Bedingte Anweisungen und *while*-Schleifen zugelassen. Der Graph selbst ist von gerichteter Natur, wobei es verschiedene Typen von Kanten gibt: Die Datenabhängigkeit, die unbedingte Kontrollabhängigkeit und die bedingte Kontrollabhängigkeit, welche mit einem *Label* zur Beschreibung des Verhaltens versehen ist.

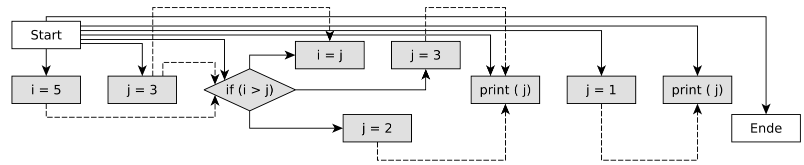


Abbildung 2.5: *Programmabhängigkeitsgraph (PDG)*

Es ist zu berücksichtigen, dass in dieser ursprünglichen Definition der PDGs die zugrundeliegende Sprache stark eingeschränkt ist. Insbesondere sind keine Funktionsaufrufe erlaubt. Somit eignet sich dieser Graph nur zur Abbildung *Intraprozeduralen*

Verhaltens. Diese Einschränkung wird in den später folgenden *Systemabhängigkeitsgraphen* (siehe Unterabschnitt 2.1.7) angegangen.

2.1.6 Program Slicing mit dem PDG

Ein *Slice* [38] kann bezüglich eines Punktes der Programmausführung (p) auf dem *Programmabhängigkeitsgraph* gebildet werden[1]. Es handelt sich dabei um die Menge aller Anweisungen dieses Programms, welche nach der Ausführung von p den Wert der dort vorkommenden Variablen X potentiell beeinflussen können [22].

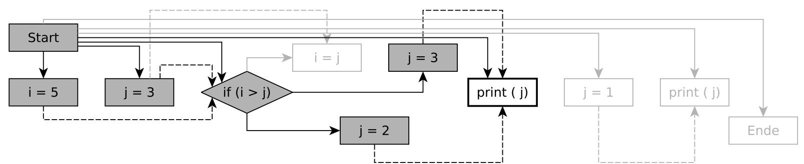


Abbildung 2.6: *Backward-Slice zu Punkt "print(j)" und Variable "j"*

Liegen Anweisungen, welche als *Slice* selektiert wurden innerhalb eines verzweigten Kontrollfluss, so ist das Prädikat, das die Verzweigung einleitet, ebenfalls zu der Menge des *Slices* hinzuzufügen [26].

Analog zu einem *Slice* definiert man den *Backward-Slice* als alle Anweisungen eines Programms, welche Variablen X beeinflussen können und zwischen Programmstart und einer Anweisung p liegen. Zur expliziten Differenzierung wird der anfangs genannte *Slice* auch *Forward Slice* genannt.

Abbildung 2.6 zeigt den *Backward-Slice* (dunkelgrau hinterlegt) des fett umrandeten Programmpunktes p – "*print(j)*".

Die Berechnung des *Backward-Slices* nach Ottenstein und Ottenstein [26] traversiert den *Programmabhängigkeitsgraph* wie folgt: Es werden zu einem Knoten solange alle eingehenden *Kontroll-* und *Datenabhängigkeiten* zurückverfolgt, bis sie eine Konstante oder eine Eingabe-Anweisung[2] beinhalten. Alle dabei besuchten Knoten werden zur Menge des *Slices* hinzugefügt.

Berechnung eines interprozeduralen Slices

Einen ersten Ansatz für die *interprozedurale Berechnung* lieferte Weiser [38]. Diese verläuft in zwei Schritten: Zunächst wird der *Slice intraprozedural* in der Prozedur berechnet, welche das Slicing-Kriterium (also den Programmpunkt p) enthält. Bezüglich des Aufrufs von Prozeduren werden sogenannte *Summary*-Informationen dieser Prozedur genutzt, allerdings wird die Prozedur für die Berechnung nicht "betreten".

[1] zumindest im intraprozeduralen Fall

[2] Gemeint sind hier Anweisungen, welche Daten von der Peripherie lesen zum Beispiel Tastatureingaben, Netzwerkdaten oder aus Dateien

Im zweiten Schritt wird anschließend ein neues Slicing-Kriterium für jede Variable x_n in jeder Prozedur p_n, welche eine Variable in X beeinflussen könnte, generiert. Anschließend springt man in den ersten Schritt zurück, solange bis keine neuen Kriterien mehr erzeugt werden. Dies wird meist als *Worklist-Agrorithmus* umgesetzt.

Bei dieser *Fixpunktiteration* handelt es sich um eine *konservative Approximation*[3] der Berechnung des *Slices*: Der Kontext der einzelnen Aufrufe wird hier wenig berücksichtigt.

Um das Ergebnis des *Interprozeduralen Slicings* zu präzisieren wird in Horwitz [22] ein zunächst ähnlicher Ansatz wie in Ottenstein [26] verfolgt. Allerdings wird dort der *Systemabhängigkeitsgraph* (siehe Unterabschnitt 2.1.7) eingeführt, welcher mehr Informationen über Aufrufumgebungen enthält. Eine Beschreibung des Slicing unter Verwendung des *Systemabhängigkeitsgraphen* findet sich in Unterabschnitt 2.1.8.

2.1.7 Definition von Systemabhängigkeitsgraphen (SDG)

Ein Problem bei der Erstellung von *Slices* aus *Programmabhängigkeitsgraphen* entsteht, wenn in dem Quellprogramm Funktionen definiert sind: In der ursprünglichen Definition der *PDG* waren Funktionsaufrufe noch nicht vorgesehen.

Zur Darstellung *Interprozeduralen* Verhaltens mittels eines *SDG* wird zunächst für jede Prozedur ein *PDG* erstellt. Dieser wird daraufhin an den Stellen, an denen Prozeduren aufgerufen werden – den sogenannten *Call-Sites* – erweitert: Enthielt der *PDG* (siehe Kapitel 2.1.5) nur direkte Abhängigkeiten, so besteht einer Ergänzung des *SDG* in durch Prozeduraufrufe induzierte transitive Datenabhängigkeiten [22]. Für diese Ergänzungen werden zunächst neue Typen von Knoten zum Zusammenfassen des Programmzustands[4], sowie Kanten zur Übergabe dieses Zustandes in die Prozeduren hinein induziert. Durch diese Knoten werden die jeweiligen Aufrufstellen, sowie Funktionseinsprungspunkte umschlossen.

Schließlich wird das Verhalten von Funktionen durch *Summary-Kanten* zusammengefasst.

Joana erweitert das Konzept der ursprünglichen SDGs durch feingranularere Kantentypen, sowie weiteren Attributen. Eine genauere Betrachtung des in *Joanas* verwendeten *SDGs* findet sich in Unterabschnitt 2.5.2.

Komplettes Beispiel eines SDG

Um ein Beispiel eines kompletten SDG zu geben muss das zuvor verwendete Beispiel noch um die Implementierung der *print()*-Funktion erweitert werden. Da der SDG in dem Fall, dass das Programm auf einem PC ausgeführt wird noch um weiteren

[3]Die *Konservative Approximation* umfasst mehr potentielle Codezeilen als nötig wären, um die Abhängigkeiten am fraglichen Programmpunkt aufzulösen.

[4]Durch eine *Umgebung* werden Variablen und der Programmzustand (insbesondere der *Instruction Pointer*) zusammengefasst. Neben der *Globalen Umgebung* existiert bei modernen Sprachen eine Umgebung je Funktion. Diese wird auf Low-Level-Ebene durch ein *Activation Record* (*Stack Frame*) realisiert.

Initialisierungscode und komplexeres Verhalten erweitert werden müsste, ist der Code zur direkten Ausführung (ohne Betriebssystem) auf einem *Atmel AVR-8* Mikrocontroller ausgelegt.

Der nun komplette Source-Code findet sich in Listing 2. Die *print()*-Funktion gibt eine Integer-Zahl (rückwärts und ASCII-Codiert) auf einer Seriellen Schnittstelle (*UART* - Universal Asynchronous Receiver Transmitter) des Mikrocontrollers aus.

```
/* UCSRA and UDR are memory - adressed registers
   UDRE is a # defined constant */
int print ( int v ) {
        int pos = 0;
        while ( v > 0) {
                while (!( UCSRA & (1 < < UDRE ))); // wait until UART-Ready
                UDR = ( char )( '0' + ( v % 10));   // write to UART
                v /= 10;
                pos ++;
        }
        return pos ;
}

void main ( void ) {
        int i = 5;
        int j = 3;
        if ( i > j ) {
                i = j;
                j = 3;
        } else {
                j = 2;
        }
        print ( j );
        j =1;
        print ( j );
}
```

Listing 2: *Sourcecode des SDG-Beispiels*

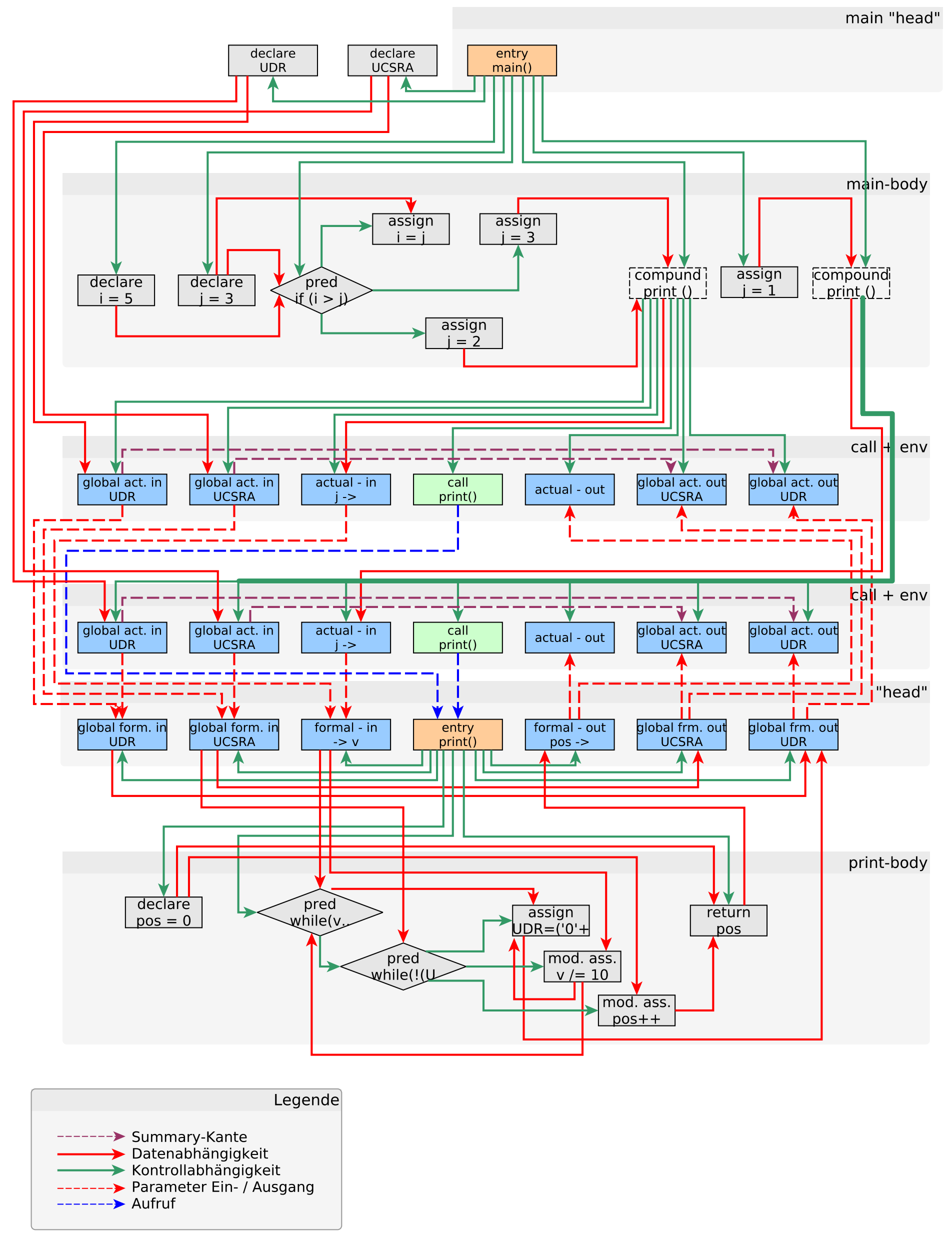


Abbildung 2.7: *Beispiel eines Systemabhängigkeitsgraphen*

Das Beispiel stellt den SDG des gegebenen Programms dar: Zu erkennen sind Kanten für Datenabhängigkeiten (Rot) und Kontrollabhängigkeiten (Grün) sowie Parameter-Kanten und schließlich Aufruf-Kanten (Blau).

Knoten besitzen neben ihrem Inhalt einen Knotentyp. Hervorzuheben sind hier die im *Systemabhängigkeitsgraph* neu eingeführten Knoten der Typen *Actual-In* und *Actual-Out*. Über sie sind Änderungen an Parametern einer Funktion nachvollziehbar. Zwischen den jeweiligen *Actual-In-* und *Actual-Out*-Kanten ist eine *Summary*-Kante aufgespannt. Diese Kanten fassen das Verhalten der aufgerufenen Funktion zusammen.

Die Knoten *Formal-In* und *Formal-Out* dienen zur Übergabe der Werte des jeweiligen *Actual*-Pendants in die Funktion hinein, beziehungsweise aus der Funktion heraus. Entsprechende Knoten sind mit Parameter-Kanten verbunden.

Die in der Graphik auftretenden Knoten des Typs "*compound*" tragen keinerlei inhaltliche Bedeutung: Sie wurden lediglich für die optische Ähnlichkeit zum ursprünglichen Beispiel eingeführt. Betrachtet man die ein- und ausgehenden Kanten gleichen Typs als verbunden, kann man über diese Knoten hinweglesen. Die breite ausgehende Kontrollabhängigkeitskante soll ein Hinweis geben, dass hier mehrere Kanten übereinander liegen. Diese Darstellung wurde aus Platzgründen gewählt.

2.1.8 Program Slicing mit dem SDG

Wie in Unterabschnitt 2.1.6 angedeutet wird bei der Methode des *interprozeduralen Slicings* nach Weiser [38] der Kontext der Funktionsaufrufe nicht ausreichend berücksichtigt. Im Gegensatz dazu arbeitet das Verfahren nach Horwitz [22] auf einem *Systemabhängigkeitsgraph* und weist ein anderes Vorgehen im Umgang mit auftretenden Funktionsaufrufen auf:

Im ersten Schritt werden *Kontrollfluss-*, *Kontrollabhängigkeits-* und *Datenabhängigkeitskanten* sowie *Parameter-in*-Kanten, jedoch keine *Parameter-Out*-Kanten verfolgt. In diesem Schritt werden somit alle Knoten selektiert, die einen gegebenen Programmpunkt p erreichen können und in der Prozedur P dieses Punktes oder in einer Prozedur, die diese aufruft liegen. Durch die Summary-Kanten werden Effekte von Funktionsaufrufen, die in P liegen nicht komplett ignoriert.

Der zweite Schritt berücksichtigt *Kontrollfluss-*, *Kontrollabhängigkeits-* und *Datenabhängigkeitskanten* sowie *Parameter-out-Kanten*, jedoch keine *Parameter-in-Kanten*. In diesem Schritt werden alle Funktionen betrachtet, die ausgehend von Funktionen die in Schritt eins selektiert wurden aufgerufen werden können (also alle *Callees* von P und alle *Callees* der *Caller* von P). Diese Funktionen werden nun daraufhin untersucht, ob sie den fraglichen Programmpunkt erreichen können [22].

Beruhte das ursprüngliche Verfahren also auf einer Vereinigung der Effekte aller Funktionen, die Einfluss auf p nehmen können (es wurde jeweils ein neues *Slicing-Kriterium* generiert), so berücksichtigt das neue Verfahren nun, inwiefern Werte in einem gegebenen Kontext überhaupt modifiziert werden können.

2.2 Weitere verwendete Begriffe

Führten die bisher dargestellten Grundlagen aufeinander aufbauend den Systemabhängigkeitsgraphen ein, werden nun zur Vervollständigung die restlichen benötigten Begrifflichkeiten erklärt.

2.2.1 Aufrufgraphen

Ein *Aufrufgraph* [28] besteht aus Funktionsnamen als Knoten. Zwischen zwei Knoten wird genau dann eine gerichtete Kante gezogen, wenn ausgehend von einer Funktion eine andere aufgerufen werden kann.

Die Erstellung dieses Graphen scheint zunächst trivial. Die Komplexität erhöht sich jedoch deutlich, sobald Referenzen auf Funktionen als Parameter beziehungsweise in globalen Variablen auftauchen können. Objektorientierte Sprachen weisen ein solches Verhalten ebenfalls implizit auf. Zur Lösung dieses Problems müssen alle Funktionen, des Aufrufgraph daraufhin untersucht werden, auf welche Prozeduren sie Referenzen erzeugen. Es kann hierfür eine *Points-To Analyse* (siehe Unterabschnitt 2.2.2) verwendet werden.

Funktionen des Quellprogramms können unter Umständen durch mehrere Knoten auf dem *Aufrufgraph* repräsentiert werden. Dies ist beispielsweise dann sinnvoll, wenn durch die Funktion ein generisches Verhalten – wie das Hinzufügen von Objekten zu einer Liste – zur Verfügung gestellt wird. In der späteren Analyse wird man dann ein genaueres Ergebnis erhalten, wenn die Funktion für die einzelnen Objekttypen getrennt betrachtet wird.

Die Berechnung des *Aufrufgraphen* erfolgt – aufgrund der verwandten Aufgabenstellung – in der Praxis meist zeitgleich mit der im Folgenden betrachteten *Points-To Analyse*.

2.2.2 Points-To-Analyse

Eine *Points-To-Analyse* ist eine statische Programmanalyse zur Feststellung auf welche Speicherstellen eine Referenz-Variable zeigen kann. Weiterhin soll durch diese Analyse ermittelt werden, welche Variablen *Aliase* zueinander sind, also die selbe Speicherstelle referenzieren können. Die Resultate einer *Points-To-Analyse* werden in der Regel in einem sogenannten *Points-To-Graph* abgelegt. Die Knoten dieses Graphen werden aus Referenzen und Objekt-Instanzen gebildet. Kann eine Referenz auf eine Instanz zeigen, so wird eine ungerichtete Kante in dem Graph gezogen. Das genaue Aussehen des Graphen hängt von dem für die Berechnung herangezogenen Algorithmus ab.

Zwei grundlegende Algorithmen für die Berechnung von Points-To Informationen finden sich in dem *Algorithmus von Andersen* [5] und dem *Algorithmus von Steensgaard* [33]. Beide basieren auf einem Satz abstrakter Ortsangaben, unterscheiden sich aber darin, wie Zuweisungen behandelt werden.

Steensgaards Algorithmus

Bei der Verarbeitung einer Zuweisung setzt dieser Algorithmus die abstrakten Ortsangaben beider Operanden auf die Vereinigung ihrer vorherigen Ortsangaben.

Dieser Algorithmus zeichnet sich durch seine sehr kurze Laufzeit aus, ist jedoch recht unpräzise.

Andersens Algorithmus

Dieser Algorithmus modifiziert lediglich die Ortsangaben der linken Seite der Zuweisung. Auch hier werden diese wieder auf die Vereinigung der Ortsangaben beider Seiten gesetzt. Die Ortsangaben der rechten Seite bleiben jedoch unverändert.

Das Ergebnis dieses Algorithmus ist somit deutlich präziser bei höherer Laufzeit.

Aufgrund der Verwandtheit der Aufgabenstellung erfolgt die Berechnung der *Points-To* Informationen häufig zeitgleich mit der Erstellung des *Aufrufgraphen* einer Anwendung.

Algorithmen zur Berechnung von Aufrufsgraphen weisen eine jeweils unterschiedliche Präzision auf. Es existieren folgende Arten von Präzision:

klassen-/typbasiert

Diese einfachste Form von Analysen unterscheidet Variablen anhand ihres konkreten Typs beziehungsweise anhand des Namens ihrer Klasse.

Die Algorithmen sind sehr schnell jedoch auch unpräzise.

aufruferabhängig

Aufruferabhängige Algorithmen berücksichtigen bei dem Aufruf einer Funktion jeweils die aufrufenden Funktionen. Der Pfad der jeweils genommenen Aufrufer wird in einem sogenannten *Call-String* hinterlegt [30]. Die meisten Algorithmen begrenzen die maximale Länge des verwendeten *Call-Strings.*

instanzabhängig

Diese Algorithmen weisen mehr Informationen bezüglich der bei einem Aufruf vorkommenden Parameter (insbesondere auch des *this-Pointers* objektorientierter Sprachen) auf: Sie werden anhand der Stelle ihrer Erstellung (eigentlich der Allokation von Speicher) unterschieden. Diese Information wird auch als *Creation-String* [1] bezeichnet.

objektsensitiv

Objektsensitive Algorithmen können Aufrufe anhand eines Zielobjekts unterscheiden und erlauben so eine exaktere Analyse, wenn das analysierte Programm polymorphe Klassen enthält.

Objektsensitive Analysen haben einen sehr hohen Speicherverbrauch und eine hohe Laufzeit.

flusssensitiv

Solche Analysen können zwischen verschiedenen Zuständen einer Variable abhängig von dem Punkt innerhalb des Kontrollflusses, an dem auf sie zugegriffen wird, unterscheiden.

Flusssensitive Analysen weisen einen sehr hohen Verbrauch an Speicher und Rechenzeit auf, sodass sie derzeit in der Praxis kaum Verwendung finden.

Als *kontextsensitiv* werden in der Regel Algorithmen bezeichnet, die einen *Call-String* verwenden. Algorithmen, die keinen *Call-String* aber dennoch einen *Creation-String* verwenden, werden häufig als *kontextfrei* bezeichnet, obwohl mit dem *Creation-String* eine Art von Kontextinformationen vorliegt.

Generell ist die Berechnung des Aufrufsgraphen unentscheidbar [21], lässt sich also nur näherungsweise beantworten. Einige Algorithmen, welche eine korrekte (im Sinne von *sound*) Annäherung erzeugen, sowie ihre häufigst verwendeten Konfigurationen seien im Folgenden betrachtet.

Kontrollflussanalyse (CFA)

Bei einer *Kontrollflussanalyse* [29] (*Control Flow Analysis - CFA*) wird ausgehend von Variablen, Statements und Expressions des höchsten Levels (beispielsweise der *main()*-Funktion) ein Graph erzeugt. Wird während dieser Propagation eine Aufrufstelle (oder eine *new*-Anweisung) gefunden, so wird im Graph eine Kante zu dem Ziel des Aufrufes gezogen und, so noch nicht vorhanden, der Graph dieses Ziels erzeugt.

Während der Ausführung einer *Kontrollflussanalyse* können Kontextinformationen für die einzelnen Aufrufe generiert werden. Je nach Art des erstellten Kontexts werden der Bezeichnung der Analyse zwei Zahlen vorangestellt, man spricht von einer *k-l-CFA*. Teilweise wird die zweite Zahl l auch nicht explizit aufgeführt. Durch k wird die maximale Länge des sogenannten *Call-Strings* festgelegt. CFA-Varianten mit einem k-Wert größer null sind somit *aufruferabhängig* und werden deshalb als *kontextsensitiv* bezeichnet. Durch l wird die Länge der *Creation-Strings* angegeben. Ist dieser Wert größer null, so ist eine Analyse *instanzabhängig*.

Im Folgenden seien einige häufig verwendete Kombinationen von Werten für k und l angegeben:

0-CFA ($k = l = 0$)

Diese 1988 von Shivers [29] beschriebene Form einer *CFA* zeichnet sich vor

allem durch eine kurze Laufzeit aus. Eine solche Analyse ist *kontextfrei* und *typbasiert*.

0-1-CFA $(k = 0; l = 1)$
Eine *0-1-CFA* [19] berücksichtigt eine *Allocation-Site* und ist somit *instanzabhängig* jedoch nicht *aufruferabhängig*. Sie wird somit als *kontextfrei* bezeichnet. Ihr Resultat entspricht weitgehend dem einer *Andersens-Analyse* [5].

Trotz der deutlich höheren Genauigkeit weist die *0-1-CFA* eine recht kurze Laufzeit auf.

0-1-C-CFA
Diese Erweiterung [14] der *0-1-CFA* konstruiert als Besonderheit genauere Kontextinformationen für Container-Objekte. Sie stellt somit eine Mischform zwischen einer rein *instanzabhängigen* und einer *objektsensitiven* Analyse dar.

Für die im späteren betrachtete IFC-Analyse ist der Genauigkeitsgewinn bei der Verwendung einer *0-1-C-CFA* vernachlässigbar, da er hauptsächlich an Stellen auftritt, die aufgrund ihres Kontrollflusses bereits als potentielle Sicherheitsverletzung markiert wurden.

Weitere Methoden

Im Zuge dieser Arbeit kamen lediglich Konfigurationen der *CFA* zum Einsatz: Die Erstellung eines im Verlauf der Arbeit beschriebenen spezialisierten Kontextes wurde lediglich für diesen Typ umgesetzt. Weitere Algorithmen seien hier nur der Vollständigkeit halber mit kurzer Erläuterung aufgelistet.

Rapid Type Analysis (RTA)
Auch die *RTA* startet bei den Einsprungspunkten eines Programms. Zunächst sind lediglich Pfade zu Konstruktoren als "erlaubt" markiert. Wird nun eine *AllocationSite* besucht, so werden alle Funktionen in der *VTable* des allokierten Typs ebenfalls als "erlaubt" markiert.

Bei Besuch einer *CallSite* werden – abhängig von der Signatur – Kanten zu allen "erlaubten" Funktionen von Klassen, welche von dem deklarierten Typ der Instanz, auf der die Methode aufgerufen wurde, ableitbar sind, gezogen.

Bei der *RTA* handelt es sich somit um eine *typbasierte* Analyse.

Agesen's Cartesian Product Algorithm (CPA)
Der *CPA-Algorithmus* [2] erzeugt für eine Funktion das kartesische Produkt der möglichen Typen ihres Rückgabewerts und aller möglichen Typen ihrer Parameter[5]. Für alle Elemente dieses Produktes wird anschließend ein *Template* in einem Verzeichnis abgelegt. Bei Besuch einer *CallSite* kann somit das Ziel anhand dieses Verzeichnisses aufgelöst werden.

[5]Für den Aufruf einer Funktion mit n Parametern enthält es also $(n + 1)$-Tupel

Simple Class Sets (SCS)
Der *SCS-Algorithmus* [20] unterscheidet sich von dem *CPA-Algorithmus* darin, dass Mengen von Typen anstatt jeweils nur eines Typs Verwendung finden.

2.2.3 Typinferenz

Durch die *Typinferenz* [25] kann zu einem Programmpunkt p und einer Variable x festgestellt werden, von welchem Typ die Variable sein kann. Typen werden jeweils an Zuweisungsstellen übernommen.

Typinferenz kann in untypisierten Sprachen zur Feststellung des Datentyps anhand von Zuweisungen oder beispielsweise zur Verifikation in typisierten Sprachen genutzt werden.

Es existieren verschiedene Verfahren zur Berechnung der *Typinferenz*.

2.2.4 Static Single Assignment (SSA)

Bei Verwendung von *SSA* [7] darf einer Variable lediglich einmalig ein Wert zugewiesen werden. Soll einer Variable des Quellprogramms ein neuer Wert zugewiesen werden, so wird in der Zwischendarstellung des Programms eine neue Variable angelegt.

```
int i = 5;
int j = 3;
if ( i > j ) {
        i = j;
        j = 3;
} else {
        j = 2;
}


print ( j );
j = 1;
print ( j );
```

```
int i_1 = 5;
int j_1 = 3;
if ( i_1 > j_1 ) {
        i_2 = j_1;
        j_2 = 3;
} else {
        j_3 = 2;
}
j_4 = Phi(j_2, j_3)
i_3 = Phi(i_2, i_1)
print ( j_4 );
j_5 = 1;
print ( j_5 );
```

Listing 3: *Ursprünglicher Beispielcode und Beispielcode in SSA-Form*

Ein besonderes Vorgehen muss angewandt werden, wenn eine Variable von unterschiedlichen *Kontrollflusspfaden* (siehe Unterabschnitt 2.1.3) beeinflusst wird. Für diesen Fall ist die Φ-Funktion definiert, die einen Parameter abhängig vom Eingangspfad "durchschaltet". Φ-Funktionen befinden sich demnach zwischen *Grundblöcken* (siehe Unterabschnitt 2.1.2).

Die Verwendung von *SSA* bietet den Vorteil, dass in der späteren Analyse- bzw. Übersetzungsphase keine Rücksicht auf Redefinitionen oder Überschattungen mehr genommen werden muss. Dies stellt somit eine starke Vereinfachung dar.

2.3 Struktur einer Android-Applikation

Applikationen für Android-Systeme weisen eine besondere Struktur auf. Einzelne Fenster und auch Dienste der Anwendung sind derart hinterlegt, dass sie unabhängig voneinander gestartet, pausiert und beendet werden können. Es ist beispielsweise also möglich das Konfigurationsfenster einer Anwendung zu öffnen, ohne dabei die ”Hauptanwendung” ausführen zu müssen.

Dieses Verhalten wird dadurch ermöglicht, dass Android-Applikationen eine hohe Zahl von *Einsprungspunkten* (siehe Unterabschnitt 2.3.3) aufweisen. Eine Android-Komponente überschreibt (*override*) zur Definition ihrer Einsprungspunkte die entsprechenden Funktionen ihrer Oberklasse. Den Komponenten selbst können über sogenannte *Intents* (siehe Unterabschnitt 2.3.5) angesprochen werden.

Jede Android-Applikation wird so gut wie immer in ihrer eigenen *Sandbox* ausgeführt. Dies äußert sich darin, dass jede Anwendung mit eigener Linux User-ID versehen ist, in eigenem Prozess und eigener VM-Instanz läuft. Diese Regel kann in bestimmten Fällen gebrochen werden (siehe Unterabschnitt 2.3.1). Durch diese Separierung der Anwendungen kann die Kommunikation zwischen Anwendungen – Android Anwendungen sind stark auf eine Kommunikation untereinander ausgelegt – lediglich auf Pfaden verlaufen, die das Betriebssystem miteinbeziehen. In Unterabschnitt 2.3.6 wird auf Möglichkeiten des Datenaustausches eingegangen.

Die Informationen darüber, was die Anwendung zur Verfügung stellt, ist in ihrem *Manifest* (siehe Unterabschnitt 2.3.8) beschrieben, einer zwingend erforderlichen Datei, die im Android-Containerformat für Applikationen (siehe Unterabschnitt 2.3.8) hinterlegt ist.

Weiterhin enthält das Containerformat natürlich die Applikation selbst. Diese liegt im Fall von Android-Applikationen meist in dem Bytecodeformat *Dalvik* (siehe Unterabschnitt 2.3.7) vor. Es existieren allerdings teilweise auch nativ Kompilierte Programme und Bibliotheken. Diese können mit den in dieser Arbeit dargestellten Hilfsmitteln allerdings nicht analysiert werden, da *WALA* derzeit noch keine Möglichkeit der Interpretation dieses Formates hat.

2.3.1 Ausführungsverhalten von Android-Applikationen

Für das Verständnis der späteren Betrachtungen von Android-Applikationen ist es hilfreich zunächst auf das ungewohnte Ausführungsverhalten Androids einzugehen.

Standardmäßig existiert für jede Applikation maximal ein Prozess unabhängig davon, wie häufig sie gestartet wird. In diesem Prozess existiert zunächst ein einziger *Thread*, der *Main-Thread*.

Soll nun – egal durch welchen Aufrufer – eine weitere Komponente (oder eine neue Instanz der selben Komponente) der Applikation gestartet werden, so wird diese zunächst ebenfalls in diesem einen *Main-Thread* ausgeführt. Alle Einsprungspunktaufrufe und *CallBack-Funktionen* des Systems laufen über diesen *Thread*, außerdem kontrolliert er das Zeichnen von Fenstern. Es ist also wichtig die Auslastung des *Main-Thread* durch das Starten weiterer *Threads*, welche den Hauptteil der Rechenarbeit übernehmen, möglichst gering zu halten. Da das *UI-Toolkit* Androids nicht *Thread-Safe* ist, kann eine starke Kommunikation zwischen den einzelnen *Threads* nötig sein.

In begründeten Fällen kann man Android mittels des *Manifest* anweisen gewisse Komponenten in einem anderen Prozess als dem der restlichen Applikation auszuführen. Über diese Einstellung lassen sich gewisse Komponenten in einem Prozess gruppieren, man kann einen Prozess je Komponente starten. Man kann Android sogar anweisen eine Komponente in einem Prozess einer anderen Applikation zu starten. Dies geht allerdings nur, wenn beide Anwendungen mit der selben Linux User-ID laufen und mit dem selben Zertifikat signiert sind.

2.3.2 Die Bestandteile des Namensraums android.app

Das Paket *android.app* kapselt die Kernkomponenten einer Android-Applikation. Diese sind *Activity*, *Service*, *BroadcastReceiver* und *ContentProvider*.

Weiterhin finden sich in diesem Paket einige Helfer für die Benutzerinteraktion (z.B. *AlertDialog*, *Notification* oder die *ActionBar*). Ausnahmen bilden derzeit (Android API-Level 18) der *Download-Manager* und *Loader*, welche im Hintergrund agieren, sowie die Klasse *Instrumentation*: Wenn die Ausführung mit *Instrumentation* aktiviert ist, so lässt sich durch diese Klasse sämtliche Interaktion des Systems mit einer Applikation mitlesen.

2.3.3 Einsprungspunkte

Bei einem Einsprungspunkt handelt es sich zunächst um eine Adresse eines Programms, an der die Programmausführung begonnen oder fortgesetzt werden kann. Hochsprachen setzen zur Festlegung solcher Punkte oft Funktionsnamen fester Signatur (z.B. *int main()*) ein.

Android-Applikationen setzen auf eine hohe Zahl von Einsprungspunkten, um die Ausführung flexibler zu unterbrechen oder nur Teilbereiche eines Programms auszuführen. Weiterhin werden bei Android Rückruffunktionen zur asynchronen Signalisierung benutzt (z.B. *onPause*, *onActivityResult*). Einsprungspunkte können gemäß eines Lebenszyklus (siehe Unterabschnitt 2.3.4) aufgerufen werden.

Die Klassennamen der Applikations-Komponenten, welche die Einsprungspunkte der Applikation zur Verfügung stellen, sind fest in ihrem *Manifest* (siehe Unterabschnitt 2.3.8) beschrieben. Die eigentlichen Einsprungspunkte werden dann durch

festgeschriebene Funktionen dieser Klassen gebildet: Die Klassen sind typischerweise von *Activity*, *Service*, *ContentProvider* oder *BroadcastReceiver* abgeleitet und überladen gegebenenfalls deren Funktionen.

Weiterhin können später *CallBack-Funktionen*[6], die durch das System aufgerufen werden, zu den modellierten hinzugefügt werden. Ein Beispiel hierfür sind Funktionen von Klassen, die *LocationListener* implementieren: Eine Anwendung kann eine solche Klasse im *LocationManager* registrieren. Android ruft die *CallBack-Funktion* dann außerhalb von für Benutzeranwendungen geltenden Konventionen auf.

Berücksichtigt werden weiterhin Einsprungspunkte für die gesamte *Applikation*, den *BackupAgent* und einige mehr: Eine Auflistung der erkannten Signaturen findet sich in Anhang A, die Methode der Erkennung wird in Abschnitt 4.1 beschrieben.

Zur Feststellung der Signaturen, die erkannt werden sollen, wurde die Android Entwickler Referenz [6] durchgearbeitet.

2.3.4 Lebenszyklus der Einsprungsklassen

Die Einsprungspunkte einer Klasse hängen von ihrer Super-Klasse ab. Im Folgenden sei die Reihenfolge, in der Einsprungspunkte aufgerufen werden können für Klassen dargestellt, an denen der *Prozess* gestartet werden kann. Die hier vorzufindenden Einsprungspunkte werden teilweise bei geändertem *Scheduling* aufgerufen.

Lebenszyklus einer app.Application

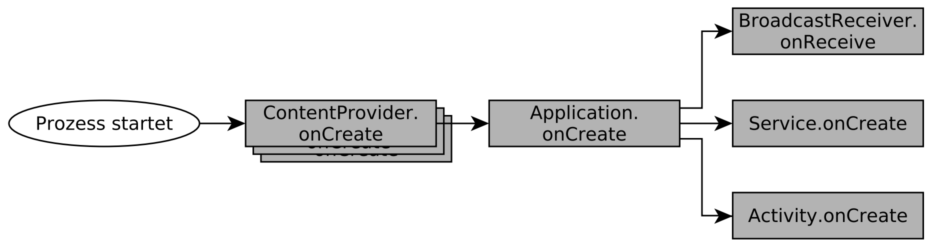


Abbildung 2.8: *Lebenszyklus einer Applikation*

Der selten implementierte *onCreate*-Einsprungspunkt der *app.Application* ist in Abbildung 2.8 dargestellt. Er wird vor dem *onCreate*-Einsprungspunkt der *Activity* oder des *Service* beziehungsweise vor *onReceive* des *BroadcastReceivers*, welcher

[6]Eine *CallBack-Funktion* ist eine Funktion des eigentlichen Programms, deren Adresse einer weiteren Funktion als Parameter übergeben wird. Sie wird durch diese für Rückfragen oder zur Signalisierung aufgerufen.

Ursache des Starts der Applikation ist, ausgeführt. Der Start weiterer *Activities*, *Services* oder weitere *onReceives* verursachen keine erneute Ausführung dieses Einsprungspunktes.

Noch vor Ausführung von *Application.onCreate* werden (sofern vorhanden) alle *ContentProvider* der Applikation gestartet (siehe Abbildung 2.3.4).

Application.onCreate kann dazu herangezogen werden einen globalen Status der Applikation herzustellen. In der Praxis werden für diesen Zweck jedoch meist statische *Singleton-Klassen*[7] bevorzugt.

Lebenszyklus einer Activity

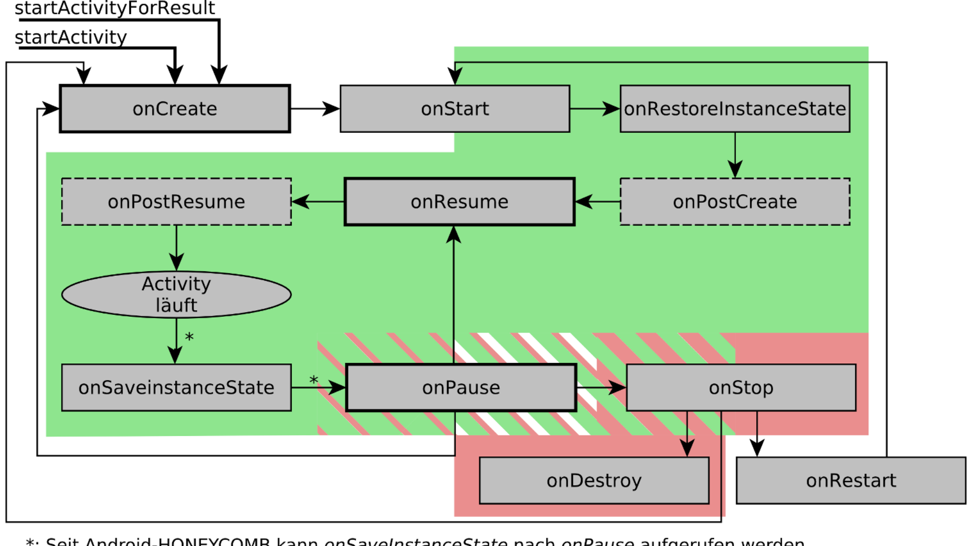


Abbildung 2.9: *Lebenszyklus einer Activity*

Abbildung 2.9 zeigt den Lebenszyklus einer *Activity*. Activities werden für Programmteile mit graphischer Ausgabe verwendet. Ihre Ausführung wird unter anderem immer dann unterbrochen, wenn eine andere *Activity* in den Vordergrund kommt. In diesem Fall fährt der Lebenszyklus mit *onSaveInstanceState* fort. *onDestroy* wird lediglich dann zwingend aufgerufen, wenn die *Activity* regulär beendet wird. Wird

[7]Von einer *Singleton-Klasse* kann zu einem beliebigen Zeitpunkt maximal eine Instanz existieren. Ein Versuch der Reinstanziierung liefert die bereits vorhandene Instanz zurück

die *Activity* jedoch aufgrund von Speicherknappheit beendet[8], so kann sie bereits zuvor abgebrochen werden. Kommt eine *Activity* aus dem Hintergrund wieder nach vorne, so wird deren Ausführung mit *onResume* oder *onRestart* fortgesetzt.

Bis auf den Einsprungspunkt *onCreate* ist die Implementierung der Einsprungspunkte optional. Dennoch wird so gut wie jede *Activity* auch *onResume* und *onPause*[9] beziehungsweise *onStop* – zum Speichern des Status[10] – implementieren. Möchte eine Applikation ihren gespeicherten Status wiederherstellen, so geschieht das in der Regel in *onStart*.

Für eine flüssige Bedienung der Applikation sollte die Ausführung nicht zu lange in den einzelnen Funktionen verweilen. Zeitaufwändige Berechnungen sollte man deshalb in einen *Service* auslagern. Ist dies nicht möglich oder gewünscht, so ließe sich die Laufzeit von *onResume* verlängern beziehungsweise *onPostResume*[11] implementieren. *onStop* bietet sich eventuell für Berechnungen mit längerer Laufzeit an. Es ist allerdings zu berücksichtigen, dass ein Prozess, der sich in *onStop* befindet jederzeit abgebrochen werden kann.

Activities einer Android Applikation gelten als gleichberechtigt. Applikationen verfügen zwar meist über eine *Main-Activity*, diese unterscheidet sich von den anderen *Activities* jedoch nur durch einen entsprechenden *Intent* (siehe Unterabschnitt 2.3.5). Wird eine andere *Activity* von einer Anwendung angefordert, so bedeutet dies nicht, dass auch die *Main-Activity* ausgeführt wird.

Da man sie in der Praxis häufig antrifft sei hier kurz die *FragmentActivity* erwähnt. Diese kann *Fragments*, eine Art untergeordnete *Activity*, enthalten. Einsprungspunkte der *FragmentActivity* sind zunächst analog zu Einsprungspunkten einer *Activity*. Allerdings werden gegebenenfalls zusätzlich die analogen Einsprungspunkte der in der *FragmentActivity* registrierten *Fragments* aufgerufen.

Lebenszyklus eines Service

Services werden für Hintergrundberechnungen herangezogen. Durch diese wird sichergestellt, dass den Berechnungen auch dann Rechenzeit zugestanden wird, wenn sich die Anwendung im Hintergrund befindet. Weiterhin wird bei der Verwendung eines *Service* eine andere Policy bezüglich des Abbruchs der Anwendung bei Speicherknappheit herangezogen.

[8] Eine *Activity* muss bei Speicherknappheit nicht zwangsläufig beendet werden. Zuvor kann sie über *onLowMemory* dazu aufgefordert werden selbständig Speicher freizugeben

[9] Die Ausführungszeit von *onPause* verzögert das Sichtbarwerden der nächsten *Activity*. Zeitaufwändige Speichervorgänge sollten von daher in *onStop* erfolgen

[10] Zwar existiert ein Einsprungspunkt *onSaveInstanceState*, dieser wird allerdings nicht zwangsweise im Lebenszyklus aufgerufen. Er dient zum temporären Hinterlegen von Daten in einem *Bundle*. Für persistentes Speichern sollte man deshalb *onPause* oder *onStop* benutzen. *onPause* ist hierbei vorzuziehen, wenn der Benutzer nach Verlassen der Ansicht erwartet, dass die Daten gespeichert sind.

[11] *onPostResume* wird systemintern dazu verwendet den Resume-Vorgang abzuschließen. Man kann diese Methode allerdings auch in seiner Applikation verwenden.

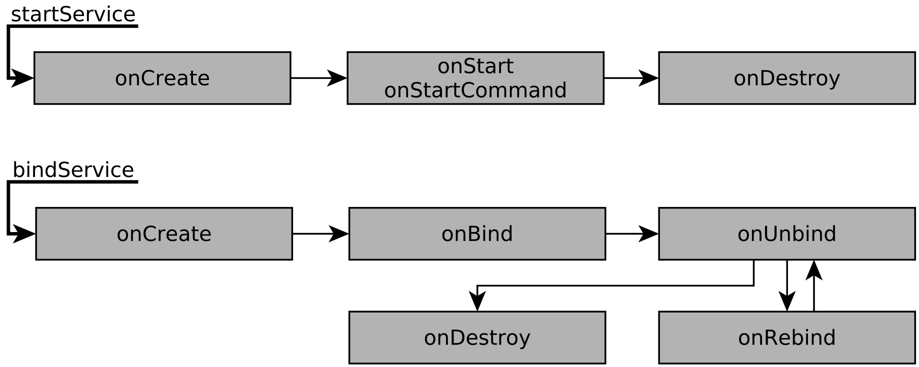


Abbildung 2.10: *Lebenszyklus eines Service*

Services weisen einen anderen Lebenszyklus auf, wie aus Abbildung 2.10 hervorgeht. Bei ihnen muss zunächst unterschieden werden, ob es sich um einen Service handelt, der über *startService* oder *bindService* gestartet wird. Ein *Service*, der über *startService* gestartet wurde, bekommt keine Rückmeldung, wenn er gestoppt wird.

Da die Einsprungspunkte der Services im *Thread* des Aufrufers ausgeführt werden, sollte die Ausführung eine kurze Verweildauer in diesen Funktionen haben um den Aufrufer nicht unnötig zu blockieren. In der Regel nutzt man die Einsprungspunkte dafür einen neuen *Thread* zu starten, der dann die eigentliche Berechnung enthält. Der Umweg über einen *Service* ist dem direkten Starten eines Threads durch eine *Activity* vorzuziehen um sicherzustellen, dass dem *Thread* genug Rechenzeit zugestanden wird[12] und er nicht vorzeitig beendet wird. Alternativ kann ein *AsyncTask*, welcher dafür ausgelegt ist den Bildschirminhalt während der Hintergrundberechnung zu aktualisieren, verwendet werden.

Die Ausführung durch einen *Service* gestarteter Threads sollte nicht durch Android zwangsweise beendet werden: Dies geschieht lediglich bei extremer Speicherknappheit. Dann werden zu beendende Prozesse durch ein Prioritätssystem ausgewählt. Diese Betrachtung liegt jedoch außerhalb des Umfangs dieser Arbeit.

Lebenszyklus eines Broadcast Receiver

Der Lebenszyklus eines *BroadcastReceivers* besteht lediglich aus einer Funktion (*onReceive*). Nach verlassen dieser Funktion gilt die Ausführung als beendet. Somit sind in dieser Funktion lediglich synchrone Operationen möglich. Insbesondere sollte man in *onReceive* keinen *Thread* starten[13].

[12] Befindet sich eine *Activity* im Pause-Zustand wird ihr keine Rechenzeit zugewiesen.
[13] Zwar kann man einen *Thread* starten, dieser wird sich aber nicht wie erwartet verhalten.

Aus diesem Grund wird ein *BroadcastReceiver* meist in Verbindung mit einem *Service* verwendet [6]. Dadurch wird verhindert, dass der Prozess zu früh beendet wird.

Lebenszyklus eines ContentProviders

ContentProvider finden Verwendung, wenn eine Applikation anderen Applikationen Daten zur Verfügung stellen will (für den Datenaustausch innerhalb einer Applikation werden sie nicht benötigt – das Entwicklerhandbuch [9] schlägt hier *Singleton-Klassen* vor).

Der Einsprungspunkt *onCreate* des *ContentProviders* wird zum Zeitpunkt des Starts der Applikation (noch vor *Application.onCreate*) aufgerufen. Neben des Startens einer *Activity*, eines *Service* oder *BroadcastReceivers* kann eine Applikation auch durch Zugriff auf einen *ContentProvider* gestartet werden.

Der *ContentProvider* residiert während der gesamten Laufzeit des Prozesses der Applikation im Speicher. Da Android einen Prozess nur bei Speichermangel beendet existiert er also potentiell bis das Gerät ausgeschaltet wird. Ein *ContentProvider* wird nicht vor seinem Beenden informiert.

2.3.5 Intents

Zur Kommunikation mit dritten Anwendungen werden bei Android *Intents* verwendet, welche durch das *Manifest* (siehe Unterabschnitt 2.3.8) einem Einsprungspunkt der Zielanwendung zugeordnet sind. *Intents* werden durch einen String mit hierarchischer Nomenklatur repräsentiert.

```
Intent i = new Intent("com.example.demo.MainActivity");
startActivity(i)
```

Listing 4: *Starten einer Applikation mittels ihres Intents*

Listing 4 zeigt wie das Programm, welches in seinem *Manifest* (siehe Unterabschnitt 2.3.8) mit einem *Intent* wie in Listing 8 (auf Seite 30) versehen ist, gestartet werden kann. Der dargestellte Code zieht nach sich, dass Einsprungspunkte in folgender Reihenfolge angesprungen werden[14]: *com.example.demo.onCreate*, *MainActivity.onCreate*, *MainActivity.onStart*, *MainActivity.onRestoreInstanceState*, *MainActivity.onPostCreate*, *MainActivity.onResume*, *MainActivity.onPostResume*.

Zu einem *Intent* können mehrere Implementierungen (durch unterschiedliche Applikationen) registriert sein. In diesem Fall kann ein explizites *Intent* durch Angabe

[14] Wenn noch kein Prozess der Anwendung gestartet wurde und die Anwendung keine *ContentProvider* enthält.

einer Applikationsinstanz gewählt werden, alternativ wird der Benutzer zur Auswahl der Applikation aufgefordert. Diese Intents nennen sich *implizite Intents*.

```
Intent i = new Intent("android.intent.action.VIEW", Uri.parse("http://kit.edu/"));
startActivity (i);
```

Listing 5: *Starten eines impliziten Intents*

Anhand der angegebenen *URI*[15] wird in dem Beispiel aus Listing 5 zunächst festgestellt, dass ein Web-Browser gestartet werden soll. Hat der Benutzer auf seinem Gerät mehrere Web-Browser installiert, so wird er nun gefragt, welchen er verwenden will.

Weiterhin existieren spezielle *Broadcast-Intents*, welche an alle Anwendungen zugestellt werden[16].

Intents haben also einen Namen und Typ. Über einen *Intent-Filter* sind sie außerdem mit einer Kategorie, einer Aktion und optional einem Datenfilter, der auf *URIs* angewandt wird, versehen. Er kann weiterhin zusätzliche Metadaten enthalten.

Sollen weitere Daten mit der Zielanwendung ausgetauscht werden, so können diese mit *Intent.setData* vor dem Aufruf zu dem *Intent* hinzugefügt werden. Die Zielanwendung kann diese dann mittels *getData* abrufen. Möchte man Daten aus einer Ziel-*Activity* abrufen, so startet man diese mit *startActivityForResult()*. Die Übergabe der Daten wird dann durch den Aufruf von *onActivityResult()* vorgenommen.

Informationen darüber welche Intents durch eine Applikation zur Verfügung gestellt werden und durch welche Klassen diese realisieren finden sich im *Manifest* (siehe Unterabschnitt 2.3.8) der Anwendung. *Broadcast-Intents* können zudem zur Laufzeit registriert werden.

2.3.6 Kommunikation in Androidanwendungen

Da jede Android-Applikation in ihrer eigenen *Sandbox* ausgeführt wird und auch *Activities* und *Services* voneinander unabhängig gestartet und beendet werden können muss die Kommunikation zwischen den einzelnen Komponenten über Nachrichtenaustausche[17] verlaufen. Android bietet hier zahlreiche Möglichkeiten an:

Remote Procedure Calls (RPC)
Android bietet die Möglichkeit über *Remote Procedure Calls (RPC)* Daten mit anderen Prozessen auszutauschen (*Inter Process Communication – IPC)*. Hierbei

[15] Bei einem *Uniform Resource Identifier* handelt es sich um eine Zeichenfolge, die aus Zugriffsschema, Authority und optional Pfad, Anfrage und Fragment zusammengesetzt ist

[16] Man kann einen "normalen" *Intent* allerdings nicht zu einem *Broadcast-Intent* umfunktionieren, also nicht einfach "mithören"

[17] Das heißt nicht direkt über *Shared Memory*

wird eine Funktion, die durch eine Komponente aufgerufen wird, in einem fremden Prozess ausgeführt.

Um *IPC* mittels der durch Android zur Verfügung gestellten Mechanismen ausführen zu können muss sich eine Applikation mittels *bindService()* mit einem *Service* binden.

Vorwärtsübermittlung durch ein Intent

Es gibt mehrere Möglichkeiten ein *Intent* mit Daten zur Übermittlung zu versehen. Die offensichtlichste besteht in der Kodierung der Daten in einer *URI*.

Eine weitere Möglichkeit besteht darin Daten mittels *putExtra* zu einem *Intent* hinzuzufügen. In einem Einsprungspunkt der Zielanwendung kann das aufrufende *Intent* mittels *getIntent()* gefunden werden. Mittels *getData* können dann die Daten wieder extrahiert werden.

Rückgabewerte einer Activity

Eine *Activity*, die durch *startActivityForResult* gestartet wird, kann in ihrem *onPause()* Entry-Point Daten mittels *setResult()* Daten zur Rückübermittlung festlegen. In der aufrufenden *Activity* wird dann *onActivityResult* mit den entsprechenden Daten aufgerufen. Die Daten selbst sind wiederum in einem *Intent* kodiert.

ContentProvider

ContentProvider werden dazu genutzt über Applikationsgrenzen hinweg Daten verfügbar zu machen. Der Zugriff weißt dabei eine Struktur ähnlich der relationaler Datenbanken auf. In Verbindung mit einem *ContentProviderClient* oder einem *ContentResolver* wird der Aufwand der *IPC* abstrahiert.

Dateisystem

Jeder Android-Anwendung ist eine eigene Linux-User-ID zugeordnet. Über *openFileInput()* kann auf den internen Speicher, mittels *getExternalFilesDir()* auf ein applikationsspezifisches Verzeichnis auf externem Speicher zugegriffen werden.

Sollen Daten mit anderen Anwendungen geteilt werden, so können sie in einem Verzeichnis, welches man mittels *getExternalStoragePublicDirectory()* erhält gespeichert werden. Über *getExternalStorageDirectory()* erhält man analog dazu das Wurzelverzeichnis eines externen Speichers.

Über ein *Flag* können durch eine Applikation abgelegte Dateien öffentlich lesbar gemacht werden (auch die im privaten Speicher).

Android unterstützt auch *Named Pipes*, sofern diese auf einem Dateisystem angelegt werden, das diese unterstützt[18].

Ein *Loader* dient dazu einen Ladevorgang asynchron zu machen. Er kann lediglich in der Applikation, in der er definiert ist verwendet werden. Während ihrer

[18] Das auf externen Speichermedien weit verbreitete Dateisystem FAT bietet keine Unterstützung für Named Pipes

Ausführung verfolgen *Loader* Änderungen an Datenquellen, aus denen sie ihre Informationen beziehen und benachrichtigen gegebenenfalls die Programmteile, die den *Loader* benutzen.

Netzwerk-Stack
Android unterstützt auch *UNIX-Domain Sockets*, unter *Java* kann man beispielsweise mittels eines *android.net.LocalServerSocket* darauf zugreifen. Über *java.net.Server-Socket* lassen sich auch reguläre Netzwerk-Sockets erstellen.

2.3.7 Dalvik Bytecode

Bei *Dalvik* handelt es sich um ein *Bytecode-Format*, welches häufig für Android-Applikationen herangezogen wird. Dieses Format ist zunächst mit dem *Bytecode-Format Javas* vergleichbar, unterscheidet sich von diesem allerdings dahingehend maßgeblich, dass das Format nicht für eine *Stack*-, sondern für eine *Registermaschine* konzipiert ist.

Mnemonik Code	Bytecode	Original	Stack nach Ausführung[a]
iconst_5	**08**	int i = 5;	5(=i)
iconst_3	**06**	int j = 3;	3(=j), i
dup_x1	**5a**		i, j, i
dup_x1	**5a**		j, i, j, i
if_icmple @else	**a4** __ __[b]	if (i > j) {	j, i
dup	**59**	i = j;	j(=I), j, i
iconst_3	**06**	j = 3;	3(=J), I, j, i
goto @endif	**a7** __ __	} else {	
else:			
dup_x1	**5a**		i, j, i
iconst_2	**05**	j = 2;	2(=J), i, j, i
endif:		}	
jsr @print	**a8** __ __	print (...);	PC[c], J, I/i, j, i
const_1	**04**	j = 1;	1, J, I/i, j, i
jsr @print	**a8** __ __	print (...);	PC, 1, J, I/i, j, i
pop2	**58**		I/i, j, i
pop2	**58**		i
pop	**57**		

Listing 6: *Das Beispiel als Java-Bytecode*

[a] Auf dem *Stack* stehen die Werte der Variablen und nicht ihre Symbole.

[b] Ein Tiefstrich repräsentiert ein Byte, dessen Inhalt vom Linker gesetzt werden muss. Existieren mehrere zueinander analoge *Mnemoniks* unterschiedlicher Adressbreite, so wurde jeweils die Short-Variante für nahe Sprungziele gewählt.

[c] *PC* steht für die Rücksprungadresse aus der Funktion heraus. Direkt nach der *jsr*-Instruktion steht diese auf dem *Stack*. Sie wird allerdings durch die aufgerufene Funktion wieder entfernt.

Mnemonik Code	Bytecode	Original
const *v0*, 5	**14***00* 0500 0000	int i = 5;
const *v1*, 3	**14***01* 0300 0000	int j = 3;
if-le *v0*, v1, @else	**37**1*0* ____	if (i > j) {
move *v0*, v1	**01**1*0*	i = j;
const *v1*, 3	**14***01* 0300 0000	j = 3;
goto @endif	**28**__	} else {
else:		
const *v1*, 2	**14***01* 0200 0000	j = 2;
endif:		}
invoke-static @print{v1}	**71**10 ____ 0100	print (j);
const *v1*, 1	**14***01* 0100 0000	j = 1;
invoke-static @print{v1}	**71**10 ____ 0100	print (j);

Listing 7: *Das Beispiel als Dalvik-Bytecode*

Listing 6 und Listing 7 zeigen jeweils *Bytecode*, wie er für das Beispiel aus den vorherigen Kapiteln generiert werden könnte[19]. Offensichtlich sind in den aufgeführten *Bytecodes* die Sprungziele noch nicht aufgelöst: Das aufzulösende Label hat in der Darstellung das Präfix "@", im *Bytecode* markiert der Tiefstrich den Platz zur Aufnahme der Adresse.

Man erkennt deutlich den Unterschied zwischen *Stack*- und *Registermaschine*. Unter anderem weist der Code für die *Stackmaschine* mehr (dafür in der *Bytedarstellung* kürzere) Instruktionen auf. Der Code der *Registermaschine* befindet sich potentiell näher an der Codierung, welche durch verbreitete Prozessoren ausgeführt wird[20].

Die *Register* bei *Dalvik* weisen eine Breite von 32Bit auf [8], nebeneinander liegende *Register* können zu einem 64Bit-*Register* zusammengefasst werden. Ist ein Großteil der Instruktionen auf die Verwendung der ersten 16 *Register* beschränkt, existieren allerdings auch solche, welche 256 *Register* verwenden können. Spezielle Kopierinstruktionen können sogar 65536 *Register* adressieren [8].

Die Nomenklatur von Typen und Klassen in *Dalvik* ist absolut identisch zu *Java*, sodass es Konvertierungsprogramme von *Java-Bytecode* zu *Dalvik* gibt: Das Programm *dx* des Android-Projekts liest *.class*-Dateien ein und gibt *Dalvik* aus.

Dalvik-Kompilate finden sich als Dateien mit der Endung *.dex* (Dalvik Executable-Format). Eine Android-Anwendung wird jedoch meist im Containerformat *.apk* (siehe Unterabschnitt 2.3.8) ausgeliefert.

Dalvik Bytecode wird von einer *virtuellen Maschine* (VM) mit *Registern* interpretiert. Eine solche *VM*, welche von Google entwickelt wurde, bildet den zentralen

[19]In der Praxis sieht der generierte Code wahrscheinlich anders aus, da dann Optimierungen vorgenommen werden oder Befehle umsortiert werden. Den Java *Bytecode* könnte man außerdem unter Verwendung "Lokaler Variablen", einem Ablageort für Daten abseits des *Stack*, schreiben.

[20]Es existieren zwar auch Prozessoren, die auf dem *Stack* operieren, diese sind allerdings nicht sehr weit verbreitet.

Teil von Android. Für jeden *Dalvik-Prozess* startet Android eine eigene *VM-Instanz* (bis auf Ausnahmen gemäß Unterabschnitt 2.3.1).

2.3.8 APK-Containerformat

Das aus *Java* bekannte Konzept von Softwarepaketen (dort das *.jar*-Format) findet sich auch bei *Dalvik-Software* wieder. Dort sind diese Pakete als *.apk*-Dateien zu finden, welche strukturell wieder sehr nahe am *.jar*-Format gehalten sind: Neben den eigentlichen Kompilaten und optional dem Quelltext können auch beliebige weitere Dateien enthalten sein. Sämtliche Dateien werden auch hier in einem ZIP-Archiv zusammengefasst, welches über spezielle standardisierte Dateien weitere Informationen über den Inhalt liefert.

Im Gegensatz zum *.jar*-Format liegen XML-Beschreibungen des Inhalts bei *.apk* in einer komprimierten Form vor. Diese Dateien können mit Hilfe des Programms *apktool* [35] aus dem *.apk*-Container extrahiert und entpackt werden.

Manifest

Eine Android Applikation wird durch eine spezielle Datei "*AndroidManifest.xml*" beschrieben. Listing 8 zeigt ein Beispiel einer solchen Datei: Die Datei spezifiziert eine Applikation mit einer einzigen Klasse für Einsprungspunkte (siehe Unterabschnitt 2.3.3), "com.example.demo.MainActivity". Das Starten dieser *Activity* wurde zuvor in Unterabschnitt 2.3.5 beschrieben.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest android:versionCode="1" android:versionName="1.0" package="com.ex.demo"
      xmlns:android="http://schemas.android.com/apk/res/android">
 <uses-permission android:name="android.permission.CAMERA" />
 <uses-feature android:name="android.hardware.camera" android:required="false" />
 <supports-screens android:anyDensity="true" android:smallScreens="true" />
 <application  android:label ="@string/app_name" android:icon="@drawable/launch">
      < activity  android:label ="@string/app_name"
              android:name="com.ex.demo.MainActivity">
         <intent- filter >
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
         </intent- filter >
      </ activity >
 </application>
</manifest>
```

Listing 8: *Beispiel einer Manifest-Datei*

Außerdem erkennt man in der Datei benötigte Berechtigungen der Software (hier: Kamerazugriff) und kann Informationen ablesen, auf welchen Geräten das Programm lauffähig ist[21].

Weitere typische Einstellungen in dieser Datei umfassen beispielsweise das Halten eines statischen Zustandes bei Unterbrechung, die Bildschirmorientierung oder auch, ob die Software-Tastatur auf dem Bildschirm ausgeklappt werden soll.

Weitere typische Bestandteile

Weitere typische Bestandteile des *.apk*-Containers finden sich neben Icons auch in Grundeinstellungen, Menüs, Beschreibungen der Anwendung und Softwaresignatur.

2.4 T.J. Watson Libraries for Analysis (WALA)

Bei *WALA* [37] handelt es sich um eine in *Java* geschriebene Bibliothek zur Analyse von Drittsoftware in *Bytecode-* oder *Scriptformaten*. Derzeit unterstützt sind *Java Source-* und *Bytecode*, sowie *Java-Script*; eine angepasste Variante *WALAs* verarbeitet weiterhin das *Bytecodeformat Dalvik*, welches unter anderem bei Android-Applikationen zum Einsatz kommt.

Die eingelesenen Daten stellt *WALA* in unterschiedlichen Abstraktionsebenen (siehe Unterabschnitt 2.4.1) zur Verfügung. Weiterhin müssen meist sogenannte *Stubs* (siehe Unterabschnitt 2.4.2) hinzugeladen werden, um Bibliotheksaufrufe auflösen zu können.

2.4.1 Programmrepräsentation mittels WALA

Das Einlesen eines Programms mittels *WALA* durchläuft mehrere Schichten und stellt jeweils unterschiedlich abstrahierte Darstellungsformen zur Verfügung. Diese seien im Folgenden kurz angerissen.

Spezifikation zu ladender Bestandteile

🖮 com.ibm.wala.ipa.callgraph.AnalysisScope

Den Ausgangspunkt des Einlesens eines Programms mittels *WALA* bildet das *AnalysisScope*. In dieser Struktur wird zunächst hinterlegt, welche *Stubs* und woher das Programm geladen werden soll. Zu einer zu ladenden Struktur wird eine Referenz zugeordnet über welchen Mechanismus diese geladen werden soll. Weiterhin sind im *AnalysisScope* Informationen darüber hinterlegt, welche Klassen in den späteren Analysen ausgespart werden sollen.

Nach der Definition des *AnalysisScope* wird eine *Klassenhierarchie* erstellt.

[21] Meist ist die größte Problematik bezüglich der Lauffähigkeit eines Programms auf einem Gerät die Bildschirmauflösung.

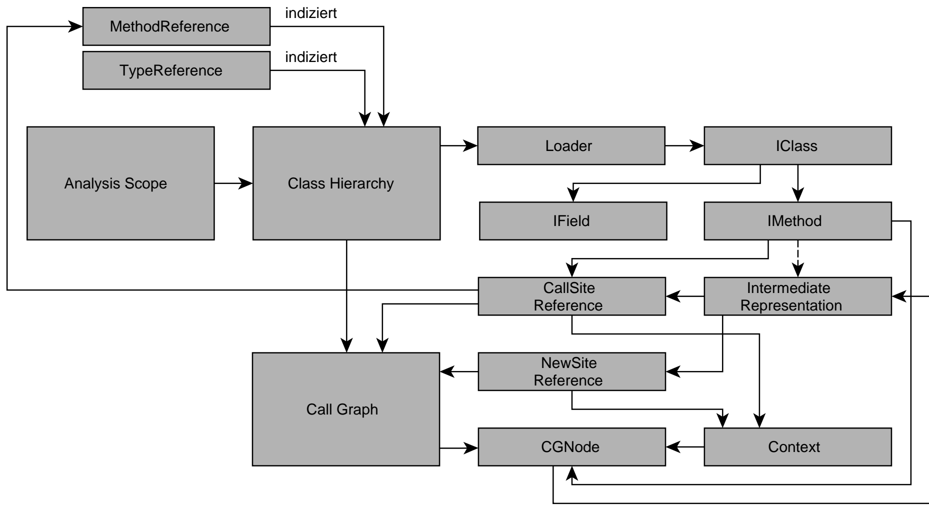


Abbildung 2.11: *Durchlauf der Erstellung einer Programmrepräsentation WALAs*

Erstellung der Klassenhierarchie

com.ibm.wala.ipa.cha.ClassHierarchy

WALAs ClassHierarchy dient nicht nur der Darstellung von Beziehungen zwischen Klassen, sie verwaltet auch die Klassen selbst. Für diese Verwaltung sind in der *ClassHierarchy* Informationen über verfügbare *Loader* hinterlegt. Die durch die Loader zur Verfügung gestellten Daten werden in der *ClassHierarchy* durch *MethodReferences* und *TypeReferences* indiziert. Diese Referenzen beinhalten neben den Namen der Klassen auch die Information über welchen *Loader* sie bezogen werden.

Die *Loader* selbst werden nach Priorität durchlaufen. Die niedrigste Priorität weist der *Primordial*-Loader auf. Dieser beinhaltet beispielsweise die *Stubs* der zu verwendenden Java-Umgebung und *primitive Datentypen*. Weitere standardmäßig vorhandene Loader finden sich im *Extension-Loader*, welcher benutzerdefinierte *Stubs* enthält, dem *Synthetic-Loader*, der zur Laufzeit generierte Klassen und schließlich dem *Application-Loader*. Weitere *Loader* lassen sich bei Bedarf hinzufügen.

Der zentrale Punkt der Klassenhierarchie liegt in der Katalogisierung von Vererbungsbeziehungen und Prüfung von Zuweisbarkeiten zwischen Variablen.

Für den in Abschnitt 4.1 beschriebenen Scan-Schritt der Modellerstellung ist die *Klassenhierarchie* bereits hinreichend.

IClasses und IMethods

🖮 com.ibm.wala.classLoader.IMethod

Die zuvor erstellte *Klassenhierarchie* kann nun nach sogenannten *IMethods* und *IClasses* durchsucht werden.

Über das Interface *IMethod* werden sprachspezifische Darstellungen von Methoden für die weitere Analyse gekapselt. Die spezifischen Darstellungen der Methoden enthalten Informationen, welche aus dem Einlesevorgang stammen (beispielsweise *Symboltabelle*, *Kontrollflussgraph* und Referenzen in einen *Abstrakten Syntax Baum*[22]).

Die durch *IMethod* verallgemeinerte Methodendarstellung gibt Aufschluss über *Signatur* und den Zugriffsrechte (z.B. *private* oder *public*), sowie in den Methoden auftretende lokale Variablen und Parameter. Über sie ist es weiterhin möglich die Position im *Bytecode* herauszufinden. Aus einer *IMethod* lassen sich zwar Aufrufstellen von Funktionen (*CallSite*) auslesen. Es kann für diese allerdings noch keine Aussage bezüglich des Sprungziels getroffen werden. Hierfür dient der im Folgenden generierte *CallGraph*.

Die *Intermediate Representation* (siehe Abbildung 2.4.1) einer Funktion ließe sich bereits aus einer *IMethod* generieren. In der Regel wird sie aber aus den später generierten *CGNodes* gewonnen, da diese unter Umständen noch Anpassungen vornehmen können.

WALAs Aufrufsgraph und CGNodes

🖮 com.ibm.wala.ipa.callgraph
🖮 com.ibm.wala.ipa.callgraph.CGNode

WALAs CallGraph (siehe Unterabschnitt 2.2.1) besteht aus sogenannten *CGNodes*. Ein *CGNode* enthält neben der zugehörigen *IMethod* auch einen *Context*.

Der inhaltliche Umfang des *Context* richtet sich nach dem Verwendeten *CallGraphBuilder*. Typische Bestandteile finden sich in Aufrufer, Aufrufsstelle und Referenzen auf die Instanzen übergebener Parameter; der *Context* lässt sich aber auch um benutzerdefinierte Einträge erweitern[23]. Die Generierung des *Context* findet in *WALA* durch einen *ContextSelector*, die Auswertung des *Context* durch einen *ContextInterpreter* statt. Letzterer bietet die letzte Möglichkeit der Anpassung der *Intermediate Representation* (siehe Abbildung 2.4.1), welche in aller Regel aus den jeweiligen *CGNodes* bezogen wird.

Wie schon in Unterabschnitt 2.2.1 beschrieben können auch in *WALA* zu einer *IMethod* mehrere *CGNodes* existieren.

[22] Im *Abstrakten Syntax Baum* werden *Tokens*, also Schlüsselwörter, Variablen oder auch Konstanten, in einer Baumstruktur hinterlegt. Er wird durch einen *Parser* aus dem Quelltext erzeugt

[23] Eine solche Erweiterung findet sich bei der in Abschnitt 4.6 beschriebenen Methode zur Auflösung von *Intents*.

Intermediate Representation (IR)

🖮 com.ibm.wala.ssa.IR
🖮 com.ibm.wala.classLoader.JavaLanguage.JavaInstructionFactory

Bei einer *intermediate Representation* handelt es sich um eine von der Quellsprache unabhängige Darstellung bei einem *Compile-Vorgang*. Der Terminus wurde bei *WALA* übernommen. Eine *intermediate Representation* wird zu jeweils einer Funktion aus ihrem *CGNode* oder mittels einer *Factory* aus den *IMethods* gebildet und enthält die im Funktionskörper vorkommenden Befehle, welche ihrerseits mit einem *Program Counter* oder *Instruction Index* versehen sind.

Befehle in *WALAs IR* sind sehr nahe an *Java-Bytecode* orientiert. Gegenüber diesem liegen die Variablen bei *WALA* allerdings in *SSA-Form* (siehe Unterabschnitt 2.2.4) vor. Instruktionen sind innerhalb der *IR* in einem *Kontrollflussgraph* (siehe Unterabschnitt 2.1.3) aus *Grundblöcken* (siehe Unterabschnitt 2.1.2) organisiert [36].

Zusätzlich zu den Befehlen *Javas* existiert in *WALA* die Φ*-Instruktion*. Sie dient der Behandlung von *SSA* bei Codeverzweigungen (siehe Unterabschnitt 2.2.4). Sie lässt sich wie eine *Java-Instruktion* verwenden, wird allerdings nicht in der *Intermediate Reperesentation*, sondern zusammen mit den *Grundblöcken* in einer gesonderten Struktur hinterlegt.

Weitere Repräsentationen

WALA unterstützt weitere Darstellungsformen von Programmen, wie beispielsweise den *Abstrakten Syntaxbaum* oder den *Kontrollflussgraphen*. Auf diese Darstellungen sei hier nicht im Detail eingegangen, da dies für die Zielsetzung der Arbeit nicht relevant ist.

Es sei jedoch darauf hingewiesen, dass WALA eine eigene Form des *Systemabhängigkeitsgraphen* (siehe Unterabschnitt 2.1.7) zur Verfügung stellt. Dieser wird von *Joana* jedoch nicht verwendet. Stattdessen erstellt *Joana* einen eigenen erweiterten *SDG* (siehe Unterabschnitt 2.5.2).

2.4.2 Stubs

Es ist technisch weder praktikabel[24] noch sinnvoll[25] bei der Programmanalyse sämtliche Bibliotheken in ihrer Originalform mit einzubeziehen. Weiterhin stellen einige Sprachen mit sogenannten "*native Methoden*" Funktionalitäten zur Verfügung, für die im Programm keine "echte" Methode existiert. Aus diesen Gründen ist es üblich sogenannte *Stubs* zu erstellen.

[24] Werden die Original-Bibliotheken verwendet, so entsteht ein kaum zu deckender Ressourcenverbrauch.

[25] Systembibliotheken werden im Sinne der Analyse als "sicher" betrachtet. Eine wiederholte Miteinbeziehung bei der Analyse eines Programms ist daher nicht notwendig.

Diese *Stubs* ersetzen Funktionen des Originals derart, dass deren Datenflüsse (siehe Unterabschnitt 2.1.4) in hinreichender Form nachempfunden werden ohne jedoch die ursprüngliche Funktionalität zur Verfügung zu stellen. Der Ausdruck *Stub* wird sowohl für eine einzelne reduzierte Funktion, als auch für die reduzierte Form der gesamten Bibliothek verwendet.

Unterabschnitt 5.4.1 behandelt die Generierung der *Android-Stubs*, die für die Erstellung des *Lebenszyklusmodells* erforderlich sind.

2.4.3 Einlesen von Android-Apps

Um *Dalvik* einlesen zu können wurde *WALA* angepasst. Diese angepasste Variante findet sich im *Repository Joanas* verlinkt. Im folgenden seien Projekte aufgeführt, die inspirierend auf die Anpassungen eingewirkt haben.

Smali: Disassembler für Dalvik-Bytecode

Bei *Smali* [31] handelt es sich um ein Projekt, mit dem man zwischen *Dalvik-Dateien* und einem Textformat konvertieren kann. Von diesem Projekt wird mit *dexlib* eine Bibliothek zur Interpretation der *Bytecodesymbole Dalviks* zur Verfügung gestellt.

ScanDroid

Ein weiteres inspirierendes Projekt, welches *Statische Analysen*[26] von Adroid-Applikationen mit Hilfe von *WALA* erstellt, findet sich in *ScanDroid* [17]. Mittels *ScanDroid* lassen sich *Call Graphen* (siehe Unterabschnitt 2.2.1) erstellen und *Komponentenabhängigkeiten* auslesen. Eine *Komponentenabhängigkeit* entsteht, wenn eine Applikation auf Daten, welche von einer anderen Applikation bzw. von einem *System Service* zur Verfügung gestellt werden, zugreift.

2.5 Joana, Java Object-sensitive Analysis

Bei einer *Information Flow Control-Analyse* (*IFC-Analyse*) wird ein Programm daraufhin untersucht, ob es Informationsflüsse als sicher eingestufter Daten in öffentliche Kanäle gibt. Weiterhin kann mit dieser Technik sichergestellt werden, dass keine unverifizierten Eingaben kritische Berechnungen beeinflussen können. Durch diese Technik wird eine feiner granulierte Rechteprüfung ermöglicht, als mit konventionellen Zugriffskontrollmechanismen: Mit diesen könnte man beispielsweise einem E-Mail-Programm den Zugriff auf das Adressbuch und das Internet erlauben, hätte aber keine Informationen darüber, ob das Programm nun das komplette Adressbuch im Internet veröffentlicht.

[26]Eine *Statischen Analyse* (*Offline Analyse*) beruht darauf Aussagen über Programme zu treffen ohne diese dabei auszuführen.

Joana [18] ist ein Werkzeug mittels dem man eine *IFC-Analyse* auf *WALA*-lesbaren Programmen[27] durchführen kann. Das Programm wird dabei als Ganzes betrachtet (*Whole Program Analyse*).

Zunächst generiert *Joana* einen *Systemabhängigkeitsgraph*. Anschließend kann man Funktionen und Variablen mit Sicherheitseinstellungen, dem sogenannten *Security Label*, manuell annotieren. In der darauf folgenden Analyse traversiert *Joana* den *Systemabhängigkeitsgraph* mit einer an das *Slicing* angelehnten Technik. Neben *direkten* erkennt *Joana* so auch *indirekte Informationslecks*, welche durch von als sicher eingestuften Daten hervorgerufenen Programmverzweigungen mit öffentlicher Ausgabe entstehen können. Sogar Sicherheitsverletzungen, welche durch unterschiedliches Laufzeitverhalten in Programmen mit Parallelverarbeitung auftreten können, sind durch *Joana* erkennbar.

Das Ergebnis einer Sicherheitsbewertung mittels *Joana* ist *konservativ approximiert*, was bedeutet, dass ein Programm als sicher einzustufen ist, sofern *Joana* keine Verletzungen der Sicherheit findet. Treten jedoch Funde auf so bedeutet dies im Umkehrschluss nicht zwangsläufig, dass das analysierte Programm unsicher ist. *Joana* trifft einige Vorkehrungen um die Anzahl der Fehlalarme zu reduzieren: Durch eine *Points-To Analyse* (siehe Unterabschnitt 2.2.2) werden interprozedurale Seiteneffekte berücksichtigt, durch eine *Exception Analyse* fallen Pfade weg, die auf Ausnahmen beruhen, die nie auftreten können. Durch *Kontext-*, *Objekt-* und *Feldzugriffssensitivität* wird zwischen unterschiedlichen Instanzen der jeweiligen Entitäten unterschieden und das Ergebnis somit weiter präzisiert. Zusätzlich zu den Daten des *Systemabhängigkeitsgraphen* berücksichtigt *Joana* weiterhin den *Kontrollfluss*[28].

2.5.1 Benutzerschnittstellen Joanas

Joana verfügt über mehrere Benutzerschnittstellen. In den folgenden Abschnitten sei auf die einzelnen Komponenten eingegangen. Allen gemein ist jedoch, dass man ihnen bei ihrer Ausführung mittels *Javas "–Xmx"-Option* mehr Speicher zusichern sollte, als *Java* das standardmäßig vergibt.

IFC-Console

🖴 joana.ui.ifc.wala.console

Die *IFCConsole* ist die Hauptoberfläche *Joanas*. Sie existiert in einer graphischen und einer interaktiven *CLI-Variante*. Mit diesem Programm lässt sich der *Systemabhängigkeitsgraph* für *Java-Anwendungen* erstellen. Alternativ kann man einen vorhandenen *SDG*, welcher beispielsweise mit der Anwendung *JoDroid* (siehe unten) erstellt wurde, laden.

[27] Manche Komponenten Joanas können eine an Java angelehnte Nomenklatur für Typen und Klassen erwarten. Derzeit ist diese allerdings bei allen *WALA*-lesbaren Formaten konsistent.

[28] Viele Implementierungen des *Systemabhängigkeitsgraphen* fügen zwar *Kontrollflusskanten* in ihren Graphen ein, sie sind jedoch nicht Teil der Definition eines *SDG*

Abbildung 2.12: *Screenshot der IFC-Console*

Anschließend kann ein Sicherheits-Verband gewählt werden. Durch diesen wird angegeben, welche *Security Labels Joana* verwenden soll und wie sich diese zueinander verhalten. Standardmäßig benutzt *Joana* zwei *Labels*: *Low* und *High*.

Im Reiter "*Annotations*" kann den gefundenen Funktionen ein *Security Label* zugewiesen werden. Interessant sind hier vor allem die Einstellungen "*secure source*" und "*insecure sink*": Daten einer als sicher eingestuften Quelle dürfen nicht zu unsicheren *Senken* propagiert werden.

Durch das Feld "*Analysis type*" im Reiter "*Run*" kann schließlich eingestellt werden, wie *Joana* mit Nebenläufigkeiten bei parallel arbeitenden Anwendungen umgeht. Der Wert "*LSOD*" ist hier die konservativste Einstellung. Verwendet das zu analysierende Programm lediglich einen *Thread* ist diese Einstellung unerheblich.

Graphviewer

joana.ui.ifc.sdg.graphviewer

Graphviewer ist eine graphische Anwendung zur Visualisierung des von *Joanas Systemabhängigkeitsgraphen* (siehe Unterabschnitt 2.5.2). Sie gibt zunächst einen

Aufrufgraphen aus. Ausgehend von dort lassen sich dann die *SDGs* der einzelnen Funktionen ausgeben. Somit ist es möglich deren innere Struktur zu betrachten.

Graphviewer wurde bei der Umsetzung des Modells herangezogen um herauszufinden, ob Schleifen richtig platziert, Aufrufe korrekt aufgelöst und Parameter wie gewünscht zugewiesen wurden.

2.5.2 Joanas SDG-Format

Joana speichert seinen *Systemabhängigkeitsgraph* in einer Datei mit der Endung *.pdg*. *Joanas SDG-Format* erweitert den in Unterabschnitt 2.1.7 beschriebenen *Systemabhängigkeitsgraph*. Er wird unabhängig von *WALAs SDG* erstellt, obwohl *WALA* als Datengrundlage verwendet wird.

Knoten *Joanas SDGs* haben einen *Typ* und einen *Untertyp*, dort genannt *Operation* (*"O"*). Attribuiert sind sie weiterhin mit einem beschreibenden *Freitextfeld* (i*Value* - *"V"*), numerischer Funktionszugehörigkeit (*"P"*), ihrer Position in den Quellen (*Source position* - *"S"* und *Bytecode Index* (*BCI*) - *'B'*) und einer optionalen *Paketangabe* (*Component* - *"C"*).

Kanten werden im Knoten ihres Fußpunkts (ihrem *Parent*) hinterlegt. Die wichtigsten Kanten für *Kontrollabhängigkeiten* (siehe Unterabschnitt 2.1.3) sind *"CD"* und *"CE"*. Sie stehen jeweils für eine *bedingte Kontrollabhängigkeit* (*"CD"*) oder eine *allgemeine Kontrollabhängigkeit* (*"CE"*).

Kanten für *Datenabhängigkeiten* (siehe Unterabschnitt 2.1.4) finden sich unter anderem in der *Heap Datenabhängigkeit* (*"DH"*) und in *"DD"* für Abhängigkeiten auf dem *Stack*.

Bei Funktionsaufrufen finden sich *Call* Kanten ("CL") und an den zugehörigen Knoten für Parameter-Ein- und Ausgänge die Kanten *"PI"* und *"PO"* für *"Parameter In"* und *"Parameter Out"*.

Es existieren weitere Knoten- und Kantentypen, auf welche an dieser Stelle nicht weiter eingegangen wird, da sie für diese Arbeit von untergeordneter Relevanz sind.

KAPITEL 3

Aufbau des Modells

Wie aus Abschnitt 2.3 hervorgeht besitzt eine Android-Applikation, anders als ein herkömmliches *Java-Paket*, mehrere Einsprungspunkte. Da viele Analysen jedoch nicht auf diese Eigenschaft hin ausgelegt sind, soll im Folgenden ein Modell zur *Kapselung* dieses Verhaltens gefunden werden.

Im Verlauf der Modellierung wird eine Funktion zu der Applikation hinzugefügt werden, welche Aufrufe der Einsprungspunkte in einer Weise vornimmt, dass durch sie ein einzelner Einsprungspunkt synthetisiert wird.

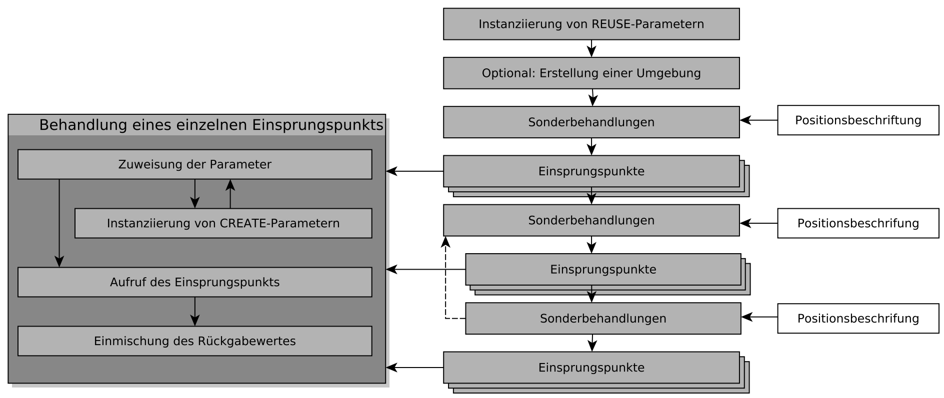


Abbildung 3.1: *Grober Aufbau eines Modells*

Abbildung 3.1 zeigt den groben Aufbau eines Modells: Gefundene Einsprungspunkte (siehe Abschnitt 3.1) liegen sortiert (siehe Abschnitt 3.3) vor. An beschrifteten Positionen zwischen den Einsprungspunkten können Sonderbehandlungen, wie die Erstellung einer Schleife, eingefügt werden (siehe Abschnitt 3.4).

Parameter zu den Einsprungspunkten können unterschiedlich behandelt werden, wie in Abschnitt 3.2 beschrieben wird. Nach dem Aufruf eines Einsprungspunk-

tes kann dessen Rückgabewert in bestimmte Arten später verwendeter Parameter eingemischt werden.

3.1 Auffinden der Einsprungspunkte

Vor der Erstellung des Modells müssen zunächst alle *Einsprungspunkte* der Applikation gefunden werden.

Zunächst werden Einsprungspunkte, durch die eine Applikation gestartet werden kann, berücksichtigt. Diese überladen Funktionen der Klassen *Activity*, *Service*, *BroadcastReceiver* oder *ContentProvider*. Die überladenen Klassen sind – bis auf den *BroadcastReceiver*, der alternativ auch zur Programmlaufzeit registriert werden kann – in *AndroidManifest.xml* (siehe Unterabschnitt 2.3.8) der Applikation aufgeführt. Dort wird diesen Klassen ebenfalls ein *IntentFilter*, also eine Spezifikation welche *Intents* (siehe Unterabschnitt 2.3.5) diese Klasse verarbeitet, zur Verfügung gestellt.

Zum Auffinden der Einsprungspunkte wird das *Manifest* zunächst nicht herangezogen. Stattdessen werden sämtliche Methoden der Applikation mit einer Liste fest codierter Einsprungspunkte abgeglichen. Anhang A enthält die Liste der fest hinterlegten Einsprungspunkte.

Optional werden weitere Methoden anhand einer Heuristik auf ihrer Signatur als Einsprungspunkt, wie in Unterabschnitt 3.1.2 beschrieben, selektiert.

3.1.1 Weitere zu berücksichtigende Funktionen

Es existieren weitere *CallBack*-Funktionen, welche durch das Betriebssystem aufgerufen werden können, die jedoch nicht den Start eine Applikation verursachen. Ein Beispiel hierfür wäre die Änderung der geographischen Position des Gerätes. Es erscheint sinnvoll solche Funktionen in der Modellierung analog zu Einsprungspunkten zu verarbeiten. Eine feste Codierung weniger Signaturen wurde vorgenommen, es zeigt sich aber, dass dieses Vorgehen aufgrund ihrer hohen Anzahl unpraktikabel ist.

3.1.2 Heuristische Selektion von Einsprungspunkten

Aufgrund der Vielzahl potentiell zu berücksichtigender Methoden wurde eine zusätzliche Selektion mittels einer Heuristik hinzugezogen.

Ein Großteil der Einsprungspunkte weist einen Namen auf, welcher mit "*on*" gefolgt von einem Großbuchstaben beginnt. Da dies jedoch nicht auf alle Einsprungspunkte zutrifft fließt diese Eigenschaft in der aktuellen Implementierung nicht mit in die Selektion ein.

Stattdessen werden alle Funktionen gewählt, welche eine Funktion der einschlägigen Einsprungspunkt-Klassen (oder optional aller Klassen der *Android-API*) überladen beziehungsweise implementieren. Diese Einstellung scheint auch von daher

sinnvoll, dass sie unintendierte Verhalten abdeckt, beispielsweise den Aufruf der *toString*-Methode durch eine *Logging-Funktion*.

Für eine so selektierte Funktion ist zunächst keine Information verfügbar, an welcher Position sie innerhalb des Modells aufgerufen werden soll (siehe Abschnitt 3.3). Eine Schätzung der Position könnte anhand dessen getroffen werden, ob die Methodennamen Teilstrings wie *Create* oder *Start* enthalten.

3.2 Instanziierungsverhalten für Parameter

Dieser Abschnitt befasst sich mit dem Umgang der von den *Einsprungspunkten* erwarteten Parameter. Eine genaue Steuerung dieser Parameter wirkt sich positiv auf das Ergebnis einer späteren Analyse aus. Für übergebene Variablen wird zunächst auf ihren Gültigkeitsbereich eingegangen, anschließend erfolgt eine Betrachtung, womit diese initialisiert werden.

Gültigkeitsbereich der Variablen

Der Gültigkeitsbereich von Referenzvariablen beeinflusst, inwiefern diese genutzt werden können um Daten zwischen Einsprungspunkten (beispielsweise durch Feldzugriffe) auszutauschen.

Im Modell sind aktuell zwei Einstellungen zur Steuerung der Gültigkeit vorgesehen: Die Einstellung *CREATE* bewirkt, dass zu einem Datentyp bei jedem Auftreten dieses Typs als Parameter eine neue Variable zur Übergabe generiert wird. Somit werden keine Datenabhängigkeiten zwischen Einsprungspunkten induziert. Dieses Verhalten bietet sich beispielsweise für den Typ *Landroid/view/KeyEvent;* an: Ein Event-Handler kann dieses Event beliebig anpassen, soll dadurch allerdings keine anderen Handler beeinflussen.

Die andere Einstellung bildet *REUSE*. Hier wird eine Instanz zu einem Datentyp erstellt, welche für das gesamte Modell Gültigkeit hat und für alle auftretenden Parameter passenden Typs verwendet wird. Da diese Instanz somit von allen Einsprungspunkten beeinflusst werden kann entstehen Abhängigkeiten zwischen diesen. Außerdem werden Rückgabewerte von Einsprungspunkten in diese Variablen eingemischt sofern die jeweiligen Typen zuweisungskompatibel sind. Ein Beispiel für die sinnvolle Verwendung von *REUSE* ist der Typ *Landroid/database*.

Die Einstellung *REUSE* stellt somit die konservative Einstellung dar: Die Einstellung *CREATE* erstellt keine Datenabhängigkeiten, die Einstellung *REUSE* erstellt deren potentiell zu viele.

In der späteren Implementierung existieren zusätzlich noch die Einstellungen *INHERIT*, durch die explizit angegeben wird, dass die Einstellung der Oberklasse übernommen werden soll, und *DEFAULT* zur Übernahme der Einstellung, welche für die gesamte Applikation als Standard gesetzt wurde.

Weitere Einstellungen, welche Kontext oder Klassennamen berücksichtigen wären denkbar. Ebenfalls könnte man mit einer Verhaltenseinstellung die Verwendung

von *null* erzwingen. In der aktuellen Implementierung wird dies allerdings nicht berücksichtigt.

Instanziierung der Variablen
Handelt es sich bei dem Typ des Parameters um eine Klasse, so muss diese entsprechend instanziiert werden. Die Wahl des hierfür verwendeten Konstruktors richtet sich nach der Anzahl der benötigten Parameter und deren Typen. So wird jedem Konstruktor eine Punktzahl zugeordnet und abschließend der am günstigsten erscheinende gewählt.

Die Parameter der jeweiligen Konstruktoren werden rekursiv instanziiert.

Handelt es sich bei dem Datentyp der Variable um ein Interface oder eine abstrakte Klasse, so wird eine Instanz aller implementierenden beziehungsweise ableitenden Klassen generiert. Diese einzelnen Instanzen werden der Variable über eine Φ-Funktion dann derart zugewiesen, dass sie von allen Instanzen datenabhängig ist.

Ist der zu instanziierende Typ ein Array, so wird hierfür ein Array, welches ein einziges Element entsprechenden Typs enthält erstellt.

3.3 Sortierung der Einsprungspunkte

Die gefundenen Einsprungspunkte sollen im späteren Modell in einer solchen Reihenfolge aufgerufen werden, dass der Lebenszyklus von Android-Anwendungen hinreichend genau abgebildet wird. Zu diesem Zweck wird eine Halbordnung auf den Einsprungspunkten eingeführt.

Da zwischen einigen Einsprungspunkten aber auch eine Verhaltensänderung des Modells nötig werden kann, wird zunächst eine Struktur aus Ablagepunkten generiert: Einige festgelegte Positionen innerhalb der Ausführungsreihenfolge seien im Folgenden als *Label* bezeichnet. Anders als bei Programmiersprachen üblich muss nach einem *Label* keine Anweisung folgen. Beim Überschreiten eines *Labels* können in der Umsetzung des Modells Sonderbehandlungen, wie das Einfügen einer Schleife, ausgeführt werden. Prinzipiell kann jede Position, repräsentiert durch einen Einsprungspunkt oder lediglich abstrakt, als *Label* verwendet werden. Die spätere Implementierung setzt jedoch stark auf den in Listing 9 festgelegten Satz fest codierter *Labels*.

$$\text{AT_FIRST} \preceq \text{BEFORE_LOOP} \preceq \text{START_OF_LOOP} \preceq \text{MIDDLE_OF_LOOP} = \text{DEFAULT} \preceq \text{MULTIPLE_TIMES_IN_LOOP} \preceq \text{END_OF_LOOP} \preceq \text{AFTER_LOOP} \preceq \text{AT_LAST}$$

Listing 9: *Codierte Labels*

Der Bereich zwischen zwei *Labels* sei im Folgenden als *Sektion* bezeichnet. Eine *Sektion* sei nach dem Einleitenden *Label* benannt. Das jeweilige einleitende *Label* sei

jeweils erstes Element seiner *Sektion* (jedoch kein Element der vorherigen Sektion). *Sektionen* dienen schließlich zur Aufnahme von Einsprungspunkten. Die Angabe der Positionierungsinformation eines Einsprungspunktes erfolgt relativ zu *Labels* oder anderen *Einsprungspunkten*.

In der Modellierung werden die Einsprungspunkte nicht nach den implementierenden Klassen sortiert, das heißt es wird von allen *Activitys* und *Services* nacheinander beispielsweise *onCreate* aufgerufen, danach von allen beispielsweise *onStart*. Eine derartige Umsetzung ist möglich, da alle *Activitys* und *Services* gleichberechtigt und zunächst unabhängig voneinander sind. Der Start weiterer Komponenten aus einer Komponente heraus, durch den Abhängigkeiten entstehen können, wird in Abschnitt 3.7 getrennt betrachtet.

3.4 Grundstruktur des Modells

Durch die Grundstruktur des Modells wird geregelt, welche Sonderbehandlungen in das Modell eingefügt werden. Es lässt sich darüber also steuern, wie das Modell den *Lebenszyklus* widerspiegelt. Die folgenden Strukturen sind derzeit vorgesehen:

Rein sequentiell
Diese Variante hat wenig mit den realen Gegebenheiten zu tun. Sie kann herangezogen werden, wenn eine schnelle Analyse gewünscht ist oder um sie als Grundlage eigener Strukturen abzuleiten.

Mit emulierter Benutzerinteraktion
Diese Variante ruft in einer Schleife *CallBack*-Funktionen auf, die einer Statusänderung des Geräts zugeordnet sind.

Mit externer Beeinflussung
Durch eine weitere Schleife wird hier das Starten und Beenden von Komponenten der Applikation durch externe Applikationen[1] nachempfunden.

Mit Neustart der Applikation
Bei Android verbleiben Applikationen zunächst auch dann im Speicher, wenn sie nicht mehr sichtbar sind. Erst bei Speicherknappheit werden sie beendet. Für den nächsten Start können Daten in einem sogenannten *instanceState* hinterlegt werden. Dieses Modell empfindet einen derartigen Neustart nach.

Die hier dargestellten Strukturen sind sehr leicht anpassbar.

[1] Der interne Start von Komponenten wird bereits durch Abschnitt 3.7 abgedeckt

3.5 Nachbearbeitung durch den Anwender

Aufgrund der Vielzahl von *CallBack*-Funktionen und der Unterschiedlichkeit von Programmen ist es fraglich, ob eine komplett automatisierte Erstellung des Modells immer alle Sachverhalte wie gewünscht abbildet. Aus diesem Grund ist es sinnvoll es dem Benutzer vor der eigentlichen Codegenerierung zu ermöglichen die Umsetzung nach seinen Wünschen anzupassen. In diesem Umfang sollen weitere Einsprungspunkte hinzufüg- oder entfernbar gemacht werden. Außerdem soll die Reihenfolge der Aufrufe anpassbar sein. Weiterhin lassen sich Verhalten für *Intents* und die *Instanziierung* ändern.

3.6 Synthese des Modells zu einem Einsprungspunkt

In diesem Schritt wird ein neuer virtueller Einsprungspunkt erstellt, der die anderen Einsprungspunkte im Rahmen der zuvor definierten Randbedingungen aufruft. Abbildung 3.2 (auf Seite 46) stellt die durchgeführten Aktionen graphisch dar.

Im Umfang der Generierung werden zunächst *Instanzen* der als *REUSE* gekennzeichneten Typen erstellt[2].

Anschließend wird über die sortierte Liste der Einsprungspunkte iteriert. Dabei wird zunächst geprüft, ob ein *Sektionswechsel* vorliegt, also ein *Label* überschritten wurde. In diesem Fall wird zunächst die mit dem *Label* assoziierte Sonderbehandlung in die Funktion mit aufgenommen.

Anschließend wird eine *Instanz* der jeweiligen *CREATE*-Parameter zu einem Einsprungspunkt erstellt. Im Falle von *Interface-Parametern* kann das die Generierung zusätzlichen Codes nach sich ziehen.

Nach dem Ende der Iteration muss abschließend geprüft werden, ob zu Sonderbehandlungen noch offene Aufgaben vorliegen: Dies geschieht, wenn in den abschließenden *Sektionen* keine Einsprungspunkte liegen. Abschließender Code wird in die synthetisierte Funktion mit aufgenommen.

3.7 Anpassung des Starts von Intents

Jeder Start einer Komponente Androids beruht auf einem *Intent* (siehe Unterabschnitt 2.3.5). Dieser *Intent* wird anschließend durch eine artspezifische Funktion

[2] In der späteren Implementierung in WALA werden REUSE-Variablen zu Parametern der synthetisierten Funktion und somit von WALA selbst instanziiert

– beispielsweise *startActivity* für *Intents*, welche auf eine *Activity* zeigen – gestartet. Eine Zuordnung solcher Start-Funktionen zu Komponententypen findet sich in Anhang D.

Da der Code zum Start von Komponenten nicht Teil der verwendeten *Stubs* ist und da in einer statischen Analyse das *Manifest* nicht in hinreichender Form einlesbar wäre, muss der Start von Komponenten durch das Modell gesondert behandelt werden. Diese Behandlung fußt zunächst darin, dass die Start-Funktion durch eine *Wrapper-Funktion* ersetzt wird.

Für den Aufbau der *Wrapper-Funktion* ist zunächst grundsätzlich die Information verfügbar, welche Komponententypen als Ziel der Start-Funktion überhaupt in Frage kommen. Anhand dessen kann so schon eine Einschränkung auf dem Modell getroffen werden.

Ist ein Aufrufkontext vorhanden, so können weiter Einschränkungen auf dem Modell stattfinden: Die Auswahl der Funktionen, welche durch den *Wrapper* aufgerufen werden, richtet sich nach den vorhandenen Informationen des *Intent Objekts*, welches an die Start-Funktion übergeben wird. Ist die eindeutige Zuordnung zu einer Klasse möglich, so wird ein neues Modell, welches auf diese Klasse beschränkt ist, generiert. Kann hingegen sichergestellt werden, dass das Ziel des Aufrufs innerhalb einer Applikation liegt, welche außerhalb des Umfangs der Analyse liegt, so wird eine eigens dafür vorgesehene Funktion aufgerufen.

Liegt keine Information über das *Intent* vor, so verbleibt als Information der Name der Start-Funktion. Anhand dieser Information wird ein neues Modell generiert, welches beispielsweise lediglich *Activities* aufruft. Anschließend wird die zuvor erwähnte Funktion für externe Ziele aufgerufen.

3.7.1 Funktion für externe Ziele

In dem Versuch das Verhalten einer externen Applikation nachzuempfinden wird zunächst lesend auf alle eingehenden Parameter zugegriffen. Ist ein Rückgabewert erwünscht, so wird dieser aus den Eingangsparametern erstellt sodass potentielle Datenabhängigkeiten bestehen bleiben. Abhängig von der Art des Aufrufs muss weiterhin der Einsprungspunkt *onActivityResult* des Aufrufers angesprungen werden.

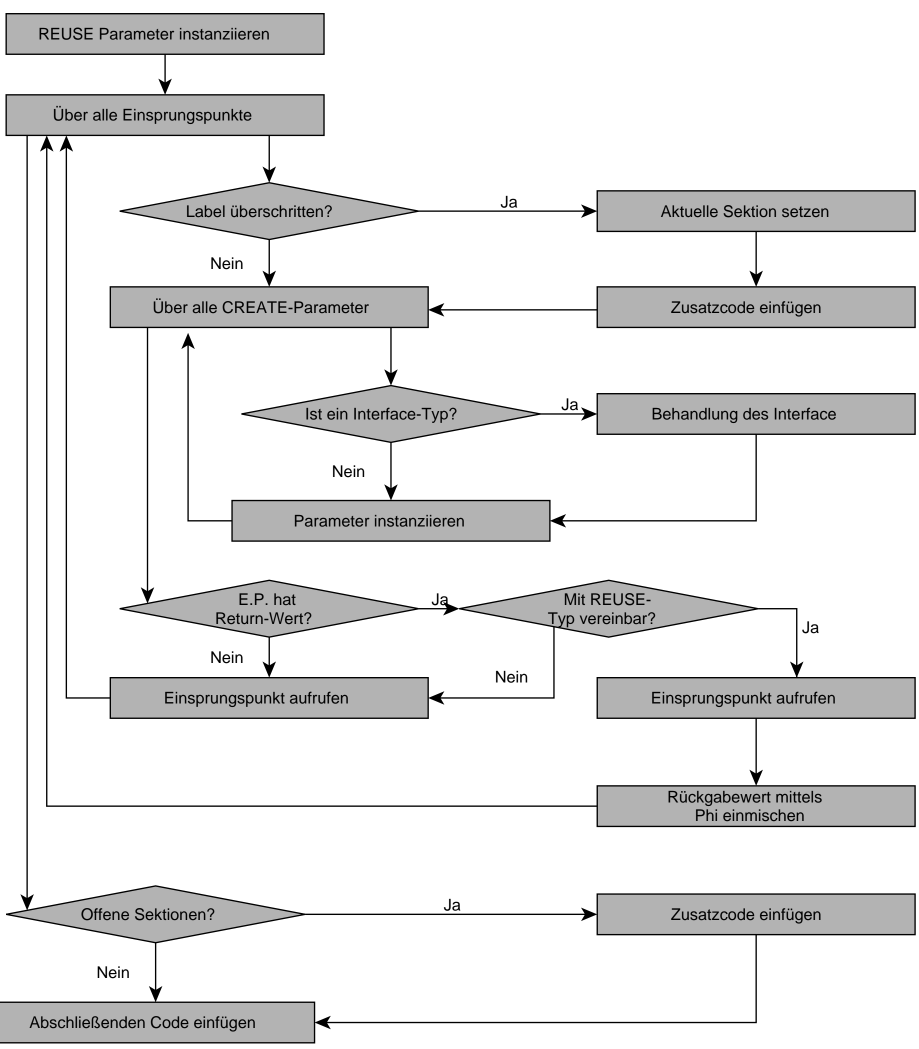


Abbildung 3.2: *Synthese des Einsprungspunktes*

KAPITEL 4

Umsetzung des Modells

In Kapitel 3 wurde der grundlegende Aufbau des Modells dargestellt. Dieses Kapitel befasst sich nun mit dessen Umsetzung in *WALA*.

Bei der Implementierung wurde Wert auf Modularität gelegt, sodass einzelne Komponenten des Modells leicht austauschbar sind und es somit gut für weitere Analyseverfahren anpassbar ist. Es wurde darauf geachtet konsequent Randbedingungen von Funktionen zu prüfen um Fehler nicht zu verschleppen; auch sind hinzugefügte Funktionen durchgängig mit *Javadoc* annotiert um eine *API-Dokumentation*, welche mittels dieses Werkzeugs oder *Doxygen* erstellt werden kann, mit nützlichen Informationen zu füllen.

Praktische Anwendung findet das so umgesetzte Modell schließlich in *JoDroid*.

Die Anordnung der jeweiligen Unterkapitel wurde zur besseren Vergleichbarkeit wie in Kapitel 3 gewählt.

4.1 Auffinden der Einsprungspunkte

com.ibm.wala.dalvik.util.AndroidEntryPointLocator
com.ibm.wala.dalvik.util.AndroidEntryPointManager

Zum Auffinden der Einsprungspunkte wird mittels *WALA* aus dem zu analysierenden Programm und hinzugeladenen *Stubs* (siehe Unterabschnitt 2.4.2) zunächst eine Klassenhierarchie (siehe Unterabschnitt 2.4.1) erstellt.

Anschließend wird über alle dort hinterlegten Funktionen iteriert und deren Signatur mit einer fest codierten Liste bekannter Signaturen verglichen. Die Signaturen der wichtigen Android-Klassen *Activity*, *Service*, *BroadcastReceiver* und *ContentProvider* finden sich in Anhang A.

Grundsätzlich werden alle Funktionen selektiert, welche eine dort aufgeführte Funktion überschreiben. Setzt man bei der Suche das *Flag WITH_SUPER* so werden

für die jeweiligen Instanzen der Klassen auch die Funktionen selektiert, die nicht überschrieben wurden, also auf der Oberklasse aufgerufen werden.

Der Konstruktor der Komponenten-Klassen enthält in der Regel keinen zu berücksichtigenden Code. Er wird bei der Instanziierung des Objektes natürlich dennoch aufgerufen. Über das *Flag WITH_CTOR* lässt sich sein Aufruf, so gewünscht, weiterhin explizit zu dem Modell hinzufügen.

Durch das Setzen des *Flags INCLUDE_CALLBACKS* werden die in Anhang B aufgeführten Signaturen weiterer Klassen analog zu den Signaturen der Klassen behandelt, die die eigentlichen Android-Komponenten repräsentieren.

Passt die Signatur einer Methode auf keine der fest hinterlegten wird abhängig von den *Flags EP_HEURISTIC* und *CB_HEURISTIC* die in Unterabschnitt 4.1.1 beschriebene Heuristik angewandt. Dabei bedeutet *EP_HEURISTIC*, dass sich die Heuristik auf Klassen beschränkt, für die auch fest codierte Einsprungspunkte existieren. Bei der Einstellung *CB_HEURISTIC* wird diese Einschränkung nicht vorgenommen.

Schließlich werden alle selektierten Funktionen in der *Singleton-Klasse AndroidEntryPointManager* hinterlegt.

Das *Manifest* (siehe Unterabschnitt 2.3.8) wird bei der Selektion der Einsprungspunkte somit zunächst nicht berücksichtigt, findet allerdings bei der späteren Zuordnung von *Intents* zu den *Einsprungspunktklassen* Verwendung.

Anpassung der Liste bekannter Signaturen

com.ibm.wala.dalvik.util.androidEntryPoints.*
com.ibm.wala.dalvik.util.AndroidEntryPointLocator

Die fest codierten Signaturen möglicher Einsprungspunkte finden sich im Java-Paket *androidEntryPoints*. Listing 10 zeigt einen Auszug aus einer dort hinterlegten Klasse: *Einsprungspunktsignaturen* sind dort jeweils als Konstante hinterlegt, wodurch ihre Referenzierung bei Angabe der Positionierungsinformationen erleichtert werden soll.

Der in Listing 10 dargestellte Code definiert also *Activity.onPause* als möglichen Einsprungspunkt. Jeder fest codierte mögliche Einsprungspunkt ist mit einer Positionierungsinformation behaftet. Auf Positionierungen wird in Abschnitt 4.3 eingegangen.

Die Funktion *populate* fügt diese lokal abgelegten Signaturen schließlich zu der Liste bekannter Signaturen hinzu. Fügt man neue Klassen hinzu, so muss ihre *populate*-Funktion in *AndroidEntryPointLocator.populatePossibleEntryPoints* hinzugefügt werden.

```
public final class ActivityEP {
    // ...
    public static final AndroidPossibleEntryPoint onPause =
        new AndroidPossibleEntryPoint(
            AndroidEntryClassName.ACTIVITY,
            "onPause",
            ExecutionOrder.between(
                new AndroidEntryPoint.IExecutionOrder[] {
                    onCreate,
                    ExecutionOrder.MIDDLE_OF_LOOP
                },
                onDestroy
            ));
    // ...
    public static void populate( List<AndroidPossibleEntryPoint> possibleEntryPoints) {
        possibleEntryPoints .add(onPause);
        // ...
    }
}
```

Listing 10: *Eintrag einer berücksichtigten EntryPoint-Signatur*

Einlesen des Manifest

com.ibm.wala.dalvik.util.AndroidManifestXMLReader
com.ibm.wala.dalvik.util.AndroidEntryPointManager

Das Einlesen des *Manifest* ist für die Generierung des Modells zunächst nicht zwingend erforderlich, erlaubt jedoch eine genauere Auflösung der Ziele von *Start-Funktionen* (siehe Abschnitt 4.6).

Die Datei *AndroidManifest.xml* liegt in einer speziellen binärcodierten Form des XML-Formats vor. Leider konnte auf Grund von Versionskonflikten keine Bibliothek, welche diese Darstellung direkt liest, herangezogen werden (siehe Unterabschnitt 7.1.5). Die Datei muss somit durch den Anwender zuvor mittels des Programms *apktool* extrahiert werden.

Der *Parser* für das Einlesen der extrahierten Datei basiert auf dem *SAX-Parser*, abstrahiert jedoch etwas weiter von diesem um leichter erweiterbar zu sein.

Aus dem *Manifest* werden folgende Informationen ausgelesen:

- Der Paket-Name der Applikation
- Zur Verfügung gestellte Intents
- *Actions* von Intents (keine URI Informationen)
- Aliase von Intents

Die so gelesenen Daten werden in den entsprechenden Feldern des *AndroidEntryPointManager* hinterlegt.

4.1.1 Heuristische Selektion aus der Klassenhierarchie

🖮 com.ibm.wala.dalvik.util.AndroidEntryPointLocator

Es existieren zwei Arten von Heuristiken: Die erste – aktiviert durch das *Flag EP_HEURISTIK* – prüft für eine Methode, ob sie eine Methode aus einer der festgelegten Klassen[1] überschreibt. Ist dies der Fall ist die Methode als Einsprungspunkt zu selektieren. Dafür ist jedoch weiterhin die Angabe einer Position innerhalb des Modells von Nöten. Eine Funktion hierfür ist vorgesehen, diese gibt allerdings derzeit als Position grundsätzlich *DEFAULT* (welches auf *MUTLIPLE_TIMES_IN_LOOP* zeigt) zurück.

Die zweite Art selektiert alle Methoden, die eine Methode überschreiben, die sich im Paket *android* befindet oder ein *Interface* implementieren, welches in diesem Paket definiert ist. Die Positionierung der Funktion innerhalb des Modells erfolgt dann analog zu der ersten Variante.

4.2 Instanziierungsverhalten für Parameter

🖮 com.ibm.wala.dalvik.ipa.callgraph.androidModel.parameters.IInstantiationBehavior
🖮 com.ibm.wala.dalvik.ipa.callgraph.androidModel.parameters.DefaultInstantiationBehavior
🖮 com.ibm.wala.dalvik.ipa.callgraph.androidModel.parameters.LoadedInstantiationBehavior

Das Instanziierungsverhalten wird über eine *IInstantiationBehavior* gesteuert. Dieses *Interface* wird zum einen durch die *DefaultInstantiationBehavior* implementiert, welche einen Satz von fest codierten Regeln enthält; eine weitere Implementierung existiert mit *LoadedInstantiationBehavior*. Letztere findet Verwendung um benutzerdefinierte Verhaltensweisen zu hinterlegen. Die Unterschiede beider Implementierungen sind marginal.

Wie in Abschnitt 3.2 beschrieben wird von dem Instanziierungsverhalten einer der Werte *CREATE* oder *REUSE* zurückgegeben. Ist letzteres der Fall, so wird eine solche Variable als Parameter an das Modell übergeben. Da sich somit die Signatur des Modells ändert sind Änderungen an den Verhalten lediglich vor der Generierung eines Modells[2] möglich. Weitere Implementierungen sollten das berücksichtigen und derlei Änderungen verweigern.

Das aktuell gesetzte Instanziierungsverhalten findet sich in der Klasse *AndroidEntryPointManager*.

[1] Um welche Klassen es sich dabei genau handelt wird durch das *Flag INCLUDE_CALLBACKS* beeinflusst. Es handelt sich bei ihnen allerdings grundsätzlich um Klassen einer fest codierten Liste.

[2] Das Setzen der kontextfreien Overrides und die später beschriebene Ersetzung der Start-Funktionen ziehen Generiertung eingeschränkter Modelle nach sich.

Bei der Abfrage des Verhaltens wird der *IInstantiationBehavior* eine *IMethod* übergeben, für die das Verhalten zu einem Parameter erfragt wird. Die *IInstantiationBehavior* kann abhängig von dieser Methode statt *REUSE CREATE* zurückgeben, jedoch nicht umgekehrt[3].

Die Typen von Androidkomponenten sollten grundsätzlich die Einstellung *REUSE* aufweisen, da ihr Rückgabewert sonst nicht ausgelesen werden kann[4].

Weiterhin ist zu berücksichtigen, dass der von der *IInstantiationBehavior* zurückgegebene Wert lediglich Gültigkeit für eine Instanz des Modells hat. Wird ein eingeschränktes Modell durch eine Start-Funktion generiert (siehe Abschnitt 4.6), so wird auch für *REUSE*-Variablen eine neue Instanz erzeugt (es sei denn sie sind Parameter der Start-Funktion)[5].

Verhalten sollten lediglich für Referenz-Typen in der *IInstantiationBehavior* nachgeschlagen werden. Jedoch sollte eine Implementierung dieses *Interfaces* auch mit anderen Typen umgehen können.

Die Auswirkung der Einstellungen *CREATE* und *REUSE* wurde bereits in Abschnitt 3.2 beschrieben.

Das *Interface IInstantiationBehavior* weist eine weitere Funktion, *getExactness*, auf. Der Rückgabewert dieser Funktion hat aktuell keinerlei Auswirkung auf das spätere Modell. Derzeit wird er bei ausschließlich zur Formatierung der Ausgabe zur Nachbearbeitung durch den Benutzer verwendet.

Die Liste der in der *DefaultInstantiationBehavior* fest hinterlegten Verhalten findet sich in Anhang C.

4.2.1 Instanziierung der Parameter

🖮 com.ibm.wala.util.ssa.IInstantiator
🖮 com.ibm.wala.dalvik.ipa.callgraph.androidModel.parameters.Instantiator

Die Klasse *Instantiator* dient dazu Instanzen von Referenztypen in synthetischen Methoden gemäß Abschnitt 3.2 zu erstellen. Sie wird dabei über das *Interface IInstantiator* als Call-Back an Hilfsmethoden übergeben. Diese Klasse bildet einen Teil der im Rahmen dieser Arbeit entstandenen Toolbox für synthetischen Methoden und wird in Unterabschnitt 5.1.8 näher beschreiben.

[3]Soll zwischen *REUSE* und *CREATE* gewechselt werden, so muss der Rückgabewert wenn keine (*null*) Methode übergeben wurde auf *REUSE* lauten

[4]Die *AndroidPreFlightChecks* schreiben eine Warnung, falls das nicht der Fall ist

[5]Durch die Einstellung *doFlatComponents* lässt sich dieses Verhalten auf die Verwendung globaler Instanzen für Komponenten umstellen

4.2.2 Verwaltung der einzelnen Instanzen

🖮 com.ibm.wala.dalvik.ipa.callgraph.androidModel.parameters.AndroidModelParameterManager
🖮 com.ibm.wala.util.ssa.SSAValueManager

Aufgrund des Auftritts von Schleifen in dem Modell und zur Einmischung von Rückgabewerten von Funktionen ergibt sich die Notwendigkeit einer geeigneten Verwaltung von *SSA-Variablen*: Die jeweilig aktuelle Version sollte nachschlagbar sein; ebenso, ob bereits – wo nötig – alle Variablen durch Φ-Funktionen zusammengeführt wurden.

Zu diesem Zweck existiert eine Klasse *SSAValueManager* (wiederum aus der Toolbox, siehe Unterabschnitt 5.1.4). Der für das Modell verwendete *AndroidModelParameterManager* erweiterte die Fähigkeiten des *SSAValueManager* um den besseren Umgang mit *REUSE*-Parametern: Initial wird der Manager durch einen *ParameterAccessor* (Toolbox, siehe Unterabschnitt 5.1.3) mit solchen Variablen befüllt.

4.3 Sortierung der Einsprungspunkte

🖮 com.ibm.wala.dalvik.ipa.callgraph.impl.AndroidEntryPoint.IExecutionOrder
🖮 com.ibm.wala.dalvik.ipa.callgraph.impl.AndroidEntryPoint.ExecutionOrder

Vor der späteren Verarbeitung müssen die Einsprungspunkte anhand ihrer vorgesehenen Aufrufstelle sortiert werden. Dies geschieht anhand eines *Comparator*, der auf dem *Interface IExecutionOrder* definiert ist. Implementiert wird dieses Interface durch *AndroidEntryPoint*, *AndroidPossibleEntryPoint*, sowie von der Klasse *ExecutionOrder*, welche zusätzlich einige Hilfsfunktionen zur Generierung einer Position und Konstanten für die Position der einzelnen *Labels* (siehe Abschnitt 3.3) enthält.

Eine beispielhafte Zuweisung einer *ExecutionOrder* zu einer *Einsprungspunktsignatur* war bereits in Listing 10 (auf Seite 49) zu sehen. Positionierungen werden gemäß Anhang A vorgeschlagen. Die vorgeschlagene Reihenfolge berücksichtigt den Lebenszyklus (siehe Unterabschnitt 2.3.4) von Android-Anwendungen.

Ein Auszug der Hilfsfunktionen für die Positionierung findet sich in Listing 11. Durch die Funktion *between()* wird eine Position in der Mitte ihrer Parameter *before* und *after* gewählt, jedoch ohne *Sektionsgrenzen* zu beachten. Durch die Funktion *after(after)* wird eine Position zwischen dem Parameter *after* und dem Ende der *Sektion*, in der dieser liegt, gewählt. Weiterhin existieren die Funktionen *directlyBefore* und *directlyAfter*. Bei ihrer Verwendung ist zu beachten, dass der Versuch einen dritten Einsprungspunkt zwischen die so positionierten Einsprungspunkte zu

```
public class AndroidEntryPoint extends DexEntryPoint {
  // ...
  public static class ExecutionOrder implements Comparable<IExecutionOrder>,
          IExecutionOrder {
    // ...
    public final static ExecutionOrder START_OF_LOOP = new ExecutionOrder();
    // ...
    public static ExecutionOrder between(IExecutionOrder after,
              IExecutionOrder before);
    public static ExecutionOrder between(IExecutionOrder[] after,
              IExecutionOrder[] before);
    public static ExecutionOrder after (IExecutionOrder after);
    public static ExecutionOrder after (IExecutionOrder[] after);
    public static ExecutionOrder directlyAfter (IExecutionOrder after);
    public static ExecutionOrder directlyBefore (IExecutionOrder before);
    public ExecutionOrder getSection();
    // ...
  }
}
```

Listing 11: *Hilfsfunktionen von ExecutionOrder*

platzieren mit einer *ArithmeticException* scheitert, welche angibt, dass die Präzision der Implementierung für die gewünschte Positionierung nicht ausreicht.

Als jeweilige Parameter *before* und *after* können auch *Arrays* übergeben werden. Es wird daraus dann das Maximum beziehungsweise Minimum gebildet. Durch dieses Vorgehen soll verhindert werden, dass ein Einsprungspunkt durch Verschieben eines referenzierten Einsprungspunktes in eine nicht gewünschte *Sektion* fällt.

ExecutionOrder codiert die Position intern in einer *Integer-Zahl*. Aus diesem Grund kann sie eine *ArithmeticException* werfen, wenn die Präzision nicht mehr ausreichend zur Umsetzung der Positionierung ist.

4.4 Nachbearbeitung gefundener Einsprungspunkte

Die in Abschnitt 4.1 und Abschnitt 4.2 festgestellten Einsprungspunkte und *Instanziierungsverhalten* sollen nun durch den Nutzer nachbearbeitbar sein. Zu diesem Zweck ist in *JoDroid* das Lesen und Schreiben einer Datei zur Ablage dieser Informationen implementiert (siehe Unterabschnitt 4.4.2).

4.4.1 Nachbearbeitung ohne JoDroid

Möchte man die *WALA*-Implementierung des Modells ohne *JoDroid* benutzen, so lassen sich die Einsprungspunkte aus der *Singleton-Klasse AndroidEntryPointManager* auslesen.

Vor dem Auslesen das *Instanziierungsverhaltens* sollte man zunächst ein Modell des Android-Lebenszyklus generieren. Dadurch wird der interne Speicher der jeweiligen *IInstantiationBehavior* gefüllt. Danach lässt sich die Klasse wie bei Klassen, die *Serializable* implementieren üblich serialisieren[6]. Alternativ kann auch über die in den Einsprungspunkten auftretenden Typen iteriert und der Verhaltenswert jeweils explizit mittels *getBehavior* ausgelesen werden. Die Genauigkeit lässt sich mittels *getExactness* auslesen.

Will man eingelesene Einsprungspunkte in einer *IInstantiationBehavior* ablegen, so verwendet man dazu eine *LoadedInstantiationBehavior*. Anschließend registriert man diese in *AndroidModelEntrypointManager*.

4.4.2 Nachbearbeitung mit JoDroid

🖮 edu.kit.joana.wala.jodroid.entrypoints.*

Die empfohlene Verwendung *JoDroids* erfolgt in zwei Schritten[7]: Im ersten Schritt wird *JoDroid* mit der Option *–scan* aufgerufen. Nun werden Einsprungspunkte wie in Abschnitt 4.1 beschrieben gesucht. Anschließend wird das *Manifest* (so gegeben) eingelesen und ein vorläufiges Modell des Lebenszyklus generiert.

Das Auslesen und Schreiben einer applikationsspezifischen Konfigurationsdatei erfolgt durch die Klassen in dem Paket *jodroid.entrypoints*: Durch die jeweiligen *Serializer-Klassen* werden die zu schreibenden Daten entweder direkt oder über Reflektion aus den einzelnen Strukturen ausgelesen. Anschließend werden die Daten durch die Klasse *Marshal* in eine Dokumentstruktur geladen, die schließlich als XML-Datei ausgegeben werden kann. Die Klasse *Writer* fasst den nötigen Code zusammen. In Anhang E findet sich eine genaue Aufstellung in der Datei verfügbarer *Tags* und Attribute. Listing 12 zeigt einen stark gekürzten Auszug aus einer so generierten Datei.

Nun lassen sich Einsprungspunkte, *Instanziierungsverhalten* und Intentauflösung anpassen: *Einsprungspunkte* werden in der Reihenfolge, in der sie in der Datei Auftreten, modelliert. Funktionen können hier hinzugefügt, entfernt oder verschoben werden. Es existiert allerdings eine Beschränkung darin, dass Methoden, die auf der

[6] Der Wert des Feldes *serializationIncludesCache* der Klasse sollte dafür auf *true* gesetzt sein.
[7] Mittels der Option "–entrypoint @all" kann man es auch in einem Schritt unter Beibehaltung der Standardeinstellungen verwenden.

```
<model>
    <config of="com.ibm.wala.dalvik.util.AndroidEntryPointManager">
        <setting of="doBootSequence" type="boolean" value="false"></setting>
    </config>
    <intents>
        <intent name="Landroid/intent/action/MAIN" resolves="STANDARD_ACTION">
            <override name="Lde/.../MainActivity" resolves="INTERNAL_TARGET">
            </override>
        </intent>
    </intents>
    < instantiation  default="REUSE">
        <behaviour of="CREATE" type="Ljava/lang/Object" exactness="EXACT">
        </behaviour>
        <behaviour of="CREATE" type="Landroid/view/View" exactness="INHERITED">
        </behaviour>
    </ instantiation >
    <entrypoints>
        <section name="ExecutionOrder.AT_FIRST">
            <entrypoint type="ACTIVITY" call="MainActivity.onCreate()V">
            </entrypoint>
        </section>
        <section name="ExecutionOrder.AFTER_LOOP">
            <entrypoint type="ACTIVITY" call="android.app.Activity.onDestroy()V">
                <with param="this" type="L.../HopActivity"></with>
                <with param="this" type="L.../MainActivity"></with>
            </entrypoint>
        </section>
    </entrypoints>
</model>
```

Listing 12: *Auszug aus einer Datei zur Konfiguration des Modells*

Oberklasse aufgerufen werden[8], immer zeitgleich[9] aufgerufen werden müssen.

Sektionen, die keine Einsprungspunkte beinhalten sind nicht in der Datei aufgeführt. Man kann sie an dieser Stelle dennoch hinzufügen. Alle existierenden *Labels* sind in Abschnitt 3.3 aufgeführt.

Für eine erfolgreiche Analyse muss meist das *Instanziierungsverhalten* einiger Typen angepasst werden[10].

Unter dem Tag "*intents*" finden sich Informationen, wie Ziele in den Start-

[8] Das sind die Methoden, die ein "with"-Subtag aufweisen. Sie entstehen nur bei gesetztem Flag *WITH_ SUPER* siehe Abschnitt 4.1.

[9] "Zeitgleich" meint hier eigentlich "direkt hintereinander". Da sich allerdings keine weiteren Methoden zwischen den Aufrufen befinden dürfen kann man den Aufruf als zeitgleich betrachten

[10] Das *Instanziierungsverhalten WALAs* kann beispielsweise aktuell keine Instanz von *Object* erstellen

Funktionen (siehe Abschnitt 4.6) aufgelöst werden sollen. Bei einem einfachen Scanvorgang finden sich hier lediglich die in *AndroidManifest.xml* aufgeführten *Intents* wiedergespiegelt. Für die Erstellung einer kompletten Liste auftretender *Intents* müsste zunächst ein *Aufrufsgraph* generiert werden[11].

Die Inhalte der *config*-Tags werden derzeit nicht wieder eingelesen, sie befinden sich lediglich zu Informationszwecken in der Datei.

Die editierte Datei liest man anschließend mittels *jodroid –ep-file* wieder ein. Der Einlesevorgang selbst ist in der Klasse *Reader* gekapselt. Um weitere Tags zu verarbeiten ist das *Tags Enum* anzupassen und eventuell ein *ParserItem* anzulegen. Nähere Informationen dazu finden sich in *JavaDoc*.

4.5 Synthese des Modells zu einem Einsprungspunkt

Die Analyse eines Programms mittels *WALA* startet grundsätzlich bei einer sogenannten *FakeRoot*-Methode (im Fall von *Dalvik*-Applikationen *DexFakeRootMethod*). Diese wird durch einen *CallgraphBuilder* verwaltet und dient zur Herstellung eines globalen Status vor der eigentlichen Analyse. Von dort aus wird später der Einsprungspunkt der Applikation aufgerufen werden.

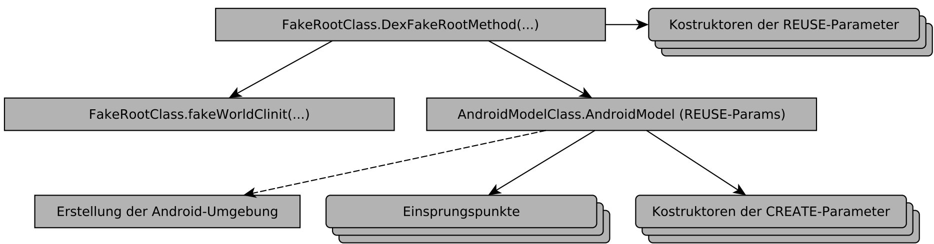


Abbildung 4.1: *Aufrufe zu Beginn einer Analyse mittels WALA*

Im Zuge der Herstellung des globalen Status wird zunächst eine Methode *fakeWorldClinit* aufgerufen. Durch diese wird sämtlicher statische Initialisierungscode, der in der Applikation auftritt aufgerufen.

Als nächstes soll das im Folgenden beschriebene Android-Modell[12] aufgerufen werden. Da dieses Modell die zuvor beschrieben *REUSE*-Typen als Parameter nimmt, werden diese zuvor durch die *FakeRoot*-Methode instanziiert.

Das hier aufgerufene Android-Modell wird sich im später generierten *Systemabhängigkeitsgraphen* unter dem Namen *com.ibm.wala.AndroidModelClass.Android*

[11] Hierfür sind *Points-To* Infromationen der an den Konstruktor des *Intents* übergebenen Parameter erforderlich. Der Aufrufsgraph muss damit unter Verwendung des *IntentContextSelector* erstellt werden

[12] Die Variante des Modells, die alle Koponenten einer Applikation enthält. Im Quelltext wird sie auch als *MacroModel* beszeichent

-*Model* wiederfinden. Diese Methode enthält androidspezifische Aufrufe, wie sie im Zuge dieser Arbeit synthetisiert werden.

Noch bevor die ersten Einsprungspunkte der Android-Applikation aufgerufen werden, kann optional eine Android-Umgebung generiert werden (siehe Unterabschnitt 4.5.1).

4.5.1 Erstellung der Umgebung

🖮 com.ibm.wala.dalvik.ipa.callgraph.androidModel.stubs.AndroidBoot

Bevor die ersten Einsprungspunkte zu dem Modell hinzugefügt werden wird teilweise eine Umgebung geschaffen, die den Ausführungskontext einer Applikation beinhaltet. Diese ist für die Analyse einer einzelnen Applikation zunächst nicht zwingend notwendig, wird aber interessant sobald Komponenten externer Programme gestartet werden sollen.

Derzeit umfassen die erstellten Strukturen Informationen über den *Main-Thread* der Applikation sowie die androideigenen Kontexte zu Paket und System. Diese Daten werden über die androidinterne Methode *attach* zu allen Komponenten innerhalb des *AnalysisScope* zugewiesen.

Nicht alle für die Generierung der Umgebung erforderlichen Strukturen sind auch Bestandteil der offiziellen *Stubs* (sie sind auch nicht im Android Referenz-Handbuch [6] aufgeführt). Aus diesem Grund muss zunächst eine erweiterte Version von *Stubs* erstellt werden. Hierfür wurde ein Hilfsprogramm *stubsbuilder* (siehe Unterabschnitt 5.4.1) geschaffen.

Alternativ lässt sich die Generierung der Umgebung mittels *AndroidEntryPointManager.setDoBootSequence* deaktivieren. In diesem Fall lassen sich auch die offiziellen *Stubs* verwenden.

4.5.2 Instanziierung der REUSE-Parameter

🖮 com.ibm.wala.dalvik.ipa.callgraph.impl.DexFakeRootMethod.ReuseParameters

In der Umsetzung des Modells ist vorgesehen, dass Parameter, deren Verhalten auf *REUSE* gesetzt ist, als Argument an das Modell übergeben werden. Dadurch ist es möglich den Zustand dieser Variablen in einer *Wrapper-Funktion* nach Ausführung des Modells auszulesen. Als Seiteneffekt lässt sich so weiterhin anhand des *Systemabhängigkeitsgraphen* erkennen, welche Parameter bei Erstellung auf *REUSE* gesetzt waren.

Um solche Parameter als Argument zu übergeben, müssen sie zunächst in der *DexFakeRootMethod* instanziiert werden[13], anschließend muss die Signatur des

[13] Man kann auch das MacroModell mittels *getMethodEncap* wrappen. Dann werden die Parameter mit dem *Instantiator* (siehe Unterabschnitt 5.1.8) erstellt

Modells entsprechend angepasst und schließlich eine entsprechende Zuordnung der Instanzen zu der passenden Stelle in der Signatur vorgenommen werden.

Dies wird durch die Klasse *ReuseParameters* vorgenommen. Anschließend werden die Parameter in einem *AndroidModelParameterManager* (siehe Unterabschnitt 4.2.2) zur besseren Verwaltung registriert.

4.5.3 Instanziierung der CREATE-Parameter

Die Behandlung von *CREATE*-Parametern erfolgt erst später, sobald Einsprungspunkte hinzugefügt werden. Solche Variablen werden anders als die *REUSE*-Variablen zuvor nicht durch *WALA*, sondern durch ein anderes System, dem *Instantiator* (siehe Unterabschnitt 5.1.8) aus der Toolbox, instanziiert. Dies ist nötig, da *WALAs* Instanziierungscode fest an *FakeRoot*-Methoden gekoppelt und außerhalb von diesen nicht verwendbar ist.

CREATE-Variablen wird im *SSAValueManager* ein *UniqueKey* zugeordnet, wodurch sie nicht mehr nachschlagbar sind, sobald der Schlüssel nicht mehr vorhanden ist.

4.5.4 Hinzufügen der Einsprungspunkte

Nachdem somit die Signatur des Modells komplettiert wurde kann man dieser Funktion nun einen Funktionskörper zuweisen. Dies geschieht durch das Erstellen einer *MethodSummary*. Im Falle des Modelles wird jedoch zunächst eine *VolatileMethodSummary* (siehe Unterabschnitt 5.1.5 in der Toolbox) erstellt. Diese Variante erlaubt es Instruktionen außerhalb der Reihenfolge, in der sie später ausgeführt werden, hinzuzufügen. Dies erleichtert die Erstellung von Schleifen.

Befüllt wird der Funktionskörper mit Anweisungen, welche durch eine *TypeSafeInstructionFactory* (siehe Unterabschnitt 5.1.2) erstellt werden. Diese weist eine andere Struktur bezüglich von *SSA-Variablen* als die sonst für diesen Zweck übliche *JavaInstructionFactory* auf. Weiterhin werden auch zusätzliche Prüfungen auf den übergebenen Parametern durchgeführt.

Die hinzuzufügenden Einsprungspunkte liegen an dieser Stelle bereits sortiert vor und werden somit linear abgearbeitet. Vor dem eigentlichen Einfügen eines Einsprungspunktes werden – so erforderlich – zunächst Sonderbehandlungen gemäß Unterabschnitt 4.5.5 durchgeführt.

Vor dem Aufruf des Einsprungspunktes muss eine Zuordnung von Variableninstanzen zu dessen Argumenten erfolgen. Handelt es sich bei einem Argument um einen *REUSE*-Typ, so wird die aktuelle Instanz bei dem *AndroidModelParameterManager* (siehe Unterabschnitt 4.2.2) erfragt. Im Falle von *CREATE*-Typen werden diese wie in Unterabschnitt 4.5.3 instanziiert.

Anschließend muss eine weitere *SSA-Variable* für das Auftreten eventuell auftretender Exceptions vorgesehen werden.

```
TypeSafeInstructionFactory insts;
VolatileMethodSummary body;
IMethod callee;
ParameterAccessor thisAcc = new ParameterAccessor(thisMRef, false)
SSAValueManager pm = new SSAValueManager(thisAcc);
Instantiator instantiator = new Instantiator(body, insts, pm, cha, null, null);
ParameterAccessor calleeAcc = new ParameterAccessor(callee);

List<SSAValue> params = thisAcc.connectThrough(calleeAcc, null, null,
        cha, instantiator, false, null, null);
SSAValue exception = pm.getException();
int pc = body.getNextProgramCounter();
CallSiteReference site = CallSiteReference.make(pc, callee.getRefernence(),
        (calleeAcc.hasImplicitThis())? Dispatch.VIRTUAL : Dispatch.STATIC);
SSAAbstractInvokeInstruction inst = insts.InvokeInstruction(pc, params,
        exception, site);
body.addStatement(inst);
```

Listing 13: *Beispiel eines synthetischen Aufrufs einer void-Funktion (unter Verwendung des ToolKits)*

Nun liegen alle erforderlichen Daten für den eigentlichen Aufruf vor. Es wird eine *CallSiteReference* für die Aufrufsstelle generiert, anschließend der Aufruf selbst durch Hinzufügen einer passenden *InvokeInstruction* zu dem Funktionskörper durchgeführt.

Hat der aufgerufene *Einsprungspunkt* einen Rückgabewert, so wird dieser in zuweisungskompatieble *REUSE*-Variablen über eine Φ-*Funktion* eingemischt und der *AndroidModelParameterManager* über die aktualisierte Variable informiert. Existiert keine solche Variable, so wird der Rückgabewert verworfen.

Wurden alle Einsprungspunkte hinzugefügt, so wird die Modellstruktur durch Aufruf der Funktion *finish()* komplettiert. Diese schließt alle noch offenen Sektionen.

4.5.5 Einfügen von Sonderbehandlungen

🖮 com.ibm.wala.dalvik.ipa.callgraph.impl.AbstractAndroidModel

Sonderbehandlungen werden immer dann eingefügt, wenn ein *Label* überschritten wird[14]. Dies geschieht mit dem *Label AT_FIRST* direkt zu Beginn der Synthetisierung noch bevor Einsprungspunkte hinzugefügt wurden.

Die jeweiligen Sonderbehandlungen sind in einer Klasse, die *AbstractAndroidModel* erweitert implementiert. In Abschnitt 3.4 wurden bereits einige mögliche Modell-Strukturen vorgestellt. Im Umfang dieser Arbeit wurden folgende Klassen zur

[14] Es ist auch die Definition eines *AbstractAndroidModels* möglich, bei dem vor jedem Einsprungspunkt Sonderbehandlungen eingefügt werden.

Abbildung solcher Strukturen implementiert:

SequentialAndroidModel ein Modell, das keinerlei Schleifen beinhaltet.

LoopAndroidModel mit zwei Schleifen: Einer äußeren zur Nachempfindung des Starts einzelner Komponenten und einer Inneren, welche *asynchrone Event*s enthält, die Auftreten, wenn eine Komponente im Vordergrund ist.

LoopKillAndroidModel erweitert das *LoopAndroidModel* um eine weitere Schleife, welche den Neustart einer zuvor aufgrund von Speicherknappheit abgebrochenen Ausführung berücksichtigt.

Eine Sonderbehandlung kann prinzipiell beliebigen Code zu dem Modell hinzufügen. Typischerweise werden jedoch lediglich Schleifen eingefügt. Dies sei im folgenden Abschnitt betrachtet.

Einfügen von Schleifen

Für das Einfügen von Schleifen in das Modell sind einige Vorkehrungen zu treffen: Listing 14 zeigt das Betreten einer Schleife; in Listing 15 wird der Code zum Vollenden der Schleife aufgeführt.

Zu Beginn der Schleife sind Φ-*Funktionen* einzufügen um Änderungen an Variablen, welche in der Schleife auftreten, zu verarbeiten. Zu diesem Zeitpunkt ist allerdings noch nicht bekannt, welche *SSA-Nummern* die später geänderten Variablen erhalten werden. Aus diesem Grund wird die Generierung der Φ-*Funktionen* zunächst verzögert.

Der *AndroidModelParameterManager* durch Aufruf von *scopeDown* auf den Wechsel vorbereitet. Anschließend wird eine Liste der betroffenen Typen erstellt. Der Aufruf *paramManager.getFree(phiType)* bewirkt, dass bei Anfragen für *REUSE*-Parameter je eine feste unbelegte *SSA-Nummer* zurückgegeben wird. Die *VolatileMethodSummary* wird angewiesen Platz für die spätere Aufnahme der Φ-*Funktionen* zu reservieren. Damit ist der Beginn einer Schleife vorbereitet. Nun folgende Einsprungspunkte können weiterhin wie in Unterabschnitt 4.5.4 beschrieben behandelt werden.

Das Verlassen der Schleife lässt sich wieder in zwei Abschnitte teilen: Zunächst muss der zuvor reservierte Platz nun wie in Listing 15 dargestellt mit tatsächlichen Φ-*Funktionen* gefüllt werden. Hierzu werden die reservierten Positionen zunächst zum Beschreiben freigegeben. Für alle betroffenen Typen werden nun in beiden *Scopes* die jeweils zu verwendenden Instanzen als Parameter der Φ-*Funktionen* und die mittels *getFree* erhaltene Nummer als Ziel gesetzt. Die so erstellte Funktion wird zum Funktionskörper hinzugefügt und im *AndroidModelParameterManager* als abgearbeitet markiert. Schließlich wird die Schleife mittels einer *GotoInstruction* geschlossen.

Abschließend sind nach der Schleife noch weitere Φ-*Funktionen* einzufügen. Dies wird in Listing 16 dargestellt. Durch *getAllForPhi* werden unter Berücksichtigung

```
public class LoopAndroidModel extends AbstractAndroidModel {
    // ...
    private int outerLoopPC = -1;
    Map<TypeReference, Integer> outerStartingPhis;

    protected int enterSTART_OF_LOOP (int PC) {
        this.outerLoopPC = PC;
        paramManager.scopeDown(/* isLoop = */ true);

        // Top-Half of Phi-Handling
        outerStartingPhis = new HashMap<TypeReference, SSAValue>();
        List<TypeReference> outerPhisNeeded = returnTypesBetween(START_OF_LOOP,
                AFTER_LOOP);

        for (TypeReference phiType: outerPhisNeeded) {
            if (paramManager.isSeen(phiType, /* include super scope */ false)) {
                SSAValue newInst = paramManager.getFree(phiType);
                outerStartingPhis.put(phiType, newInts);
            }
        }

        body.reserveProgramCounters (outerPhisNeeded.size());
    }
```

Listing 14: *Start einer Schleife*

des Scope alle Instanzen selektiert, die in einer Φ-*Funktionen* zu verarbeiten sind. Anschließend werden diese von der weiteren Verwendung ausgenommen und eine neue Nummer für die weitere Verwendung gesetzt.

4.5.6 Abschließende Arbeiten

Sind alle Einsprungspunkte zu dem Modell hinzugefügt wird zunächst die Funktion *finish()* des *AbstractAndroidModel* aufgerufen. Durch sie wird veranlasst, dass zunächst alle verbleibenden *Labels* übersprungen – und damit ihre Sonderbehandlungen eingefügt – werden. Abschließend wird auf ihm die Funktion *leave_AT_LAST* aufgerufen.

Da bei der Erstellung des Modells eine *VolatileMethodSummary* (siehe Unterabschnitt 5.1.5) zum Einsatz kam müssen nun noch die Instruktionen des Funktionskörpers sortiert werden und eine "herkömmliche" *MethodSummary* erstellt werden. Da der *AndroidModelParameterManager* die Verwendung von *SSA-Variablen* vor deren Definition erlaubt, muss nun geprüft werden, ob alle Variablen letztendlich rechtzeitig definiert wurden.

```
public class LoopAndroidModel extends AbstractAndroidModel {
    // ...
    protected int enterAFTER_LOOP (int PC) {
        // Insert the Phis at the beginning of the Block
        int phiPC = outerLoopPC + 1;
        boolean oldAllowReserved = body.allowReserved(true);

        for (TypeReference phiType : outerStartingPhis.keySet()) {
            SSAValue oldPhi = outerStartingPhis.get(phiType);
            SSAPhiInstruction phi = insts.PhiInstruction(phiPC, oldPhi,
                    new int[] {
                        paramManager.getSuper(phiType),
                        paramManager.getCurrent(phiType)
                    });
            phiPC++;
            body.addStatement(phi);
            paramManager.setPhi(phiType, oldPhi, phi);
        }
        body.allowReserved(oldAllowReserved);

        // Close the Loop
        body.addStatement(insts.GotoInstruction(PC, outerLoopPC));
        paramManager.scopeUp();

      // ...
```

Listing 15: *Schließen der Schleife*

4.6 Auflösen der Ziele von Intents

Die Frage der Auflösung des Starts weiterer Komponenten Androids stellt sich erst deutlich später, sobald der *Aufrufsgraph* erstellt wird. Abhängig von den Einstellungen des *CallGraphBuilders* erfolgt das Auflösen der in Anhang D aufgelisteten *Start-Funktionen* leicht unterschiedlich.

```
        // ...
        //body.addStatement(insts.GotoInstruction(PC, outerLoopPC));
        //paramManager.scopeUp();

        for (TypeReference phiType : outerStartingPhis.keySet()) {
            PC = body.getNextProgramCounter();

            List<SSAValue> all = paramManager.getAllForPhi(phiType);

            paramManager.invalidate(phiType);
            SSAValue newValue = paramManager.getFree(phiType);
            SSAPhiInstruction phi = insts.PhiInstruction(PC, newValue, all);
            body.addStatement(phi);

            paramManager.setPhi(phiType, newValue, phi);
        }
    }
}
```

Listing 16: *Fortsetzung nach der Schleife*

4.6.1 Kontextfreie Variante

🖨 com.ibm.wala.dalvik.ipa.callgraph.androidModel.stubs.Overrides
🖨 com.ibm.wala.dalvik.ipa.callgraph.propagation.cfa.IntentStarters

Sollen die später beschriebenen Klassen *IntentContextSelector* und *IntentContextInterpreter* nicht verwendet werden, so können die *Start-Funktionen* durch einen *StartComponentMethodTargetSelector* auf die im Folgenden behandelten *Wrapper-Funktionen* umgeschrieben.

Erstellt wird der *MethodTargetSelector* durch die Funktion *overrideAll()* der *Overrides*-Klasse: Zunächst wird für jede Android-Komponente ein *Mini-Modell*, in dem die auftretenden Einsprungspunkte auf die des jeweiligen Komponententyps eingeschränkt sind, generiert. Anschließend wird über alle *Start-Funktionen* (siehe Anhang D) iteriert. Für jede Funktion ist hinterlegt, welcher Komponententyp als Ziel dieser Funktion dienen kann. Für die Funktion wird nun eine Umleitung zu dem passenden *Mini-Model* angelegt.

Der so angelegte Satz von Umleitungen muss schließlich noch registriert werden. Dazu wird er zunächst optional noch mit *Eltern-Umleitungen* versehen und anschließend in den *AnalysisOptions* hinterlegt.

4.6.2 Variante mit spezialisiertem Kontext

com.ibm.wala.dalvik.ipa.callgraph.propagation.cfa.IntentContext
com.ibm.wala.dalvik.ipa.callgraph.propagation.cfa.IntentContextSelector
com.ibm.wala.dalvik.ipa.callgraph.propagation.cfa.IntentContextInterpreter
com.ibm.wala.dalvik.ipa.callgraph.propagation.cfa.IntentStarters

Bei dieser Variante wird über die Klasse *IntentContextSelector* ein auf das Auflösen von *Intents* spezialisierter Kontext erstellt, welcher bei Auftritt einer der in Anhang D definierten Funktionen (über einen *IntentContextInterpreter*) zum Einsatz kommt.

Beide Klassen werden in Unterabschnitt 5.2.4 genauer beschrieben, dennoch sei hier kurz auf sie eingegangen. Der *IntentContextSelector* dient dazu einen spezialisierten Kontext zu einem Aufruf einer *Start-Funktion* hinzuzufügen. Er wird vom *CallGraphBuilder* jeweils vor den Besuchen weiterer Funktionen angesprungen. Soll der Konstruktor eines *Intents* besucht werden, so werden dessen Parameter analysiert: Handelt es sich bei den Parametern um Konstanten[15], so kann deren Inhalt ausgelesen werden. Ausgehend von diesen Daten wird ein *WALA-internes Intent-Objekt*[16] erstellt und unter einem sogenannten *InstanceKey* hinterlegt. Ein solcher *InstanceKey* ist fest mit einer Instanz verbunden, wird das *Intent-Objekt* später verwendet, so kann es über diesen eindeutig zugeordnet werden.

Jede *Start-Funktion* nimmt als Parameter einen *Intent*. Soll eine solche Funktion besucht werden wird dem *IntentContextSelector* der *InstanceKey* dieses Parameters übergeben. Er kann genutzt werden, um das zuvor erstellte *Intent-Objekt* in einer Tabelle nachzuschlagen. Somit kann ein *IntentContext* erstellt werden, der zu dem Aufruf hinzugefügt wird.

Verarbeitet wird dieser Kontext durch den *IntentContextInterpreter*. Dieser schlägt das Ziel des Intents im *AndroidModelEntrypointManager* nach. Je nach Genauigkeit der Auflösung des Ziels wird eine nun eine *Wrapper-Funktion* gemäß Unterabschnitt 4.6.5 generiert. Die *Intermediate Representation* der *Start-Funktion* wird dann durch die des *Wrappers* ersetzt.

Das Klassenpaar *IntentContextSelector*, *IntentContextInterpreter* wird analog zu den zuvor beschriebenen kontextfreien *Overrides* in den *AnalysisOptions* hinterlegt. Wird diese kontextsensitive Variante verwendet, so ist es nicht nötig zusätzlich die kontextfreien Umleitungen zu registrieren[17].

---

[15] Als konstant sind auch Variablen anzusehen, deren Inhalt zur *Compile-Zeit* eindeutig bestimmt werden kann, sowie das *.class*-Attribut von Klassen

[16] Das *WALA-interne Intent-Objekt* unterscheidet sich in der Ausprägung von dem Androids.

[17] Man kann dennoch zusätzlich kontextfreie Overrides registrieren, sie treten allerdings nie in Kraft

4.6.3 Die Intent-Ziel-Tabelle

🖶 com.ibm.wala.dalvik.util.AndroidEntryPointManager
🖶 com.ibm.wala.dalvik.ipa.callgraph.propagation.cfa.Intent

Android benutzt für die Auflösung von *Intents* einen String *action* und ein *URI*-Objekt. Das *WALA-Modell* nutzt derzeit lediglich *action*. Für solche *actions* kann ein später beschriebener Eintrag in der *Intent-Ziel-Tabelle* vorliegen.

Befüllt wird diese Tabelle entweder durch Einlesen von *AndroidManifest.xml* (siehe 4.1) oder über die im Scan-Schritt (siehe Unterabschnitt 4.4.2) erstellte Einsprungspunktdatei (siehe Anhang E). Die Tabelle enthält somit standardmäßig nur Einträge für *Intents*, die von der Applikation selbst zur Verfügung gestellt werden. Soll die Tabelle so erweitert werden, dass sie auch Einträge für alle in der Applikation auftauchenden *Intents* enthält, so muss zunächst ein kompletter *Aufrufsgraph* für das Programm erstellt werden. Anschließend können aufgetretene *Intents* über die Funktion *AndroidEntryPointManager.getSeen()* ausgelesen werden. Die Ziele der *Intents* müssen dann manuell aufgelöst und in der Tabelle hinterlegt werden.

Einträge der Tabelle

🖶 com.ibm.wala.dalvik.util.AndroidSettingFactory

Die Einträge der Tabelle sind zunächst nicht intuitiv verständlich: Schlüssel und Wert werden jeweils aus einem der *WALA-internen Intent-Objekte* (wie sie bei dem *IntentContextSelector* zum Einsatz kommen) gebildet. Diese Überschreibungen werden solange verfolgt, bis sich ein *Intent* selbst überschreibt. Dabei ist zu beachten, dass sich die Gleichheit von *Intent-Objekten* lediglich auf die Felder *action* und *uri* des *Intents* bezieht, die Größer-Kleiner-Relationen jedoch nur die Genauigkeit der jeweiligen Ziele betrachten. Entlang einer Überschreibungskette können demnach alle Einträge gleich und dennoch echt größer zueinander sein.

Um Inkonsistenzen vorzubeugen ist der direkte Zugriff auf diese Tabelle nicht möglich. Sie kann mittels der Funktion *setOverride* des *AndroidEntryPointManager* beschrieben, mittels *getIntent* gelesen werden. Um Einträge hinzuzufügen erstellt man die *Intent-Objekte* mittels *AndroidSettingFactory*, diese kümmert sich auch um das Setzen entsprechender Konfidenz-Niveaus. Einträge können zu der Tabelle nur hinzugefügt (also nicht geändert oder gelöscht) werden. Das Hinzufügen ist nur dann erfolgreich, wenn die neue Überschreibung mehr Informationen als die vorhandenen Einträge enthält.

Ein *Intent* besitzt eine der Auflösungsarten *Intern*, *Extern*, *Standard-Aktion*, *SystemService*[18] und *Unbekannt*.

[18] *SystemServices* werden eigentlich nicht über ein Intent gestartet. In der Implementierung bot es sich allerdings an mit ihnen geschlossen mit Intents zu verfahren.

Damit ein *Intent* als *intern* aufgelöst wird, muss eine Klasse existieren, deren Namen auf den *action*-Eintrag des *Intents* lautet. Weiterhin muss ein Überschreibungseintrag in der Tabelle existieren, der von vornherein als intern erstellt wurde. Es ist eine weitere Möglichkeit mittels einer strikten Prüfung auf Internalität vorgesehen. Diese Prüfung liefert derzeit allerdings grundsätzlich ein "unbekannt"-Ergebnis.

Die Feststellung, ob ein *Intent* extern aufgelöst werden soll verläuft zunächst analog zu der Prüfung, ob ein *Intent intern* aufgelöst werden soll. Hier existiert allerdings bereits die erwähnte Prüfung auf Externalität: Sie liefert einen wahren Wert zurück, wenn sich das Ziel in einem anderen Java-Paket, als das analysierte Programm befindet.

Die Art *Standard-Aktion* ist für *action*-Werte vorgesehen, die durch das Android-System festgelegt sind (beispielsweise "Landroid/intent/action/MAIN"). Diese Art wird zugeordnet, wenn das in *action* verwendete Paket mit "android" beginnt.

Unbekannt fasst schließlich alle Intents, für die keine andere Art feststellbar war.

Abhängig von der Art des Intents wird für den *IntentContextInterpreter* eine jeweils angepasste *Wrapper-Funktion* zu einem eingeschränkten Modell erstellt.

4.6.4 Generierung eingeschränkter Modelle

Für den Start weiterer Komponenten werden je nach Informationslage eingeschränkte Modelle generiert. Sämtliche so generierten Modelle finden sich ebenfalls in der *om.ibm.wala.AndroidModelClass*, sind in dem späteren *SDG* jedoch nur sichtbar, wenn sie auch aufgerufen werden.

Die erste Art eingeschränkter Modelle seien im Folgenden als *Mini-Modell* bezeichnet. Ihr Name beginnt mit *startAll* gefolgt von der Art des jeweils zugeordneten Komponententyps (z.B. *startAllActivity*). Wie bereits aus dem Namen hervorgeht besteht die Einschränkung aus allen Klassen, welche eine Komponente gegebenen Typs implementieren und sich im *AnalysisScope* befinden. In der Regel wird ein *Mini-Modell* nicht direkt, sondern durch eine *startUnknown*-Funktion aufgerufen.

Eine weitere Art findet sich im Folgenden unter der Bezeichnung *Micro-Modell*. Diese werden herangezogen wenn die zu startende Komponente eindeutig bekannt ist, beinhalten jedoch lediglich Einsprungspunkte zu dieser. Somit wird zu einer Komponente X ein Modell mit dem Namen *startX* generiert.

Eine Letzte Art von Modellen existiert mit dem *IntentModel*. Dieses kann verwendet werden, wenn die Analyse zu Beginn nicht alle Komponenten berücksichtigen soll. Mit ihm ist es möglich den Start der Applikation durch ein *Intent*, beispielsweise dem *Main-Intent*, nachzuempfinden. Es ruft Einsprungspunkte von *Application*, *ContentProvider* und der Komponente, die dem gewählten *Intent* zugeordnet ist, auf.

Durch die *startUnknown*- und *startExternal*-Funktionen werden keine Modelle im eigentlichen Sinn generiert: *startUnknown* ruft zunächst das passende *Mini-Modell*, anschließend eine *startExternal*-Funktion auf.

Die *startExternal*-Funktionen finden für nicht intern auflösbare Ziele Verwendung.

Sie greifen lesend auf die übergebenen Parameter zu und generieren daraus einen Rückgabewert. Da keine Informationen über das eigentliche Ziel bekannt sind sollten sie in der späteren Analyse als unsicher eingestuft werden.

4.6.5 Generierung von Wrapper-Funktionen für das Modell

Diese *Wrapper-Funktionen* dienen dazu ein *Android-Modell* über eine andere *Signatur*, als der für das Modell typischen aufzurufen. Sie beinhalten darüber hinaus allerdings noch zusätzlichen Code für die Festlegung androidinterner Aufrufgegebenheiten und rufen falls nötig *CallBack*-Funktionen auf.

Generieren lassen sich *Wrapper-Funktionen* mittels der Methode *getMethodAs* des jeweiligen *Android-Modells*. Im Folgenden seien die Aktionen beschrieben, die zusätzlich zu dem Aufruf des Modells stattfinden.

Vor Aufruf des Modells

🖮 com.ibm.wala.dalvik.ipa.callgraph.androidModel.stubs.AndroidStartComponentTool

Android verwaltet einen *internen Kontext* (nicht zu verwechseln mit dem *Analyse-Kontext* in *WALA*). Er enthält unter anderem Informationen, von welcher *Komponente* aus in welchem *Thread* und mit welchen *Berechtigungen*[19] ein Aufruf erfolgt.

Zunächst muss der Typ des vorliegenden Kontexts festgestellt werden, da diese unterschiedlich verarbeitet werden. Erkannt werden derzeit *ACTIVITY*, *CONTEXT_IMPL* und *CONTEXT_BRIDGE* (und der *ContextWrapper*, der allerdings gleich weiter aufgelöst wird). Der entsprechende Kontext wird anschließend in sehr groben Zügen ausgelesen.

Anschließend wird aus dem ausgelesenen Kontext ein sogenannter *IBinder* ausgelesen (es sei denn die der *Start-Funktion* zugeordneten *StarterFlags* enthalten entweder *CONTEXT_FREE*, sodass kein *IBinder* ausgelesen wird, oder das *Flag QUENCH_PERMISSIONS*, das besagt, dass der *IBinder* nicht aus dem Aufruferkontext, sondern aus einem *IntentSender*-Objekt, welches an die *Start-Funktion* – und somit der *Wrapper-Funktion* – übergeben wird, bezogen wird). Der so ausgelesene *IBinder* wird nun der zu startenden Komponente zugewiesen. Dies geschieht, falls es sich bei dem Ziel um eine *Activity* handelt durch Setzen des Feldes *mToken*. Weitere Implementierungen existieren hier noch nicht.

Nun müssen noch die Parameter zu dem Modell generiert werden. Nimmt die *Start-Funktion* kein Argument des Typs *android.os.Bundle* so lautet dieses – Android-Typisch – immer auf *null*. Weitere Parameter werden durch die *connectThrough*-Funktion der Klasse *ParameterAccessor* (siehe Unterabschnitt 5.1.3) entweder von den Parametern der *Wrapper-Funktion* übernommen oder neu instanziiert.

Nun wird das Modell aufgerufen.

[19] ein Aufruf kann über ein spezielles Konstrukt sogar mit fremden Berechtigungen erfolgen

Nach Aufruf des Modells

Ist das *Flag CALL_ON_ACTIVITY_RESULT* der *Start-Funktion* gesetzt, so muss die zugehörige *CallBack*-Funktion der aufrufenden Android-Komponente, sobald die gestartete Komponente nicht mehr sichtbar ist, mit den gesetzten Rückgabewerten der Komponente aufgerufen werden.

Der Einfachheit halber wurde dieser Aufruf nach die Ausführung des Modells verschoben. Dies sollte allerdings kaum negativen Auswirkungen auf das Analyseergebnis haben[20].

Die Rückgabewerte der gestarteten *Activity* werden aus den entsprechenden Feldern ausgelesen und die *CallBack*-Funktion aufgerufen.

[20] Das Ergebnis kann unter Umständen ungenauer werden, jedoch keine Datenpfade verlieren

KAPITEL 5

Weitere Anpassungen der Software

Im Zuge der Arbeit mussten einige Anpassungen und Erweiterungen der Software vorgenommen werden. Zunächst wird auf Änderungen, die an *WALA* vorgenommen wurden, eingegangen; Die Beschreibungen der Änderungen an *Joanas JoDroid* (siehe Abschnitt 5.3) und *Joana* selbst folgen. Abschließend werden einige neu erstellte Hilfsprogramme genannt.

5.1 Toolbox für Synthetische Methoden in WALA

com.ibm.wala.util.ssa.*

Methoden, welche zur Analysezeit angelegt werden, werden in *WALA synthetischer Methoden*[1] genannt. Das direkte Erstellen solcher *synthetischer Methoden* ist dadurch, dass keinerlei Typprüfungen stattfinden und durch die reine Verwendung von Zahlen für *SSA-Variablen*, sehr fehleranfällig. Deshalb entstand im Rahmen dieser Arbeit eine Toolbox, die die Erstellung solcher Methoden vereinfachen soll und Fehler durch zusätzliche Prüfungen besser erkennt.

In Listing 17 ist der Aufruf einer Void-Funktion ohne ToolBox dargestellt. Dieser Aufruf funktioniert lediglich aus *FakeRoot*-Methoden heraus; in anderen Methoden ist die benötigte Funktion *createInstance* nicht verfügbar. Listing 18 zeigt den selben Aufruf unter Verwendung der Toolbox. Er ist in dieser Form aus jeder synthetischen Methode heraus möglich.

Man erkennt bereits in diesem einfachen Beispiel die Vereinfachung des Aufrufes, weiterhin dass keine Parameter mehr mit Nummern benannt sind (*nextLocal*). Sollen in der synthetisierten Methode Schleifen aufgenommen werden oder sind aus anderen Gründen Φ-Funktionen erforderlich oder sollen die benötigten Parameter

[1] Solche *WALA-synthetische* Methoden sind nicht mit *synthetischen Methoden* Javas zu verwechseln. Letztere werden durch den Java-Compiler bei der Implementierung parametrisierter Interfaces erzeugt. Sie sind auch als *Bridge-Method* bekannt.

anderweitig übernommen werden, so fällt der Unterschied noch drastischer aus. Der Programmcode der herkömmlichen Variante könnte schnell mehrere Seiten lang werden und birgt die Gefahr einer Verwechslung der Nummern von SSA-Variablen, die von WALA ohne Warnung übernommen würden.

Die einzelnen Bestandteile dieser Toolbox seien im Folgenden näher beschreiben.

```
void callMethod(MethodReference callee) {
// MethodSummary body; // Expected to be a field in the containing class
 JavaInstructionFactory insts = new JavaInstructionFactory();

// Create variable instances for the parameters
int params[] = new int[callee.getNumberOfParameters()];
for (int i=0; i < callee.getNumberOfParameters(); ++i) {
    for (TypeReference t: callee.getParameterTypes(i)) {
        // getParameterTypes usually has a single entry
        if (t.isReferenceType()) {
            params[i] = createInstance(t, body);    // createInstance only available in
                // FakeRoot-Methods
        } else {
            params[i] = nextLocal++;
        }
    }
}
 final int exception = nextLocal++;

boolean isStatic = cha.resolveMethod(callee).isStatic();

 if (! istStatic ) {
    int newParams[] = new int[params.length + 1];
    newParams[0] = createInstance(callee.getDeclaringClass(), body);
    System.arraycopy(params, 0, newParams, 1, params.length);
    params = newParams;
}

// Insert the call
 final int programCounter = body.getNextProgramCounter();
 final CallSiteReference site = CallSiteReference.make(pc, callee,
        (isStatic)? Dispatch.VIRTUAL : Dispatch.STATIC);
 final SSAAbstractInvokeInstruction inst;
 inst = insts.InvokeInstruction(programCounter, params, exception, site);
body.addStatement(inst);
}
```

Listing 17: *Aufruf einer void-Funktion aus einer FakeRoot-Methode ohne Toolbox*

```
void callMethod(MethodReference callee) {
// VolatileMethodSummary body; expected to exist in declaring class
TypeSafeInstructionFactory insts = new TypeSafeInstructionFactory(cha);
ParameterAccessor thisAcc = new ParameterAccessor(thisMRef, false); // false indicates
    // that we don't have an implicit this-pointer.
SSAValueManager pm = new SSAValueManager(thisAcc);
Instantiator instantiator = new Instantiator(body, insts, pm, cha, mRef, null);
    // null tells the Instantiator to use the default exclusions
ParameterAccessor calleeAcc = new ParameterAccessor(callee, cha);

List<SSAValue> params = thisAcc.connectThrough(calleeAcc, null, null,
        cha, instantiator, false, null, null); // The first and second null tell the
        // ParameterAccessor that no newly creates instances should override the
        // creation of instances. false, null tells the Instantiator to create
        // unmanaged instances. (i.e. they can't be looked up in the pm by a key)
        // The last null tells the Instantiator that if during a possible recursion
        // while creating an instance no existing variable should be used.
SSAValue exception = pm.getException();
int pc = body.getNextProgramCounter();
CallSiteReference site = CallSiteReference.make(pc, callee,
        (calleeAcc.hasImplicitThis())? Dispatch.VIRTUAL : Dispatch.STATIC);
SSAAbstractInvokeInstruction inst = insts.InvokeInstruction(pc, params,
        exception, site);
body.addStatement(inst);
}
```

Listing 18: *Beispiel eines synthetischen Aufrufs einer void-Funktion (unter Verwendung des ToolKits)*

5.1.1 Kapselung von SSA-Variablen

com.ibm.wala.util.ssa.SSAValue

Die Klasse *SSAValue* kombiniert zunächst lediglich die der Variable zugewiesene Zahl mit einem Datentyp der Variable. Für eine solche Variable ist ebenfalls hinterlegt, ob ihr schon ein Wert zugewiesen wurde. Diese Eigenschaft wird von der *TypeSafeInstructionFactory* (siehe Unterabschnitt 5.1.2) ausgewertet. Weiterhin wird der Variable eine Methode zugeordnet, um ihren Gültigkeitsbereich zumindest etwas einzuschränken.

Optional kann einer *SSA-Variable* auch ein Name zugewiesen werden, welcher in die synthetische Methode übernommen wird[2]. Hierzu muss allerdings eine *Summari-*

[2] Variablen erscheinen somit beispielsweise in einem mittels *JoDroid* erstellten *SDG* mit Namen

zedMethodWithNames (siehe Unterabschnitt 5.1.6) statt einer *SummarizedMethod* erstellt werden.

Um *SSA-Variablen* zueinander zuzuordnen oder um sie im *SSAValueManager* (siehe Unterabschnitt 5.1.4) wieder aufzufinden besitzen durch diese Klasse dargestellte Variablen weiterhin einen *VariableKey*. Dieser wird durch eine der folgenden Klassen implementiert:

NamedKey
Um auf eine Variable mit einem solchen Schlüssel wieder aufzufinden benötigt man ihren Namen, sowie Typ.

TypeKey
Dieser Schlüssel findet bei Variablen Verwendung, denen kein Name zugewiesen werden soll. Er hat den Nachteil, dass in einer Methode keine zwei unabhängigen Variablen selben Typs existieren können.

WeaklyNamedKey
Hierbei handelt es sich um einen *NamedKey*. Im Unterschied zu diesem lässt sich auf eine Variable mit einem *WeaklyNamedKey* jedoch auch durch einen *TypeKey* zugreifen.

UniqueKey
Auch ein *UniqueKey* hat einen Variablen-Typ zugeordnet. Im Unterschied zu den vorherigen Schlüssel lassen sich jedoch keine zwei gleichen *UniqueKeys* erzeugen.

Die durch diese Klasse zur Verfügung gestellte Darstellung einer *SSA-Variable* findet in allen folgenden Klassen der Toolbox Verwendung.

5.1.2 Die typsichere Instruction-Factory

com.ibm.wala.util.ssa.TypeSafeInstructionFactory

Diese Klasse führt Prüfungen der Zuweisbarkeit von Variablen aus und stellt sicher, dass Referenzen, wie Methoden oder Felder, auch tatsächlich existieren.

In den jeweiligen Methodensignaturen sind *SSA-Variablen* durch *SSAValues* (siehe Unterabschnitt 5.1.1) dargestellt, *Arrays* sind durch Listen ersetzt. Außerdem sind Typ-Parameter, welche bereits aus den *SSAValues* hervorgehen, komplett entfernt.

SSAValues denen durch die erfragte Instruktion ein Wert zugewiesen wird, werden als belegt markiert. Dadurch kann sichergestellt werden, dass jede vorkommende *SSA-Variable* genau eine Zuweisung hat.

Sind alle Prüfungen erfolgreich, so wird das Erstellen der Instruktion an eine *JavaInstructionFactory* delegiert.

5.1.3 Zugriff auf Parameter synthetischer Funktionen

🖮 com.ibm.wala.util.ssa.ParameterAccessor

Möchte man auf die Parameter einer Funktion zugreifen, so ergibt sich in *WALA* das Problem, dass *IMethods* und *MethodReferences* ein inkonsistentes Nummerierungsschema haben. Diese Problematik wird durch die Klasse *ParameterAccessor* angegangen.

Diese Klasse lässt sich aus *IMethods* und *MethodReferences* erstellen. Auch bei ihr kann man über die Position auf einen Parameter zugreifen. Dabei wird jedoch ein weiteres Nummerierungsschema eingesetzt.

Die eigentliche Idee hinter der Klasse ist jedoch, die Parameter einer Funktion ohne jegliche Verwendung von Nummern referenzieren zu können. In der aktuellen Implementierung geht dies leider nur über den Variablen-Typ. Hierzu existieren die Funktionen *firstExtends* und *allExtend*, welche einen Parameter anhand der Zuweisbarkeit selektieren. Weiterhin existiert die Methode *firstOf*, bei der der Typ exakt gleich sein muss.

Die Hilfsfunktion *connectThrough* dieser Klasse erleichtert das Schreiben von *Wrapper-Funktionen* indem sie die einzelnen Parameter automatisch an eine innere Funktion weiterleitet. Sollte es dabei von Nöten sein eine neue Instanz zu einem Typ zu erzeugen, so kann man dieser Methode einen *IInstantiator* (siehe Unterabschnitt 5.1.8) übergeben, der den dafür nötigen Code generiert.

5.1.4 Verwaltung von SSA-Variablen

🖮 com.ibm.wala.util.ssa.SSAValueManager

Wenn eine *synthetische Funktion* mehrere *Grundblöcke* (siehe Unterabschnitt 2.1.2) enthält, müssen in diese in der Regel mit Φ-Funktionen versehen werden. Die jeweils zu verwendende Version einer Variable herauszufinden kann dabei schnell unübersichtlich werden. Die Klasse *SSAValueManager* dient in diesem Fall dazu den Umgang mit *SSA-Variablen* zu erleichtern. Hierzu wird jeder Variable einer von acht Status zugeordnet, welcher Aussage darüber erlaubt, in wie weit die Variable zugewiesen wurden oder ob der Zugriff auf sie erlaubt ist.

Variablen können im *SSAValueManager* anhand ihres *VariableKeys* (siehe Unterabschnitt 5.1.1) nachgeschlagen werden. Es wird dann die jeweils aktuellste in einer *Umgebung* gültige Variante der *SSA-Variable* zurückgegeben. Weiterhin erhält man Vorschläge, welche Variablen an *Grundblockgrenzen* durch eine Φ-*Funktion* behandelt werden sollen.

```
// TypeSafeInstructionFactory insts; expected to exist
// VolatileMethodSummary body; expected to exist
// SSAValueManager pm; expected to exist
// Instantiator instantiator; expected to exist

TypeReference T = TypeReference.JavaLangString;
NamedKey key = new NamedKey(T, "myName");

SSAValue myName = instantiator.createInstance(T, true, key, null);   // myName_0
// start a loop
    pm.scopeDown(true);                          // true indicates it's a loop
    SSAValue myName_phi = pm.getFree(T, key); // There will be a Phi myName_1
    int pc = body.getNextProgramCounter();
    body.reserveProgramCounters(1);              // Reserve for the phi

    // pm.getCurrent(key) points to myName_1
    // pm.getSuper(key) points to myName_0

    myName = instantiator.createInstance(T, true, key, null); // myName_2
    myName = instantiator.createInstance(T, true, key, null); // myName_3

    // pm.getCurrent(key) points to myName_3

    body.allowReserved(true);
    SSAPhiInstruction phi = insts.PhiInstruction(pc, myName_phi, pm.getAllForPhi(key));
    // generates myName_1 = Phi(myName_0, myName_2, myName_3)
    body.addStatement(phi);
    pm.setPhi(myName_phi, phi);
    pm.scopeUp();
// end the loop
```

Listing 19: *Beispiel für die Verwendung des SSAValueManager*

5.1.5 Reihenfolgeänderung von Instruktionen

🖮 com.ibm.wala.dalvik.ipa.summaries.VolatileMethodSummary

In der üblichen *MethodSummary* kann man Funktionen lediglich in der Reihenfolge ihrer späteren Ausführung hinzufügen. Unter bestimmten Voraussetzungen möchte man von diesem Verfahren jedoch eventuell abweichen: Dies ist beispielsweise sinnvoll, wenn eine Methode Schleifen enthalten soll. Aus diesem Grund wurde die Klasse *VolatileMethodSummary* entwickelt, in der dies möglich ist.

Neben der generellen Möglichkeit Instruktionen verspätet einzufügen lassen sich auch Bereiche reservieren. Ansonsten existieren alle Funktionen der herkömmlichen *MethodSummary*. Da jedoch die Instruktionen vor der Weiterverarbeitung zunächst sortiert werden müssen, wurde davon Abstand genommen die Klasse abzuleiten; mittels der Funktion *getMethodSummary* kann eine herkömmliche *MethodSummary* aus einer *VolatileMethodSummary* erstellt werden. Dabei werden die Instruktionen anhand ihres Feldes *iindex* sortiert.

Weitere Vorkehrungen mussten getroffen werden um zu verhindern, dass bereits bestehende Instruktionen überschrieben werden. Mittels der Funktion *overwriteStatement* ist dies dennoch möglich. Der Schreibzugriff auf zuvor reservierte Bereiche lässt sich mittels der Funktion *allowReserved* steuern.

5.1.6 Variablennamen in Synthetischen Methoden

com.ibm.wala.dalvik.ipa.summaries.SummarizedMethodWithNames

Damit Variablennamen in *synthetische Methoden* übernommen werden, muss die Erstellung der *Intermediate Representation* leicht erweitert werden. Dies geschieht durch die Klasse *SummarizedMethodWithNames*. Ansonsten verhält diese sich wie die herkömmliche *SummarizedMethod*.

5.1.7 Prüfung der Zuweisbarkeit von Primitiven

com.ibm.wala.util.PrimitiveAssignability

Die von der *ClassHierarchy* zur Verfügung gestellte Zuweisbarkeitsprüfung arbeitet lediglich auf Referenztypen. Durch die Klasse *PrimitiveAssignability* wird diese Funktionalität jedoch auch für primitive Typen zur Verfügung gestellt.

5.1.8 Instanziierung von Variablen

com.ibm.wala.util.ssa.IInstantiator
com.ibm.wala.dalvik.ipa.callgraph.androidModel.parameters.Instantiator

Die Klasse *Instantiator* erzeugt in Synthetischen Methoden den nötigen Code um eine Instanz beliebigen Typs zu erzeugen. Das genaue Verhalten ist dabei abhängig von dem gegebenen Typ. Für ein *Primitiv* wird lediglich eine neue *SSA-Nummer* generiert und im *SSAValueManager* (siehe Unterabschnitt 5.1.4) hinterlegt. Bei *regulären Klassen* ergibt sich zunächst die Frage, welcher Konstruktor zur Verwendung kommen soll. Dazu wird jedem Konstruktor eine Punktzahl zugewiesen und anschließend jener verwendet, der am besten geeignet scheint. Die Punktzahl hängt dabei von der Anzahl der Parameter und deren jeweiligem Typ ab. Tabelle 5.1

Abfrage	Punktzahländerung
Konstruktor ist *private*	-10
Konstruktor ist *protected*	-1
Für jeden primitiven Parameter	-1
Für jedes Array als Parameter	-30
Für jeden Klassen-Typ als Parameter	-100
Konstruktor will gleiche Klasse als Parameter	-1000
Für jeden sonstigen Referenz-Typ als Parameter	-7
Für einen unbekannten Typ[2]	-800

Tabelle 5.1: *Punktzahländerung bei der Selektion des Konstruktors*

zeigt dazu die fest codierten Punktzahländerungen. Sie wurden ohne spezifische Analyse "nach Gefühl" festgelegt.

Für die jeweiligen Parameter werden rekursiv neue Instanzen erstellt. Dabei wird regelmäßig eine Menge gesehener Typen aktualisiert um Endlosrekursion zu vermeiden. Die während dieser Rekursion erstellten Instanzen sind standardmäßig mit einem *UniqueKey* (siehe Unterabschnitt 5.1.1) versehen und befinden sich nicht im Management des *SSAValueManager* (siehe Unterabschnitt 5.1.4). Dadurch können sie lediglich während der Erstellung des ursprünglichen Typs verwendet werden[3].

Im Falle der Instanziierung von *Array-Klassen* wird eine ein-elementiege Version erstellt, welche eine Instanz des nächst-inneren Typs enthält. *Interfaces* und *Abstrakte Klassen* erfordern zunächst die Suche aller implementierenden Klassen (bzw. aller ableitenden im Falle von Abstrakten Klassen). Diese werden zunächst lediglich rekursiv zu einer Menge hinzugefügt[4]. Erst dann wird für jedes Element dieser Menge eine Instanz erzeugt. Abschließend werden die einzelnen Instanzen mittels einer *Phi*-Funktion zusammengefasst.

5.2 Weitere Anpassungen an WALA

Für die Umsetzung des Modells mussten weitere Anpassungen an *WALA* vorgenommen werden, die außerhalb der zuvor erwähnten Toolbox liegen. Auf diese sei im Folgenden eingegangen.

[2] Dieser Fall sollte nie eintreten.

[3] Über ihre numerische SSA-Darstellung wäre das nach wie vor möglich.

[4] Würde direkt eine Instanz erstellt, so bestünde die Gefahr einer Endlosrekursion durch zyklische Abhängigkeiten

5.2.1 GoTo-Befehl für Synthetische Methoden

🖮 com.ibm.wala.ssa.SSAGotoInstruction
🖮 com.ibm.wala.cfg.InducedCFG

Da für die Erzeugung des Modells des *Android-Lebenszyklus* Schleifen benötigt werden, musste der Satz an Instruktionen, welche von *WALA* für *synthetische Methoden* zur Verfügung gestellt werden, um die *GoTo-Instruktion* erweitert werden.

Ein *Stub* dieser Funktion lag bereits vor. Da die *JavaInstructionFactory* in ihrer Originalform keinen Befehlszähler mitführt, schien die Implementierung wohl zu aufwändig. Die angepasste Variante *WALAs*, welche bei *Joana* Verwendung findet, besitzt diesen Index jedoch.

Man kann auf diese Funktion beispielsweise über die *JavaInstructionFactory* zugreifen. Die *GotoInstruction* setzt automatisch eine *Grundblockgrenze* (siehe Unterabschnitt 2.1.2) bei dem Sprungziel (und hinter die *GotoInstruction* selbst). Den Sprung setzt sie durch eine zusätzliche Kante im Kontrollflussgraphen (siehe Unterabschnitt 2.1.3) um. Als Sprungziel kann keine Φ-*Funktion* gewählt werden, da diese nicht Teil *WALAs Intermediate Reperesentaion* sind.

5.2.2 Einlesen Androids Manifest-Datei

🖮 com.ibm.wala.dalvik.util.AndroidManifestXMLReader

Das *Manifest* (siehe Unterabschnitt 2.3.8) der jeweils analysierten Applikationen wird in Unterabschnitt 4.6.2 zur Auflösung der Ziele von *Intents* benötigt. Die Datei liegt in einem komprimierten XML-Format vor. Leider kann durch diese Klasse die Datei aufgrund von Bibliothekskonflikten nicht entpackt werden. Stattdessen stellt sie eine Erweiterung des *SaxParsers* dar. In internen Strukturen ist die Dokumentstruktur des *Manifest* abgebildet. Eingelesene Dateien werden direkt auf die Einstellungen der Modellgenerierung angewandt.

5.2.3 AndroidEntryPoint

🖮 com.ibm.wala.dalvik.ipa.callgraph.impl.AndroidEntryPoint
🖮 com.ibm.wala.dalvik.util.AndroidEntryPointLocator

Die Klasse *AndroidEntryPoint* erweitert den *DexEntryPoint* um Positionierungsangaben gemäß Abschnitt 3.3. *DexEntryPoint* selbst ist ein *DefaultEntrypoint* *WALAs*.

Einsprungspunkte können mittels des *AndroidEntryPointLocator* aus der *Klassenhierarchie* ausgelesen werden. Dies ist in Abschnitt 4.1 näher beschrieben.

5.2.4 Aufrufkontext für Start-Funktionen

🖶 com.ibm.wala.dalvik.ipa.callgraph.propagation.cfa.IntentContext
🖶 com.ibm.wala.dalvik.ipa.callgraph.propagation.cfa.IntentContextSelector
🖶 com.ibm.wala.dalvik.ipa.callgraph.propagation.cfa.IntentContextInterpreter

Um die Ziele der Start-Funktionen (siehe Anhang D) genauer auflösen zu können ist es von Nöten genauere Informationen über den *Intent*, welche diese Funktionen als Parameter aufweisen, zu haben.

Zu diesem Zweck wurde zunächst der *IntentContextSelector* erstellt. Dieser tritt zunächst in Kraft, wenn der Konstruktor eines *Intents* aufgerufen wird. In diesem Fall werden die Parameter des Aufrufes ausgelesen, insbesondere der Wert von *action*. Wird diesem eine Konstante zugewiesen, so kann deren Inhalt bestimmt werden[5]. Die so ermittelten Informationen werden – zugeordnet zu dem *InstanceKey* des *Intent-Objekte*s des analysierten Programms – für die spätere Verwendung in *WALA* hinterlegt.

Sobald bei der Erstellung des *CallGraph* nun eine Start-Funktion besucht werden soll[6], tritt der *IntentContextSelector* wieder in Aktion: Zunächst wird optional ein *ContextSelector*, welcher als *parent* angegeben ist, nach dem Kontext gefragt. Dieser wird anschließend um die zuvor gespeicherten Daten erweitert (mittels eines speziellen *ContextKeys*). Bei dieser Erweiterung werden keine Einträge des ursprünglichen Kontext überschrieben.

Die auf diese Weise zu dem Aufruf hinzugefügten Informationen werden später durch den *IntentContextInterpreter* ausgewertet: Sind diese spezifisch genug um die Zielklasse eindeutig zu identifizieren und wenn das Ziel weiterhin im *AnalysisScope* liegt, so wird ein neues Lebenszyklus- Modell generiert, welches auf Funktionen der Zielklasse beschränkt ist. Anschließend wird die *Intermediate Representation* der Start-Funktion durch die einer *Wrapper-Funktion* ersetzt. Diese *Wrapper-Funktion* wird unter anderem das vorgenannte Modell aufrufen. Sie findet in Unterabschnitt 4.6.5 nähere Betrachtung.

Wird das Ziel eines *Intents* jedoch eindeutig als einer externen Applikation zugehörig erkannt, so wird eine andere *Wrapper-Funktion* generiert, welche eine der in Unterabschnitt 3.7.1 spezifizierten Funktionen für externe Ziele aufruft. Ist das Ziel nicht eindeutig erkennbar, so wird wie im kontextfreien Fall (siehe Unterabschnitt 4.6.1) weiter verfahren.

[5]Für variable Argumente lässt sich der Inhalt nicht so leicht bestimmen, es wird jedoch davon ausgegangen, dass in der Praxis hier meist Konstanten auftreten

[6]Bei Besuch der *CallSite* in der aufrufenden Funktion

5.3 Anpassungen an Joana

Auch an *Joana* selbst wurden einige kleinere Anpassungen vorgenommen.

5.3.1 Aufbereitung der Fortschrittsinformationen

🖮 edu.kit.joana.wala.jodroid.CliProgressMonitor

Sowohl *Joana* als auch *WALA* haben für lang laufende Aufgaben einen Parameter für einen *ProgressMonitor*. Dieser ist dafür vorgesehen den Prozess abzubrechen, sobald kein weiterer Fortschritt mehr festzustellen ist. In einigen Dateien mussten Anpassungen vorgenommen werden, sodass ein tatsächlicher Fortschrittswert übermittelt wird. Dieser Wert wurde anschließend in eine Ausgabe auf der Konsole umgesetzt. Diese optische Ausgabe kann mit einer Delegierung zum bisherigen Monitor versehen werden um dessen Funktionalität beizubehalten.

5.3.2 Umbenennung anonymer Variablen

🖮 edu.kit.joana.wala.core.PDGNodeCreationVisitor

Zur besseren Lesbarkeit im *Graphviewer* (siehe Abbildung 2.5.1) wurde die Benennung anonymer Variablen dahingehend angepasst, dass sie im Fall von Referenztypen den Klassennamen enthält. Dazu muss in einem späteren Schritt der Analyse die *Typinferenz* neu berechnet werden[7], was eine leicht erhöhte Laufzeit zur Folge hat.

Dies dient ausschließlich dem Debuggen und hat keinerlei Auswirkung auf die Analyse. Man kann dieses Verhalten über die *Joana-Konfiguration* (de-)aktivieren, indem man den Wert *showTypeNameInValue* der *SDGBuilderConfig* entsprechend setzt.

5.4 Hilfsprogramme

In diesem Abschnitt werden einige Hilfsprogramme betrachtet, die während der Umsetzung des Modells geschrieben wurden. Sie sind für die Modellierung nicht zwingend erforderlich, können aber in bestimmten Fällen nützlich sein.

[7] Da die ursprüngliche Typinferenz zu diesem Zeitpunkt bereits verworfen wurde

5.4.1 Generierung erweiterter Stubs

wala/joana.wala.jodroid/stubsBuilder

Bei *stubsBuilder* handelt es sich um eine *Python-Applikation* zur Erweiterung der offiziellen *Stubs*, welche Teil des *Android-SDK* sind. Für die Erweiterung ist weiterhin der Quelltext Androids, welcher ebenfalls über das *SDK* installiert werden kann, erforderlich.

Konfiguriert wird der *stubsBuilder* über eine Datei *stubsBuilder.ini*. In dieser Datei wird beschrieben, welcher Quelltext zu den *Stubs* hinzugefügt werden soll und welche Funktionen bei der Generierung auszusparen sind. Hierüber lassen sich auch komplett neue Funktionen zu den *Stubs* hinzufügen. Bei der Übernahme von Klassen aus dem Original-Sourcecode lassen sich durch *stubsBuilder* automatisiert Textersetzungen durchführen. Da auf interne Strukturen zugegriffen wird, wird man für jede Version Androids zunächst die *stubsBuilder.ini* anpassen müssen. Eine Variante für *API-Level 18* liegt bei.

5.4.2 Auffinden nicht aufgelöster Aufrufe

wala/joana.wala.jodroid/unresolvedCalls.pl

Dieses Hilfsprogramm durchsucht einen mittels *JoDroid* erstellten *SDG* (eine *.pdg*-Datei) nach Aufrufstellen, von denen keine *CL*-Kante ausgeht, die somit also bei der Erstellung des *Systemabhängigkeitsgraphen* nicht aufgelöst werden konnten.

5.4.3 Konvertierung des Formates SuSis zu Joana

edu.kit.joana.SuSi2Joana

Das Programm *SuSi* [34] (kurz für "*Sources and Sinks*") ist ein Werkzeug für die automatische Annotation von *Datenquellen* und *-senken* einer Android-Applikation für eine *IFC-Analyse*. *SuSi* erkennt zu annotierende Funktionen ausgehend von einem Satz fest hinterlegter Signaturen. Von dort aus werden durch einen Algorithmus, der auf überwachtem maschinellen Lernen beruht, weitere Methoden annotiert. Neben der Annotation als Quelle oder Senke werden gefundene Funktionen mit einer *Kategorie* versehen.

Von der Möglichkeit die Ausgabe von *SuSi* anzupassen wurde nicht Gebrauch gemacht. Stattdessen liest *SuSi2Joana* die für *FlowDroid* [15] generierten Annotationen. Diese werden anschließend mit einem *Systemabhängigkeitsgraph* in *Joanas .pdg-Format*, wie er mittels *JoDroid* geriert werden kann, verglichen. Als Ausgabe

von *SuSi2Joana* erhält man eine *ifcscript-Datei*, also eine Datei, die Anweisungen wie sie von Joanas *IFC-Console* verstanden werden, enthält.

Abhängig von den Einstellungen von *SuSi2Joana* wird ein binärer Verband gewählt oder ein mehrwertiger Verband basierend auf SuSis Kategorien verwendet. Im Falle des mehrwertigen Verbandes müssen die einzelnen Elemente des Verbandes (durch Editieren dessen Spezifikationsdatei) manuell in eine Relation zueinander gesetzt werden.

KAPITEL 6

Analyse

In diesem Kapitel sollen nun einige Android-Applikationen mit *JoDroid*, und damit unter Verwendung des im Umfang dieser Arbeit entstandenen Modells, analysiert werden.

Zunächst stellt sich die Frage, welche Programme für diesen Zweck herangezogen werden sollen; ihre Schwachstellen sollten vor der Analyse bekannt sein. Während der Entwicklung wurde ein solches Programm – *AllComponentsTest* (siehe Unterabschnitt 6.4.2) – geschrieben. Weitere geeignete Software findet sich in *DroidBench* (siehe Unterabschnitt 6.4.1) und *InsecureBank* (siehe Unterabschnitt 6.4.3). Die angesprochenen Applikationen seien zu Anfang der jeweiligen Analysen genauer betrachtet.

Vor den eigentlichen Analysen wird im Folgenden zunächst auf die verwendeten Grund-Einstellungen eingegangen.

6.1 Analyseeinstellungen

Einstellungen können an mehreren Stellen vorgenommen werden. Namentlich sind das die Einstellungen *WALAs*, Einstellungen zu der *Modellierung Androids*, weitere für die Generierung des *Systemabhängigkeitsgraph* durch *Joana* und schließlich die Konfiguration der *IFC-Analyse* in Joanas *IFC-Console* und die damit verbundene Wahl der *Quellen* und *Senken*.

In den folgenden Abschnitten werden die jeweils vorgenommenen Einstellungen an den einzelnen Komponenten beschrieben.

6.1.1 Einstellungen WALAs

Die Einstellungen an *WALA* teilen sich wiederum in zwei Gebiete auf: Zunächst wird der Umfang der zu analysierenden Eingabedaten festgelegt; daraufhin können

Einstellungen für die Analyse selbst vorgenommen werden. Letztere überschneiden sich mit den von *Joana* vorgegebenen Einstellungen.

Setzen des Umfangs der Eingabedaten

🖶 com.ibm.wala.ipa.callgraph.AnalysisScope
🖶 com.ibm.wala.dalvik.util.AndroidAnalysisScope

Welche Daten analysiert werden sollen und wo diese auffindbar sind wird in *WALA* in einem sogenannten *AnalysisScope* festgelegt. Für die Erstellung eines *AnalysisScope* existieren mehrere Hilfsklassen. Im Umfang der Analyse findet hierfür *AndroidAnalysisScope* Verwendung.

Zunächst werden die zu verwendenden *Stubs* (siehe Unterabschnitt 2.4.2) festgelegt. In der Analyse kommt hier eine erweiterte Form der im *Android-SDK* mitgelieferten *Stubs* in *API-Level 18* zum Einsatz. Die erweiterte Fassung wurde mittels des Hilfsprogramms *stubsbuilder* (siehe Unterabschnitt 5.4.1) erstellt und enthält weitere androidinterne Methoden für die Zuweisung von Kontext.

Als nächstes sind optional *Exclusions* festzulegen. Hierbei handelt es sich um einen Satz regulärer Ausdrücke[1] über die angegeben wird, welche Klassen nicht Teil der Analyse sein sollen. Dadurch wird die Analyse – gerade auch in Belangen des Speicherverbrauchs – deutlich schlanker. *Joana* liegt bereits eine Datei *exclusions.txt* für Java-Programme bei. Diese wurde um einige Android-Klassen erweitert.

Schließlich ist noch das zu analysierende Programm selbst anzugeben. *Android-AnalysisScope* kümmert sich hier um die passenden Einstellungen für die Datenextraktion aus dem *APK-Container*. Das so erstellte *AnalysisScope* wird zum einen jeweils den Funktionen *WALAs* übergeben, zum anderen in *Joanas scfg* gesetzt.

Analyseeinstellungen WALAs

🖶 com.ibm.wala.ipa.callgraph.AnalysisOptions

Einstellungen bezüglich der Art der Analyse legt *WALA* in der Klasse *Analysis-Options* ab. Diese Daten werden später durch *Joana* ersetzt werden, zu Beginn einer Analyse mittels *JoDroid* befinden sich jedoch noch einige Maßnahmen außerhalb der Kontrolle *Joanas*. Somit muss ein vorläufiger Satz von *AnalysisOptions* gesetzt werden.

Zunächst wird eine frische Instanz der *AnalysisOptions* ohne Definition jeglicher Einsprungspunkte erstellt. Diese wird daraufhin zunächst mit Standardvarianten von Selektoren für das Auffinden von Klassen und Überschreibungen (*BypassLogic*) versehen. Diese Einstellungen sind für die Erstellung des Macromodells bereits hinreichend. Es wird nun die Methode für dieses Modell synthetisiert und als einziger Einsprungspunkt gesetzt.

[1] Es ist auch eine direkte Benennung von Klassen ohne Verwendung regulärer Ausdrücke möglich, was in der Analyse allerdings nicht zum Einsatz kam

Die Überschreibungen in den *AnalysisOptions* werden daraufhin um solche erweitert, die für die kontextfreie Intentauflösung gemäß Unterabschnitt 4.6.1 zuständig sind. Weiterhin werden die Optionen mit einem *IntentContextSelector* und einem *IntentContextInterpreter* (siehe Unterabschnitt 5.2.4) versehen, die Aufrufe, die sie nicht verstehen, an die Standardbehandlungen delegieren.

All diese Änderungen werden abschließend auch in *Joanas scfg* hinterlegt, sodass sie auch auf die von *Joana* erneut generierten *AnalysisOptions* angewandt werden können.

6.1.2 Konfiguration der Lebenszyklusmodellierung

com.ibm.wala.dalvik.util.AndroidEntryPointManager

Einstellungen bezüglich der Generierung des Android-Lebenszyklus werden in der *Singleton-Klasse AndroidEntryPointManager* vorgenommen. Die dortigen Einstellungen wurden auf den Standardeinstellungen belassen. Diese seien im Folgenden beschrieben.

Mechanismus der Einsprungspunktselektion

com.ibm.wala.dalvik.util.AndroidEntryPointLocator.LocatorFlags

Für die Erkennung der Einsprungspunkte wurden sämtliche der in Abschnitt 4.1 beschriebenen *Flags* gesetzt. Dies zieht nach sich, dass durch eine Heuristik (siehe Unterabschnitt 4.1.1) alle Funktionen, die eine Funktion Androids überschreiben, als Einsprungspunkt selektiert werden. Weiterhin werden nicht überladene Funktionen der Superklassen hinzugefügt. Diese Einstellung liegt nicht in *AndroidEntryPointManager* vor sondern wird dem *AndroidEntryPointLocator* direkt übergeben.

Instanziierungsverhalten

Das Instanziierungsverhalten von Variablen wurde jeweils wie von *JoDroid* vorgeschlagen übernommen. Die so gewählten Verhalten beruhen somit auf einer automatisierten Abwägung zwischen Konservativität und Separation von Datenpfaden.

Struktur des Lebenszyklus

In Abschnitt 3.4 und Unterabschnitt 4.5.5 wurden mögliche Strukturen für die Generierung des Lebenszyklus vorgestellt. In den folgenden Analysen kommt hier ein *LoopAndroidModel* zum Einsatz – also eine Variante, die den Neustart von Applikationen auf Grund von Speicherknappheit nicht berücksichtigt. Für eine generelle *IFC-Analyse* ohne Wissen über den internen Aufbau der analysierten Programme sollte an dieser Stelle das konservativere *LoopKillAndroidModel* bevorzugt werden.

Generierung der Startumgebung

Über die Einstellung *doBootSequence* kann das Modell angewiesen werden eine erweiterte Startumgebung vor Ausführung des Lebenszyklus anzulegen. Diese Einstellung war bei den späteren Analysen nicht aktiv. Die Aktivierung dieser Einstellung setzt voraus, dass im *AnalysisScope* zuvor erweiterte *Stubs* geladen wurden.

Setzen bekannter Intents

Die Liste bekannter *Intents* kann aus Androids *Manifest-Datei* gelesen werden. Dieser Schritt wurde bei allen Analysen durchgeführt. Von der Möglichkeit diese Liste nach Erstellung des Aufrufsgraphen zu erweitern (und den Aufrufsgraph dann erneut zu erstellen) wurde nicht Gebrauch gemacht. Auch wurden keine Intents manuell angelegt.

6.1.3 Einstellungen bezüglich des Systemabhängigkeitsgraphen

Einige Einstellungen *Joanas* werden die zuvor in den *AnalysisOptions* getroffenen Einstellungen ersetzen. Weitere Einstellungen sind orthogonal dazu.

Sensitivitätseinstellungen

Joana bietet für die Erstellung des Aufrufgraphen mehrere Möglichkeiten bezüglich der Sensitivität. Die Analysen wurden jeweils *objektsensitiv* erstellt.

Eine objektsensitive Analyse hat einen hohen Ressourcenaufwand. Um sie für die Anwendungen zu ermöglichen wurden diese zunächst lediglich mit der Einstellung *CONTEXT_SENSITIVE* analysiert. Anschließend wurden anhand dieses Ergebnisses die *Exclusions* erweitert und gegebenenfalls die *Stubs* angepasst. So wurde der Analyseumfang jeweils schlank genug für die Einstellung *OBJECT_SENSITIVE* gemacht.

Weitere Einstellungen

Weitere vorgenommene Einstellungen sind in Tabelle 6.1 aufgeführt und seien nicht weiter im Detail beschrieben. Einstellungen, die keinen Einfluss auf das Analyseergebnis haben sind nicht aufgeführt.

Wie man sieht ist die Analyse für Ausnahmen deaktiviert. Sie ist für das aktuelle Modell noch nicht vorgesehen: Ausnahmen würden über Komponentengrenzen hinweg verschleppt. Auch werfen die verwendeten *Stubs* sehr viele *Exceptions*.

Die alternative Einstellung *ACCURATE* für die statische Initialisierung wurde in *Joana* noch nicht komplett umgesetzt. Aus diesem Grund wurde Abstand von ihr genommen.

Einstellung	Wert
scfg.immutableNoOut	Main.IMMUTABLE_NO_OUT
scfg.immutableStubs	Main.IMMUTABLE_STUBS
scfg.ignoreStaticFields	Main.IGNORE_STATIC_FIELDS
scfg.exceptions	ExceptionAnalysis.IGNORE_ALL
scfg.prunecg	Main.DEFAULT_PRUNE_CG
scfg.staticInitializers	StaticInitializationTreatment.SIMPLE
scfg.fieldPropagation	FieldPropagation.OBJ_GRAPH
scfg.computeInterference	false

Tabelle 6.1: *Weitere Einstellungen in Joana*

6.1.4 IFC-Einstellungen

In der *IFC-Analyse* mussten bis auf die Wahl von Quellen und Senken keine besonderen Einstellungen vorgenommen werden.

Selektion der Datenquellen und -Senken

Es wurden Versuche unternommen Quellen und Senken automatisiert mittels des Werkzeuges *SuSi* [34] zu selektieren. Dies führte allerdings zur Ausgabe zahlreicher androidinterner Datenpfade als Sicherheitsverletzung.

Aus diesem Grund wurde auf die manuelle Selektion der Annotationen zurückgegriffen. Die jeweils durchgeführten Einstellungen sind programmspezifisch und in den späteren Analysen jeweils gesondert dargestellt.

Wahl des Sicherheits-Verbands

Im Umfang der Versuche mit *SuSi* wurden zunächst mehrwertige Verbände herangezogen. Bei der manuellen Selektion weniger Quellen und Senken war der standardmäßig eingestellte zweiwertige Verband jedoch hinreichend.

Analyse von Nebenläufigkeiten

Diese Einstellung war bei den späteren Analysen unerheblich: Entweder wiesen sie nur einen *Thread* auf oder die in implizit erstellten *Threads* verlaufenden Datenpfade haben keinen Einfluss auf das Ergebnis. Die Einstellung verblieb somit auf dem Standardwert *"Classical non-interference"*.

6.2 Vorgehen bei den Analysen

Für die jeweils analysierten Programme wurde zunächst die Datei *AndroidManifest.xml* mittels dem Programm *apktool* aus dem *APK-Container* extrahiert und anschließend zusammen mit der *APK-Datei* in einem eigenen Ordner abgelegt.

Anschließend wurde mittels eines Scan-Vorgangs Konfigurationsdatei (*.ntrP-Datei*) für die Modellgenerierung erstellt. Die hierfür an *JoDroid* zu übergebenden Parameter lauten auf: *–scan normal –manifest AndroidManifest.xml*

Die so generierte Datei wurde jeweils unverändert für die Generierung des *Systemabhängigkeitsgraphen* (*.pdg-Datei*) übernommen. Der Verwendete Aufruf dafür lautet demnach auf *–manifest AndroidManifest.xml –exclusions exclusions.txt*[2] *–ep-file *.ntrP –construct main*. Der Wert *Main* für die Konstruktion weist *JoDroid* an die Analyse mit dem *Main-Intent* zu starten. Ist kein solcher *Intent* vorhanden, musste auf die alternative Einstellung *All*, durch die alle Komponenten gleichberechtigt gestartet werden, zurückgefallen werden.

Die so erstellte *.pdg-Datei* wurde nun in die *IFC-Console Joanas* geladen, Quellen und Senken (wie später beschrieben) manuell gesetzt und die *IFC-Analyse* schließlich gestartet.

Stichprobenartig wurden für einige Sicherheitsverletzungen *Chops*[3] generiert und genauer betrachtet. *Chops* lassen sich in der *IFC-Console* entweder durch Eingabe des Befehls *chop* mit zwei Programmpunkten oder alternativ durch Start der Konsole mit der Option *-Dchopview* graphisch erzeugen. Da die *Chops* mitunter einen sehr großen Umfang annehmen können sei in den folgenden Betrachtungen nicht weiter auf sie eingegangen. Sie waren allerdings ein wichtiges Werkzeug für das Debugging von *JoDroid*.

Allen aufgeführten Befehlen gemein ist, dass Java über die Optionen *-Xmx2048m -Xss16m* mehr Arbeitsspeicher zugesichert und über *-ea* Assertions aktiviert wurden.

6.3 Vergleich mit alternativen Programmen

Die mittels *JoDroid* generierten Ergebnisse wurden lediglich im Falle von *DroidBench* mit dem alternativen Werkzeug *FlowDroid* verglichen, da hier bereits Werte vorlagen.

Bei *FlowDroid* [16] [15] handelt es sich um eine Erweiterung des Werkzeuges *Soot* [32] für die Analyse von Android-Applikationen.

---

[2] Die Exclusions wurden wie erwähnt angepasst um ein schlankeres Ergebnis zu erhalten. Lässt man den Parameter weg, so werden Standardwerte genommen

[3] *Chops* sind grob mit *Slices* (siehe Unterabschnitt 2.1.6) vergleichbar. Sie geben den Programmcode auf Pfaden zwischen zwei Punkten aus.

6.4 Analyse der Anwendungen

In diesem Abschnitt wird nun die Analyse einiger Programme dargestellt. Zunächst werden die Ergebnisse *JoDroids* anhand der *Benchmark-Suite DroidBench* betrachtet. Anschließend mit *AllComponentsTest* ein ausführlicheres Programm analysiert. Schließlich findet mit *Insecure Bank* ein Programm aus anderem Umfeld Betrachtung.

6.4.1 DroidBench

Bei *DroidBench* [12] handelt es sich um einen Satz kleiner Programme, welche jeweils einer Schwierigkeit bei einer Analyse zugeordnet sind. Anhand dieser Programme lässt sich die Leistung eines Analysesystems bewerten.

Die durchgeführten 58 (der 62) Benchmarks verfügen über durchschnittlich 1,7 Klassen und eine durchschnittliche Gesamtzahl von 3,8 Methoden. Nicht berücksichtigt wurden dabei die automatisch generierten Klassen *R* und *BuildConfig*. In den jeweiligen Quelltexten der Programme sind Datenquellen und -Senken vorgegeben, diese wurden in der Analyse manuell übernommen. Die von *JoDroid* vorgeschlagenen Einstellungen für Instanziierungsverhalten und Platzierung der Einsprungspunkte wurde jeweils unverändert übernommen.

Soweit nicht anders angegeben wurde jeweils die weniger konservative Option *–construct main* verwendet, bei der die Analyse mit dem Start des entsprechenden Intents beginnt.

Im Folgenden seien die Analyseergebnisse der einzelnen Tests dargestellt und mit den Ergebnissen einer Analyse mit *FlowDroid* [15] verglichen. Ein Ergebnis ist dabei optimal, wenn die Anzahl der gefundenen Schwachstellen mit der Anzahl der erwarteten Schwachstellen (spalte *Lecks* in den folgenden Tabellen) übereinstimmt und keine Schwachstelle unerkannt blieb.

Analysen unter AndroidSpecific

Tests unter diesem Präfix finden sich Programme in sehr einfacher Form: Von einer einzigen Android-Komponente werden lediglich wenige Einsprungspunkte implementiert (oft lediglich ein einziger). Alle Quellen und Senken befinden sich in dieser Komponente.

Das besondere an *InactiveActivity* ist, dass sie über ein *Flag* in der *Manifest-Datei* als deaktiviert markiert ist. Es kann somit keine Instanz der Klasse durch das System erstellt werden. Da die *Activity* mit einem *Intent-Filter* versehen ist wird sie jedoch unter Umständen dennoch exportiert. Nach Aussage der Autoren von *DroidBench* stellt eine über das *Manifest* deaktivierte *Activity* kein Sicherheitsrisiko dar.

[4] Der Wert wurde aus [16] übernommen.

Test	Quellen	Senken	Lecks	FlowDroid	JoDroid
DirectLeak1	1	1	1	1[4]	1
InactiveActivity	1	1	0	0[4]	1
Library1	1	0	0		-
Library2	0	1	1		-
LogNoLeak	0	1	0	0[4]	0
Obfuscation1	1	1	1		1
PrivateDataLeak1	1	2	1	1[4]	**0**
PrivateDataLeak2	1	1	1	1[4]	1
PrivateDataLeak3	1	1	1		**0**

Tabelle 6.2: *Ergebnisse der Analysen unter AndroidSpecific*

Eine Analyse des Paares *Library1* und *Library2* konnte nicht vorgenommen werden: *Library2* benötigt Code aus *Library1*, die Analyse zweier *APK-Pakete* zur selben Zeit ist in *JoDroid* jedoch derzeit nicht vorgesehen.

Das Verpassen der Schwachstellen bei *PrivateDataLeak1* liegt darin begründet, dass die Senke in einer *CallBack-Funktion* liegt, welche in einer Layout XML-Datei einem Bedienelement zugeordnet wird. Die aktuelle Version von *JoDroid* berücksichtigt allerdings keine solchen Ressourcen-Dateien.

In *PrivateDataLeak3* werden sensible Daten in das Dateisystem geschrieben. Die Analyse müsste erkennen, dass die selbe Datei später wieder gelesen wird. Erst von dort aus wird die Senke erreicht.

Analysen unter Livecycle

Auch die Tests mit dem Präfix *Livecycle* stellen lediglich eine Android Komponente zur Verfügung. Im Falle von *ActivityLifecycle2* ist deren Implementierung auf zwei Klassen aufgeteilt.

Datenquellen und *-Seneken* liegen jeweils in unterschiedlichen Einsprungspunkten der Komponenten, die gemäß des Lebenszyklus nacheinander aufgerufen werden. Es kommen verschiedene Typen von Komponenten zum Einsatz.

Die *ApplicationLifecycle*-Tests verwenden zusätzlich zu den Einsprungspunkten der Komponenten weitere, die für die gesamte Applikation zutreffen.

Die Schwachstelle in *ActivityLifecycle1* wird nicht erkannt, da sich *java.net.URL* in den Exclusions befindet. Der Pfad bis unmittelbar vor die Verwendung von *URL* würde erkannt.

Der Fluss in *ActivityLifecycle2* wird lediglich mit der konservativeren Einstellung *–construct all*, jedoch nicht mit *–construct main* erkannt.

Auch die Analysen, die nicht *Activitys* betreffen müssen mit der Option *–construct all* erstellt werden, sei es weil kein *"Main"-Intent* existiert oder weil die Application sonst nicht berücksichtigt würde.

Test	Quellen	Senken	Lecks	FlowDroid	JoDroid
ActivityLifecycle1	1	1	1	1[4]	0
ActivityLifecycle2	1	1	1	1[4]	0/1
ActivityLifecycle3	1	1	1	1[4]	1
ActivityLifecycle4	1	1	1	1[4]	1
ApplicationLifecycle1	1	1	1		1
ApplicationLifecycle2	1	1	1		1
ApplicationLifecycle3	1	1	1		1
BroadcastReceiver1	1	1	1	1[4]	1
ServiceLifecycle1	1	1	1	1[4]	1

Tabelle 6.3: *Ergebnisse der Analysen unter Livecycle*

Analysen unter InterAppCommunication

Die Benennung dieser Analysen ist irreführend: *ActivityCommunication1* implementiert zwei Komponenten in der selben Applikation und übergibt Daten mittels einer statischen Variable.

Die Tests *IntentSink1* und *IntentSink2* verwenden zwar je ein *Intent* jedoch wird die Verwendung dieses *Intents* direkt als *Senke* markiert.

Es findet also keinerlei tatsächliche Kommunikation über das System statt.

Test	Quellen	Senken	Lecks	FlowDroid	JoDroid
ActivityCom.1	1	1	1	1[4]	0/1
IntentSink1	1	1	1	**0**[4]	1
IntentSink2	1	1	1	1[4]	1

Tabelle 6.4: *Ergebnisse der Analysen unter InterAppCommunication*

Der Datenpfad in *ActivityCommunication1* beruht darauf, dass eine zweite *Activity* durch eine externe Applikation gestartet wird. Demnach wird der Pfad nur unter Verwendung von *–construct all* erkannt.

Analysen unter Callbacks

In diesen Analysen kommen *Callback*-Funktionen zum Einsatz, welche bei Benutzerinteraktion oder Statusänderung des Gerätes durch Android angesprungen werden. Nur wenige solcher Funktionen sind in *JoDroid* fest hinterlegt, weitere werden jedoch durch eine Heuristik (siehe Unterabschnitt 4.1.1) erkannt.

FlowDroid verwendet für die Erkennung eine fest hinterlegte Liste potentieller Klassen.

Im Falle des Tests *Button2* liegt die Zuordnung der *CallBack-Funktion* wieder in einer XML-Datei vor. Diese *Layout-Dateien* werden von *JoDroid* derzeit nicht behandelt.

Die Analyse von *MultiHandlers1* terminiert derzeit nicht.

Test	Quellen	Senken	Lecks	FlowDroid	JoDroid
AnonymousClass1	1	1	1	1[4]	1
Button1	1	1	1	1[4]	1
Button2	1	3	2	4[4]	**0**
Button3	1	1	1		0/1
LocationLeak1	1	2	2	2[4]	2
LocationLeak2	1	2	2	2[4]	2
LocationLeak3	1	1	1		1
MethodOverride1	1	1	1	1[4]	1
MultiHandlers1	1	2	0		-
Ordering1	1	2	0		0
RegisterGlobal1	1	1	1		**0**
RegisterGlobal2	1	1	1		**0**
Unregister1	1	1	0		1

Tabelle 6.5: *Ergebnisse der Analysen unter Callbacks*

Die Analysen *RegisterGlobal* und *Unregister1* können in der aktuellen Version nicht das korrekte Ergebnis liefern, da benötigte Funktionen in den *Stubs* fehlen. Jedoch auch wenn sie vorhanden wären ist nicht von einer korrekten Behandlung auszugehen.

Analysen unter ImplicitFlows

Diese Tests obskurifizieren die Daten entweder direkt oder geben abhängig von den Werten der als Quelle markierten Daten unterschiedliche konstante Ergebnisse aus.

Test	Quellen	Senken	Lecks	FlowDroid	JoDroid
ImplicitFlow1	1	1	1	**0**[4]	1
ImplicitFlow2	1	2	2	**0**[4]	2
ImplicitFlow3	1	2	2	**0**[4]	2
ImplicitFlow4	1	5	2	**0**[4]	2[5]

Tabelle 6.6: *Ergebnisse der Analysen unter ImplicitFlows*

Analysen unter GeneralJava

Die Analysen in diesem und den folgenden Anschnitten haben den Zweck das generelle Verhalten des Analysewerkzeuges unabhängig von Eigenheiten Androids zu prüfen. Da die Testprogramme jedoch in einem androidspezifischen Format vorliegen, können sie nicht unabhängig von den anderen Tests betrachtet werden.

[5] Eine Senke im *Catch-Block* konnte nicht markiert werden.

Test	Quellen	Senken	Lecks	FlowDroid	JoDroid
Loop1	1	1	1	1[4]	1
Loop2	1	1	1	1[4]	1
SourceCodeSpecific1	1	1	1	1[4]	1
StaticInitialization1	1	1	1	**0**[4]	**0**
UnreachableCode	1	1	0	0[4]	0

Tabelle 6.7: *Ergebnisse der Analysen unter GeneralJava*

Die Analysen, die *Exceptions* behandeln wurden nicht durchgeführt: Die vorliegende Version des Lebenszyklusmodells kann mit Ausnahmen nicht korrekt umgehen: Sie würden über Komponentengrenzen hinweg getragen. Aus diesem Grund wurde bei der Erstellung des Systemabhängigkeitsgraphs die Exception-Analyse deaktiviert.

In dem Test *StaticInitialization1* konnte die Senke in dem statischen Initialisierungscode nicht markiert werden: Sie wird in der *IFC-Console* nicht angezeigt. Selbst wenn sie markierbar wäre, wäre eine Erkennung des Pfades aufgrund der Art der Behandlung von diesen Initialisierern in *WALA* unwahrscheinlich.

Die nicht erreichbare Funktion in *UnreachableCode* wird zunächst durch die *CB_HEURISTIK* als Einsprungspunkt selektiert. Ob sie in dem Modell aufgerufen wird hängt von den Einstellungen ab.

Analysen unter FieldAndObjectSensitivity

Dieser Satz von Programmen enthält keine tatsächlichen Sicherheitsverletzungen: Zu Klassen existieren hier mehrere Instanzen. Eine Instanz enthält dabei sensitive Daten, eine andere Instanz schreibt Daten auf eine unsichere Senke. Die Aufgabe des Analysewerkzeuges ist es nun zwischen den Instanzen zu unterscheiden um festzustellen, dass kein Informationsleck existiert.

Test	Quellen	Senken	Lecks	FlowDroid	JoDroid
FieldSensitivity1	1	1	0	0[4]	0
FieldSensitivity2	1	1	0	0[4]	0
FieldSensitivity3	1	1	1	1[4]	1
FieldSensitivity4	1	1	0	0[4]	0
InheritedObjects1	1	1	1	1[4]	1
ObjectSensitivity1	1	1	0	0[4]	1
ObjectSensitivity2	1	2	0	0[4]	0

Tabelle 6.8: *Ergebnisse der Analysen unter FieldAndObjectSensitivity*

Bei *ObjectSensitivity1* wären zwei *LinkedLists* zu unterscheiden. Für diese liegt allerdings keine entsprechende Behandlung vor, sodass alle Instanzen diesen Typs zusammengefasst betrachtet werden.

Analysen unter ArraysAndLists

Diese Tests weisen große Ähnlichkeit zu den *FieldAndObjectSensitivity-Tests* auf. Jedoch kommen hier unterschiedliche Positionen in Kollektionen zum Einsatz.

Test	Quellen	Senken	Lecks	FlowDroid	JoDroid
ArrayAccess1	1	1	0	1^4	1
ArrayAccess2	1	1	0	1^4	1
HashMapAccess1	1	1	0		1
ListAccess1	1	1	0	1^4	1

Tabelle 6.9: *Ergebnisse der Analysen unter ArraysAndLists*

Für die Erkennung des Zugriffs auf unterschiedliche Elemente einer *Collection* wäre eine *String-Analyse* nötig. Diese ist derzeit noch nicht in *Joana* implementiert.

Analysen unter Reflection

Die Verwendung von *Reflection* stellt die Analysewerkzeuge vor das Problem, dass der Typ von Objekten erst zu einem späteren Zeitpunkt in der Analyse – eventuell erst anhand von berechneten Objektbezeichnern – bekannt wird.

Test	Quellen	Senken	Lecks	FlowDroid	JoDroid
Reflection1	1	1	1		1
Reflection2	1	1	1		0
Reflection3	1	1	1		0
Reflection4	1	1	1		0

Tabelle 6.10: *Ergebnisse der Analysen unter Reflection*

6.4.2 AllComponentsTest

Das Programm *AllComponentsTest* entstand während der Entwicklung des Lebenszyklusmodells. Es wurde bei Änderungen am Modell jeweils um neue Funktionalitäten erweitert.

	ohne automatisch generierten Code	mit automatisch generiertem Code
Anzahl der Klassen	7	22
Anzahl der Methoden	25	42
Codezeilen	355	581

Tabelle 6.11: *Eckdaten von AllComponentsTest*

Das Programm weist lediglich eine Quelle und eine Senke auf. Der durch die Analyse zu erkennende Datenpfad verläuft jedoch durch jegliche Art androidtypischer Grundkomponenten. Zwischen den Komponenten werden die Daten auf jeweils unterschiedlichen Wegen übergeben.

Interner Aufbau von AllComponentsTest

Die Applikation liest zunächst aus dem Telefonbuch die lokalen Index-Nummern aus. Nach mehrfacher Weitergabe der Daten werden sie abschließend in einem HTTP-Header in das Netzwerk gesendet.

Bei Start der *Main-Activity* (*onCreate*) wird zunächst einem Bedienelement ein *onClick*-Callback zugewiesen. Ausgehend von dieser Funktion wird der Ladevorgang initiiert.

Für das Laden wird zunächst ein *CurserLoader* mit Ziel eines applikationsinternen *ContentProviders* erstellt. Dieser leitet Anfragen an einen androideigenen *ContentProviders* weiter, der das Telephonbuch zur Verfügung stellt.

Sobald die Daten verfügbar sind (*onLoadFinished*) werden diese zu einem *Intent* hinzugefügt. Mittels dieses *Intents* wird die Klasse *DHIntentService* gestartet. In dessen Funktion *handleActionGrab* werden die Daten in der Klasse selbst hinterlegt. Sie werden von dort erst später wieder abgeholt.

Die Klasse *DHIntentService* versendet nun einen *Broadcast*, welcher durch die Klasse *DumpedReceiver* empfangen wird. Dieser startet wiederum einen Intent, welcher die Komponente *HopActivity* startet.

Die *HopActivity* startet in *onResume* zunächst ein Intent, welches den *DHIntentService* dazu veranlasst die Daten in eine *Singleton-Klasse* zu schreiben. Anschließend holt sie die Daten in ihrer Funktion *onPause* von dort ab.

Nun wird ein *AsyncTask*, *HTTPDownload*, mit den Daten als Argument gestartet. Diese Daten werden an die *run*-Methode von *HTTPDownload* übergeben. Von dort aus werden sie schließlich in das Internet versandt.

Ausschluss von Klassen

Bei der Erstellung von *AllComponentsTest* wurde zunächst eine Grundstruktur mittels eines *Wizzards* von *Eclipse* erstellt. Da dieser für Programme mit graphischer Oberfläche bereits einigen Quelltext erzeugt steigt die Größe des Systemabhängigketisgraphen massiv.

Ein weiteres Problem liegt in der Instanziierung von Objekten, deren Typ nicht eindeutig aufgelöst werden kann: Hier muss die Analyse konservativ approximieren.

Um das Programm dennoch objektsensitiv analysieren zu können mussten somit einige Klassen von der Analyse ausgeschlossen werden. Dies geschieht durch die Übergabe einer Datei *exclusions.txt* an *JoDroid*. Ein Auszug der für die Analyse verwendeten Ausschlüsse findet sich in Listing 20.

```
java/lang/StringBuilder
android/content/IntentSender
android/media/.*
```

Listing 20: *Auszug der Ausschlüsse bei AllComponentsTest*

Entfernen der Support-Bibliothek

Viele Programme beinhalten in ihrem Paket eine Version der *Android-Support Bibliothek* für Rückwärtskompatibilität. Es bietet sich an diese vor der Analyse zu entfernen. Die *Android-Stubs* beinhalten bereits die nötige Funktionalität.

Eventuell ist dabei darauf zu achten, dass es sich bei der dem Programm beigefügten Bibliothek um eine unveränderte Variante handelt.

Anpassungen an der Einsprungspunktdatei

Zunächst wurden Einsprungspunkte entfernt, die einen Parameter des Typs *Object* aufweisen. Hier müsste die Analyse konservativ approximieren. Da somit zu viele Klassen als Kandidat für diesen Parameter gewählt würden wäre das Ergebnis unbrauchbar. Die entfernten Einträge lauten auf:

```
HopActivity$HTTPDownload.doInBackground([Ljava/lang/Object;)Ljava/lang/Object;
MainActivity.onLoadFinished(Landroid/content/Loader;Ljava/lang/Object;)V
```

Weiterhin wurden einige Einsprungspunkte entfernt, die durch die verwendete Heuristik generiert wurden. Würde man diese berücksichtigen, do gäbe es alternative Pfade von Quelle zu Senke. In der Analyse soll allerdings berücksichtigt werden, ob Intents korrekt verarbeitet werden. Diese Pfade sind demnach unerwünscht. Die somit entfernten Einträge lauten auf:

```
MainActivity$2.run()V
DHIntentService.startActionDump(Landroid/content/Context;Ljava/io/Serializable;)V
DHIntenuService.handleActionDump(Ljava/io/Serializable;)V
DHIntentService.startActionGrab(Landroid/content/Context;Ljava/lang/String;)V
DHIntentService.handleActionGrab(Ljava/lang/String;)V
MainActivity.delayedHide(I)V
MainActivity.access$2(Lde/tobiasblaschke/allcomponentstest/MainActivity;I)V
MainActivity.access$1(Lde/tobiasblaschke/allcomponentstest/MainActivity;)...
```

Selektion der Datenquelle und -Senke

Die eigentliche Quelle der Analyse wäre:

```
com.android.internal.telephony.ArrayListCursor
```

In der vorliegenden Version kann jedoch noch nicht zu dieser Stelle aufgelöst werden: Aufgrund später beschriebener Gründe wurde die Berücksichtigung von *ContentProvidern* in der Analyse sogar komplett deaktiviert.

Die stattdessen selektierte Quelle ist der Zugriff auf den Cursor:

```
MainActivity.onLoadFinished(Landroid/content/Loader;Landroid/database/Cursor;)V:2 high
```

Als einzige Datensenke wurde das Versenden der Daten gewählt. Es ist der erste Parameter der Funktion *doInBackground*:

```
HopActivity$HTTPDownload.doInBackground([Ljava/lang/String;)Ljava/lang/String;->p1 low
```

Ergebnis der Analyse

Eine manuelle Analyse des *Systemabhängigkeitsgraphen* mittels *Graphviewer* (siehe Abbildung 2.5.1) ergab, dass sämtliche *Intents* eindeutig und korrekt aufgelöst wurden. Die Ersetzung einzelner Funktionen durch *Wrapper* (siehe Unterabschnitt 4.6.5) und die damit verbundene Generierung eingeschränkter Modelle (siehe Unterabschnitt 4.6.4) erfolgte ebenfalls ohne Probleme.

Die im Programm enthaltene Sicherheitsverletzung wurde durch die *IFC-Analyse* erkannt.

6.4.3 Insecure Bank

Bei *Insecure Bank* von *Paladion Labs* [23] handelt es sich um ein Programm, mit dem man Beträge zwischen Konten transferieren kann. Transaktionen werden dabei in einer lokalen Datenbank gespeichert und über eine HTTP-Verbindung auf einem entfernten Server ausgeführt.

Die Software wurde absichtlich derart entwickelt, dass sie Schwachstellen aufweist. So werden unter anderem kritische Daten in die globale Log-Datei geschrieben, bei Transaktionen Geräteinformationen an den Server übermittelt und Informationen in das Dateisystem und *SharedPreferences* geschrieben.

	ohne automatisch generierten Code	mit automatisch generiertem Code
Anzahl der Klassen	9	17
Anzahl der Methoden	26	26
Codezeilen	573	623

Tabelle 6.12: *Eckdaten von Insecure Bank*

Analysiert man das Programm in seiner ursprünglichen Form, so wird man ein stark überapproximiertes Analyseergebnis erhalten. Im Folgenden wird die

Umwandlung in ein äquivalentes Programm beschrieben und erklärt, warum die jeweiligen Anpassungen vorgenommen wurden.

Überführung in ein äquivalentes Programm

Bestimmte Konstrukte innerhalb eines Programmes können durch *Joana* derzeit noch nicht in hinreichendem Detailgrad gefasst werden. Treten sie auf, so muss an diesen Stellen konservativ approximiert werden, was ein ungenaueres Resultat der Analyse nach sich zieht.

Diese Konstrukte wurden durch solche ersetzt, die einen vergleichbaren Effekt haben, jedoch in der Analyse besser verarbeitet werden können.

Auffinden von UI-Objekten

Das Auffinden von Elementen der Benutzeroberfläche kann zur Laufzeit durch *findViewById* erfolgen. Für eine eindeutige Zuordnung zu einem tatsächlichen Objekt müsste *Joana* hier eine sogenannte *String-Analyse* durchführen. Dies ist in der aktuellen Version des Werkzeuges jedoch noch nicht vorgesehen, wodurch eine Kombination aller möglichen Kandidaten von der Funktion zurückgegeben wird.

Eine Betrachtung des Quelltextes von *Insecure Bank* zeigte, dass jede *Id* lediglich einmal nachgeschlagen wird. Somit konnte das Auftreten von *findViewById* jeweils durch eine passende *New-Anweisung* ersetzt werden.

Entfernen des Aufrufes getBaseContext

Durch die Funktion *ContextWrapper.getBaseContext(Context)* wird ausgehend von dem *Context* einer Komponente ein dort hinterlegter *Context* gewählt. Diese Methode ist in der verwendeten Version von *Stubs* nicht abgebildet, sodass wiederum von allen verfügbaren *Context* ausgegangen wird und die Analyse überapproximiert.

Dieses Problem wurde durch Entfernen des Aufrufes und Verwendung des als Parameter angegebenen *Context* behoben.

Ausschluss von StringBuilder

Es wurden mehrere Implementierungen von *StringBuilder* getestet, jedoch führte keine zu dem gewünschten Ergebnis. Der Ausschluss dieser Klasse aus der Analyse (und die somit vorgenommene Standardbehandlung für ausgeschlossene Klassen) führte zu dem gewünschten Ergebnis.

Generierung des Systemabhängigkeitsgraphen

Der Systemabhängikeitsgraph wurde *objektsensitiv* ausgehend von dem *Main-Intent* erzeugt. Eine *Exception-Analyse* fand wiederum nicht statt.

Da die Berücksichtigung zu vieler Funktionen – gerade in einer objektsensitiven Analyse – die Größe des Systemabhängigkeitsgraphen (und damit die Verarbeitungszeit) stark wachsen lässt wurden Einstellungen so gewählt, dass dieser möglichst kompakt verbleibt jedoch wichtige Funktionen nicht überspringt.

Dies schlägt sich zunächst in der Selektion der Einsprungspunkte nieder: Tabelle 6.13 zeigt die dort vorgenommenen Einstellungen.

Flag	**Wert**	**Begründung**
EP_HEURISTIC	aktiviert	impliziert durch *CB_HEURISTIC*
CB_HEURISTIC	aktiviert	benötigt für UI-CallBacks
WITH_SUPER	deaktiviert	massive Verkleinerung des SDG
INCLUDE_CALLBACKS	aktiviert	hat bei *Insecure Bank* keinen Effekt

Tabelle 6.13: *Auszug aus den Analyseeinstellungen (scan) von Insecure Bank*

Ein Auszug weiterer Einstellungen, welche die Erstellung des Systemabhängigketisgraphen beeinflussen findet sich in Tabelle 6.14.

Einstellung	**Wert**	**Begründung**
doBootSequence	deaktiviert	Verkleinerung des SDG
doFlatComponents	deaktiviert	
exceptions	IGNORE_ALL	Modell inkompatibel mit Exceptions
modelBehavior	LoopAndroidModel	weniger konservativ, hinreichend

Tabelle 6.14: *Auszug aus den Analyseeinstellungen (SDG) von Insecure Bank*

Die Berücksichtigung von *Exceptions* bei der Modellierung würde das Problem nach sich ziehen, dass diese oft auch Effekte über Komponentengrenzen hinweg mit einbeziehen würden. Das Ergebnis wäre zu konservativ.

Eine vollständige Liste aller Einstellungen findet sich in Anhang G.

Annotation der Datenquellen und -senken

Für die Analyse mittels *JoDroid* müssen zunächst wieder Datenquellen und -senken mit einer Sicherheitsannotation versehen werden. Da diese für die Software nicht vorgegeben sind wurde zunächst manuell der Quelltext nach Kandidaten durchsucht. In Listing 21 und Listing 22 sind die gewählten Annotationen dargestellt.

Ergebnis der Analyse

Eine Betrachtung des Ergebnisses zeigt zunächst, dass alle vorkommenden *Intents* und *SystemServices* korrekt aufgelöst werden.

In der *IFC-Analyse* werden – bis auf zwei Verletzungen (siehe unten) – alle Flüsse korrekt erkannt. Eine Überapproximierung liegt nicht vor.

Dass für die Quellen, die Gerätinformationen auslesen (*c* und *d* in Listing 21) die gleichen Ausgaben generiert werden ist direkt anhand des Quelltextes von *Insecure*

[6] Schöner wäre hier die Selektion einer Klasse. Leider werden die *sharedPreferences* unter Verwendung der aktuellen Stubs hierfür nicht weit genug aufgelöst.

a in Funktion DataHelper.selectAll
Der Aufruf von $SQLiteDatabase.db.query$ (ByteCode-Index 23 der Funktion selectAll)
b in Klasse LoginScreen
Das Feld $''Password_Text''$, welches später einem UI-Element zugeordnet wird.
c Funktionen zum Auslesen von Geräteinformationen (Es wurde jeweils die gesamte Funktion annotiert):
- TelephonyManager.getLine1Number()
- TelephonyManager.getSimSerialNumber()
- TelephonyManager.getDeviceId()

d in Funktion LoginScreen.dologin
Die Aufruf von $android.provider.Settings.Secure.getString(..., ANDROID_ID)$ (ByteCode-Index 97 der Funktion dologin)
e in Klasse PostLogin
Die Felder $''from''$, $''to''$ und $''ammount''$, die UI-Elementen zugeordnet sind
f in Funktion RawHistory.onCreate
Der Aufruf von $wView.loadUrl(''file://'' + ... +'' /rawhistory.html'')$ (ByteCode-Index 65 der Funktion onCreate)

Listing 21: *Als sensitiv eingestufte Datenquellen in Insecure Bank*

1. in Klasse DataHelper
Das Feld $''insertStmt''$, ein prepared Statement für den Datenbankzugriff

2. in Funktion LoginScreen.rememberme
Der Aufrufe von $putString$ (ByteCode-Indizes 46 und 67 der Funktion), die Daten in die *sharedPreferences* schreiben[6]

3. in Funktion PostLogin.dotransfer
Der Aufruf von $BufferedWriter.close()$ (ByteCode-Index 255 der Funktion), da hierdurch eine Datei geschrieben wird.

4. die Funktion RestClient.postHttpContent

Listing 22: *Als unsicher eingestufte Senken in Insecure Bank*

Bank ersichtlich: Bis auf *getLine1Number* werden die Daten direkt nacheinander ausgelesen, eine Weiterverarbeitung erfolgt mit Hilfe eines Strings "*deviceId*", der die kombinierten Informationen enthält.

Die direkt im Anschluss ausgelesene *getLine1Number* wird zusammen mit "*deviceId*" behandelt. Es ist somit offenkundig, dass die Meldungen zu dieser Quelle somit mit den vorherigen Meldungen übereinstimmen sollten.

Die Daten werden mittels der Funktion *RestClient.postHttpContent* (Senke 4) versandt. Der Datenfluss erfolgt über die Zwischenstation *RestClient.sidechannel*.

Nun seien die Textfelder, welche Transaktionsdaten beinhalten (e in Listing 21)

betrachtet. Für diese werden drei Pfade erkannt: Zu den Senken 1, 3 und 4.

Über einen *onClick*-Handler eines Knopfes der Activity *PostLogin* ist die Funktion *dotransfer* erreichbar. Von dort erfolgt ein Datenversand in das Internet über die Klasse *Transfer* (zu Senke 4). Anschließend wird die Transaktion in das Dateisystem geschrieben (Senke 3). Zuletzt werden die Daten in eine Datenbank geschrieben und dadurch Senke 1 erreicht.

Aufgrund einer *if*-Abfrage werden zu dem Feld *ammount* weitere Ausgaben generiert. Eine nähere Betrachtung dieser führt allerdings zu dem Ergebnis, dass diese den bereits genannten Senken zugeordnet ist.

Als nächstes sei die Quelle *b*, *Password_ Text*, betrachtet. Von hier aus existieren zunächst Pfade zu den Senken 2 und 4.

Das Schreiben der *sharedPreferences* (Senke 2) erfolgt durch die Methode *rememberme*, die durch einen *onClick*-Handler eines Knopfes erreichbar ist. Sie liest die Daten direkt aus dem Passwort-Feld aus.

Der Pfad ins Internet (Senke 4) erfolgt analog, jedoch über die Funktion *dologin()*.

Weiterhin sollten auch Pfade von *Password_ Text* zu den verbleibenden Senken existieren: Der Start von *PostLogin* hängt von einem Wert ab, der als *Json-Objekt* von einer HTTP-Verbindung zurückgegeben wird. Diese HTTP-Verbindung nimmt (wie erkannt) jedoch *Password_ Text* als Eingabe.

Das Verpassen der Verletzung liegt hier in den verwendeten *Stubs* und *Exclusions* begründet. Von einer Anpassung der *Stubs* wurde aufgrund der komplexen Zusammenhänge des Datenaustausches über das Netzwerk abgesehen.

Für die verbleibenden Quellen *a* und *f* werden keine Verletzungen ausgegeben: Die Daten von *selectAll* werden auf dem Bildschirm ausgegeben und in das *System-Log* geschrieben. Beides wurde nicht als Senke markiert. Die Daten der anderen Quelle werden ebenfalls lediglich auf dem Bildschirm ausgegeben.

KAPITEL 7

Probleme

Dieses Kapitel beschreibt Probleme, die bei der Umsetzung des Modells und bei der Ausführung von Analysen auftraten, sowie eingeschlagene Wege zur Lösung der Probleme.

7.1 Probleme während der Implementierung

Alle Probleme, die bei der Implementierung auftraten, konnten umgangen oder gelöst werden.

7.1.1 Verwendung von FakeRoot-Methoden

WALA nutzt für den Beginn einer Analyse sogenannte *FakeRoot*-Methoden um zunächst einen globalen Status für die Applikation herzustellen. Aus diesen Methoden heraus werden weiterhin die Einsprungspunkte des zu analysierenden Programms aufgerufen.

Erste Implementierungen Androids Lebenszyklus erweiterten diese Methoden um die für die Modellierung benötigten Instruktionen. Bei diesem Ansatz trat das Problem auf, dass die Art der gewählten *FakeRoot*-Methode von dem gewählten *CallGraphBuilder* abhängig ist und das Modell somit unter Umständen nicht Teil der Analyse war.

Hier wurden zunächst Konstrukte für die Delegierung zu weiteren *CallGraphBuildern* geschaffen um diesen frei wählbar zu machen. Ein weiteres Problem bestand darin, dass diese Aufrufe im *CallGraph* nicht aufgelöst wurden: Die *Intermediate Representation* enthält also eine *SSAAbstractInvokeInstruction*, für die *CallSite* dieser Instruktion waren allerdings keine *CGNodes* als Ziel hinterlegt. Diese Zuordnungen wurden explizit aufgelöst, was die neuerliche Problematik nach sich zog, dass deren *CallSites* ebenfalls nicht zugeordnet wurden.

Die aktuelle Implementierung generiert eine neue Methode abseits der *FakeRoot*-Methoden. Dieser Ansatz behebt die dargestellten Problematiken und ist deutlich flexibler. Durch die Reduktion der Komplexität des Quelltextes ist die Umsetzung nun auch besser lesbar.

7.1.2 Instanziierung von Variablen

Der zuvor erwähnte Verzicht auf die Implementierung in *FakeRoot*-Methoden zog das Problem nach sich, dass die *WALA-internen* Methoden zur Instanziierung von Variablen aus synthetischem[1] Code heraus nicht mehr verwendet werden konnten, da diese fest an *FakeRoot*-Methoden gebunden sind. Für diesen Zweck wurde eine neue Klasse (siehe Unterabschnitt 5.1.8) geschaffen. Diese scheint weiterhin flexibler zu sein als *WALAs* eigene Ansätze.

7.1.3 Zyklische Abhängigkeiten von Typen

Erste Implementierungen der Initialisierung von Variablen hatten das Problem, dass bei der Verwendung von *Interfaces* zyklische Abhängigkeiten möglich sind. In diesem Fall verfiel der Code in eine Endlosrekursion. Das Problem wurde dadurch behoben, dass vor der eigentlichen Instanziierung zunächst eine Menge konkreter Klassen generiert wird und Typen, die bereits Teil der Menge sind, übersprungen werden.

7.1.4 Erkennung der Ursache von Exceptions

WALA prüft in vielen Funktionen nicht die übergebenen Parameter. Dies führt dazu, dass Exceptions teilweise erst später auftreten, als die Stelle des fehlerhaften Quelltextes. Im Folgenden seien erkannte mögliche und nicht offensichtliche Ursachen solcher *Exceptions* aufgeführt.

UnimplementedError: How can parameter be implicit?

Diese Ausnahme tritt dann auf, wenn einer *SSA-Nummer*, die einem Parameter zugeordnet ist, in dem Körper der zugehörigen Funktion ein Wert zugewiesen wird.

Finden die in Abschnitt 5.1 beschriebenen Werkzeuge Verwendung sollte ein solcher Fehler nicht mehr auftreten können.

IllegalArgumentException: DST=null

SSAInstructions haben sowohl einen *iindex* als auch einen *ProgramCounter*. Diese *Exception* kann ein Hinweis darauf sein, diese vertauscht wurden.

Binary-op-instruction 'and' is signed

Entsteht bei einem Feldzugriff auf eine Instanz einer Klasse, welche sich nicht

[1] Synthetisch im Sinne der *WALA-Implementierung*, nicht im Sinne des *Java-Bytecodes*

in korrektem Zustand befindet. Dieser Fehler tritt unter anderem auf, wenn im Modell ein Typ auf *CREATE* gesetzt ist und somit nicht durch vorherige Einsprungspunkte modifiziert wurde.

Illegal Value-Number -1: In node ...
Tritt bei einer negativen *SSA-Nummer* auf. Unerwartet ist der Fehler, wenn eine *InvokeInstruction* für *void*-Funktionen für eine Funktion mit Rückgabewert verwendet wurde.

Die im Rahmen der Arbeit durchgeführte Implementierung enthält ausgiebigere Prüfungen um eine solche Verschleppung von Fehlern zu vermeiden.

7.1.5 Behandlung des komprimierten Formats von AndroidManifest.xml

Android legt sein *Manifest* in einem binärcodierten Format ab. Zwar existieren Bibliotheken zur Behandlung dieses Formats diese verwenden allerdings eine aktuellere Version der *ANTLR*-Bibliothek als *Joana*. Die jeweiligen Versionen sind inkompatibel zueinander.

In Java ist es nicht möglich[2] eine Bibliothek in zwei Versionen zu laden. Aus diesem Grund konnte keine der Bibliotheken zum Lesen des komprimierten Formats herangezogen werden.

In der aktuellen Implementierung muss demnach vor Start der Analyse diese Datei (beispielsweise mit dem Programm *apktool*) in ein herkömmliches XML-Format entpackt werden.

7.2 Generelle Fragestellungen zur Analysierbarkeit von Android Applikationen

Die folgenden Probleme stellen generelle Schwierigkeiten bei einer statischen Analyse von Android-Applikationen dar. Ihre Lösbarkeit bedarf zumindest eingehenderen Betrachtungen.

[2] Es ist möglich über einen angepassten *ClassLoader* zunächst eine Version zu laden, diese später wieder zu entfernen und eine andere Version zu laden. Allerdings zieht dies auch nach sich, dass durch den angepassten *ClassLoader* Änderungen an Javas *Code-Cache* durchgeführt werden müssen um die Versionen nicht zu vermischen. Aufgrund der Komplexität und Fehleranfälligkeit dieses Ansatzes wurde von einer Umsetzung abgesehen.

7.2.1 Berücksichtigung pseudo-externer Abhängigkeiten

Android-Applikationen tauschen intern häufig Daten mittels *Singleton-Klassen* aus[3]. Aus diesem Grund muss Wert auf die Reihenfolge der aufgerufenen Funktionen gelegt werden.

Startet eine interne Komponente A eine weitere interne Komponente B, so wird die Abhängigkeit aktuell erkannt[4]. Ruft nun A jedoch eine externe Komponente C auf, die wiederum B startet, so kann nicht sichergestellt werden, dass die Abhängigkeit erkannt wird. Abhilfe schafft hier derzeit lediglich die Wahl einer konservativeren Modellstruktur (siehe Unterabschnitt 4.5.5).

7.2.2 Behandlung eingehender Intents

Androidkomponenten werden jeweils durch ein *Intent* gestartet. In einer statischen Analyse ist es nicht möglich zu erkennen, dass eine Komponente nicht durch eine externe Applikation aufgerufen werden kann. Wird eine Komponente durch eine externe Applikation aufgerufen, so können zum einen sensitive Daten zusammen mit dem *Intent* in eine Komponente gelangen; zum anderen können sensitive Daten die Applikation über die Funktion *setResult* verlassen.

Möchte man nun eine *IFC-Analyse* für eine Android-Applikation ausführen, so gibt es die Möglichkeit alle Funktionen, die Daten aus dem *Intent* lesen oder Daten als Rückgabewert setzen als unsicher einzustufen. Hierunter würden auch die Funktionen *getIntent*, *newIntent* und *setResult* fallen. Durch dieses Vorgehen wird man in einer Analyse sehr viele Sicherheitswarnungen generieren.

Eine andere Möglichkeit bestünde darin sämtliche auf dem Gerät installierte Software zu analysieren und dadurch eine Liste möglicher Aufrufer und deren Sicherheitsangaben zu erlangen.

Eine letzte Möglichkeit besteht darin sich auf die Angaben in *AndroidManifest.xml* zu berufen, ob ein *Intent* zu einer Komponente überhaupt exportiert wird. Dies birgt aber zum einen die Gefahr, dass es eventuell Möglichkeiten zum Umgehen dieser Einstellungen geben könnte. Zum anderen wird dadurch keine Applikation davon abgehalten eine Klasse dynamisch in den eigenen Adressraum zu laden und dort auszuführen.

---

[3] Hierzu liegen keine Praxisdaten vor. Da dieses Vorgehen in Androids Developer-Guide [9] empfohlen wird sei an dieser Stelle davon augegangen, dass dem so ist.

[4] Es sei denn man deaktiviert alle *Overrides* in *AndroidEntryPointManager*

7.2.3 Direkte Speicherzugriffe

Wie in Unterabschnitt 2.3.1 beschrieben ist es in einer Android-Applikation möglich eine Komponente derart zu gestalten, dass sie in einem Prozess einer anderen Anwendung ausgeführt wird. Bietet diese Möglichkeit ein sehr mächtiges Werkzeug, so ist sie unter Sicherheitsaspekten kritisch zu betrachten: Jeder Prozess verfügt über einen eigenen Speicherbereich. Wenn man Komponenten zu einem fremden Prozess hinzufügt, so muss man starkes Augenmerk darauf legen, dass keine Informationen aus diesem Speicher nach außen gelangen.

Bei der hier nötigen Sicherheitsbetrachtung ist es auch nötig auf native Programme einzugehen. Da diese CPU-gebunden sind scheint eine geschlossene Betrachtung der Einflüsse auf die Datensicherheit schwer möglich.

7.2.4 Geschlossene Analysierbarkeit einer einzelnen Applikation

Aufgrund der soeben betrachteten Möglichkeiten des Speicherzugriffes und Aufgrund von Problematiken mit *Intents*, wie sie in Unterabschnitt 7.2.2 und Unterabschnitt 8.1.2 dargestellt sind, stellt sich die Frage, inwiefern eine Applikation überhaupt geschlossen analysiert werden kann.

Der einzig gangbare Weg scheint hier die Analyse sämtlicher Anwendungen, die auf einem Gerät installiert sind. Nur so kann sichergestellt werden, dass sensitive Informationen nicht unerwartet nach außen getragen werden können.

KAPITEL 8

Offene Arbeiten

Dieses Kapitel führt weitere Arbeiten auf, die entweder von vornherein nicht Teil der Arbeit waren oder zu denen Ideen im Zuge der Implementierung entstanden. Weitere Gebiete, in welchen genauere Betrachtungen möglich sind finden sich in Abschnitt 7.2.

8.1 Ideen zur Verbesserung der Genauigkeit

Die folgenden Unterabschnitte beinhalten Ideen, wie die Modellierung des Lebenszyklus von Androidanwendungen noch genauer gestaltet werden kann.

8.1.1 Verwendung von Komponenten externer Applikationen

Durch die ausgeprägte Möglichkeit Androids Komponenten fremder Anwendungen zu integrieren wird die Analyse einer einzelnen Applikation erschwert: Alle abhängigen Komponenten sollten Teil der Analyse sein. In Unterabschnitt 7.2.2 wird auf die Problematik des Aufrufes einer Komponente der analysierten Anwendung durch eine externe Anwendung eingegangen. An dieser Stelle soll der Aufruf einer externen Komponente durch die analysierte Anwendung betrachtet werden.

Bindet eine Android-Anwendung Komponenten einer externen Anwendung ein, so ist es möglich diese Komponente grundsätzlich als unsicher einzustufen. Es gibt allerdings auch Möglichkeiten einer differenzierteren Betrachtung: Auch in einer statischen Analyse ist es begrenzt[1] möglich Aussagen darüber zu treffen, welche externe Komponente geladen wird. Ausgehend von dieser Information können nun mehrere Ansätze gewählt werden.

[1] Wenn das Sprungziel durch Zusammensetzen eines *Strings* zur Laufzeit definiert wird ist es zur Analysezeit schwer zu erkennen. In diesem Fall müssen alle Ziele als Möglichkeit angenommen werden

Zum einen ist es möglich die externe Applikation zu dem Umfang der analysierten Anwendung hinzuzufügen. Dadurch erscheint der Aufruf dem Analysewerkzeug nicht mehr extern. Das Problem dieses Ansatzes liegt in dem hohen Ressourcenverbrauch für die kombinierte Analyse mehrerer Anwendungen.

Zum anderen bestünde eine Möglichkeit darin die Sicherheitseinstellungen aus *AndroidManifest.xml* der Zielapplikation auszulesen und basierend auf diesen Daten die ungünstigsten Annahmen zu treffen. Hierbei ist zu berücksichtigen, dass durch die Verwendung von *IntentSender*-Objekten sonst allgemeingültige Zugriffsbeschränkungen umgangen werden können.

Die letzte Möglichkeit findet sich in der Erstellung einer Datenbank bereits analysierter Komponenten. Aus einer solchen Datenbank könnte eine Zusammenfassung der Ergebnisse der vorherigen Analyse übernommen werden.

8.1.2 Umgang mit impliziten Intents

Eng verknüpft mit den vorherigen Betrachtungen ist die Fragestellung, wie mit impliziten *Intents* umgegangen werden soll. In diesem Fall ist es nicht mehr offen ersichtlich, durch welche Anwendung Daten zur Verfügung gestellt werden.

Hier besteht die Möglichkeit einer installationsspezifischen Analyse, bei der bekannt ist, durch welche Applikationen ein solches *Intent* bearbeitet werden kann.

Alternativ müssten alle Applikationen auf dem Markt, welche dieses *Intent* zur Verfügung stellen analysiert werden.

8.1.3 Verfeinerung von Instanziierungsverhalten

In der aktuellen Umsetzung sind mit den Einstellungen *CREATE* und *REUSE* lediglich zwei extreme Formen von Instanziierungsverhalten umgesetzt: Die erste Einstellung unterdrückt Datenabhängigkeiten, die andere induziert Abhängigkeiten unabhängig von der Umgebung, in der ein Typ Verwendung findet. Letzteres kann zu Fehlalarmen führen.

Das Ergebnis einer Analyse könnte durch die Einführung weiterer Verhalten deutlich verfeinert werden: Abhängig von dem aufzurufenden Einsprungspunkt oder der Android-Komponente, zu der ein Typ als Parameter auftaucht könnte eine passende Instanz aus einem Satz bekannter Instanzen passenden Typs wiederverwendet beziehungsweise dieser Satz erweitert werden.

Ein Beispiel für die Sinnhaftigkeit solchen Verhaltens findet sich in dem Typ *android.os.Bundle*, einer Kollektion beliebiger Daten. Von diesem Typ ist bekannt, dass er als Parameter zu den Funktionen *onCreate* und *onSaveInstanceState* einer *Activity* als Parameter auftaucht. Für diese beiden Funktionen sollte eine Instanz dieses Typs pro *Activity* vorhanden sein. Zugriff auf diese Instanz von außerhalb der genannten Funktionen ist nicht möglich.

Aktuell kann dieser Typ lediglich auf *REUSE* gesetzt werden, wodurch fälschlicherweise Datenabhängigkeiten zum einen zwischen diesen Funktionen aller Activitys

induziert werden, zum anderen taucht dieser Typ auch in Verbindung mit *Intents* auf. Erste Vorkehrungen für die Implementierung eines weiteren Instanziierungsverhaltens, welches genannte Problematik berücksichtigen könnte, wurden getroffen. Es existiert derzeit jedoch noch keine Umsetzung.

8.1.4 Genaueres Abbilden weiterer Lebenszyklen

In dem in der Arbeit erstellten Modell wurden Lebenszyklen einiger wichtiger Klassen abgebildet. Allerdings existieren unter Android eine Vielzahl an Unterarten dieser Klassen oder weitere unabhängige Klassen, welche einen eigenen Zyklus von Rückruffunktionen aufweisen. Für diese Klassen könnten man zusätzliche Positionierungsinformationen anlegen um ihr Verhalten innerhalb des Modells genauer abbilden zu können.

Damit verbunden könnte man eine Analyse durchführen, inwiefern das Ergebnis der Gesamtanalyse durch genauere Platzierung beeinflusst wird.

8.1.5 Auswertung Androids Kontextinformationen

Android erstellt intern Kontextinformationen zu Komponenten und Anwendungen. Im Zuge der Modellierung werden entsprechende Informationen derzeit lediglich rudimentär und unvollständig erzeugt. Auch die Auswertung dieser Informationen ist kaum umgesetzt.

Androids Kontext wird vor allem dann interessant, wenn Kommunikation über Applikationsgrenzen hinweg erfolgen soll. In diesem Fall sind dort unter anderem Zugriffsrechte für die jeweiligen Aufrufe hinterlegt.

Da diese Informationen durch Android zur Laufzeit evaluiert werden ist deren Betrachtung im Rahmen einer *IFC-Analyse* nicht zwingend erforderlich, könnte aber unter Umständen die Anzahl von Fehlalarmen reduzieren.

8.1.6 Auswertung von Elementen der Benutzerschnittstelle

Das Aussehen einer Ansicht einer Androidkomponente wird häufig in den Ressourcen eines Anwendungspaketes definiert. Anhand dieser Definitionen werden dann vor dem Zeichnen einer Ansicht entsprechende Objekte erzeugt.

Diese Informationen werden in der derzeitigen Umsetzung nicht explizit ausgewertet. Dadurch ist zu der Analysezeit nicht bekannt, welche Steuerelemente sich auf der Benutzeroberfläche befinden. Entsprechend kann nicht zwischen *Rückruf-Funktionen* der Steuerelemente differenziert werden. In der aktuellen Implementierung werden genannte *Rückruf-Funktionen* anhand einer Heuristik selektiert und in einer vereinigten Form für alle Steuerelemente aufgerufen. Dadurch kann beispielsweise nicht zwischen dem Klicken auf unterschiedliche Bedienelemente unterschieden werden.

Ein Werkzeug, welches zu solchen Unterscheidungen fähig ist, ist *FlowDroid*. Eine Betrachtung inwiefern eine explizite Betrachtung der *Rückruf-Funktionen* das Analyseergebnis verbessert könnte vorgenommen werden.

8.2 Unbehandelte Aufgaben

Die im Folgenden kurz aufgeführten weiteren Erweiterungen waren im Zug der Implementierung angedacht, konnten jedoch aus Zeitgründen nicht mehr umgesetzt werden.

8.2.1 Berücksichtigung des Originals bei der Überschreibung von Methoden

Im Zuge der Auflösung von Intents werden in der Applikation auftretende Start-Funktionen (siehe Anhang D) durch *Wrapper-Funktionen* gemäß Unterabschnitt 4.6.5 ersetzt. Durch diese Ersetzung wird die originale Variante (aus der *Android-API*) der jeweiligen Funktion nicht mehr berücksichtigt. Dies sollte im Sinne einer konservativen Analyse allerdings dennoch der Fall sein. Leider ist in *WALA* ein Aufruf der originalen Variante der Funktion schwer umzusetzen.

8.2.2 Verbesserung der Auflösung von Intents

Bei der kontextsensitiven Generierung des Aufrufgraphen werden *Intents* wie in Unterabschnitt 4.6.2 beschrieben aufgelöst. Diese Auflösung kann weiter verfeinert werden.

Derzeit werden *"System-Intents"*, also *implizite Intents* deren Namen durch Android vorgegeben ist nicht besonders behandelt. Liegen Überschreibungen für solche *Intents* vor, so kann es passieren, dass externe Ziele außer Acht gelassen werden. Ansonsten können solche *Intents* derzeit lediglich wie *Intents* mit unbekanntem Ziel behandelt werden.

Eine weitere Aufgabenstellung befindet sich in der Interpretation von *URI*-Informationen. Da diese aktuell ebenfalls nicht aufgelöst werden können sollten aktuell auch alle *URI-behafteten Intents* als *Intents* unbekannten Ziels betrachtet werden.

Auch die Behandlung von Funktionen wie *startActivities*, die gleich mehrere *Intents* zum Ziel haben, werden derzeit nicht korrekt behandelt.

8.2.3 Erweiterung des Scanvorgangs

Es war angedacht in einem erweiterten Suchvorgang vor der Nachbearbeitung der Modellierungseinstellungen durch den Benutzer bereits eine leichtgewichtige Version eines Aufrufgraphen zu erstellen. Dadurch wäre es möglich alle in der Applikation vorkommenden *Intents* (also auch Ziele in externen Anwendungen) aufzulisten. Anhand dieser Informationen könnten die Einstellungen für die Auflösung der *Intentziele* besser bearbeitet werden.

Derzeit ist ein solches Verhalten möglich, beruht aber auf dem kompletten Erstellen eines vorläufigen Systemabhängigkeitsgraphen[2]. Es ist somit sehr kostenintensiv.

8.2.4 Einlesen weiterer Analyseparameter

Derzeit werden einige Einstellungen der Analyse lediglich in die *ntrP-Datei* (siehe Anhang E) serialisiert, jedoch nicht wieder zurückgelesen.

Eine Anpassung solcher Analyseeinstellungen durch den Benutzer ist somit lediglich grob durch die Wahl eines anderen "Presets" mittels einer Kommandozeilenoption möglich.

Würden diese Einstellungen aus der Datei wieder deserialisiert wäre eine viel feingranularere Steuerung des Analysevorgangs möglich.

[2] Nach Erstellung des Systemabhängigketsgraphen wird die *ntrP-Datei* erneut geschrieben und enthält dann die nötigen Informationen. Basierend auf dieser neuen Variante dieser Datei muss – so ein anderes Verhalten als in der Datei angegeben ist gewünscht ist – der *SDG* dann noch einmal erstellt werden

KAPITEL 9

Fazit

In dieser Arbeit wurde eine (teil-)automatisierte Generierung des Lebenszyklus von Android-Applikationen vorgestellt und implementiert. Aufgrund der hohen Konfigurierbarkeit und der Umsetzung in *WALA* direkt, ermöglicht die erstellte Implementierung nicht nur die Betrachtung von Android-Applikationen im Rahmen einer *IFC-Analyse* mittels *JoDroid* und *Joana*; viel mehr kann sie auch als Ausgangspunkt für andere *WALA-Basierte* Analysewerkzeuge, welche unter anderem auch eine weniger konservative Art der Modellierung bevorzugen könnten, zur Integration von Kompatibilität zu Android-Anwendungen genutzt werden.

Für ähnliche Implementierungen eines Lebenszyklusmodells können die separat gehaltenen Spezifikationen von Einsprungspunkten und der damit verbundenen Information der Platzierung, sowie die ausgiebigen Spezifikationen der Funktionen, durch die weitere Komponenten Androids gestartet werden können, eine Hilfe sein: Die Daten sind derart hinterlegt, dass die leicht zu erweitern oder zu übernehmen sind.

Die Integration eines allgemein gehaltenen Werkzeugkastens für die Generierung synthetischer Funktionen zur Analysezeit kann auch für andere Entwicklungen an *WALA* nützlich sein und hilft verbreitete Fehler zu vermeiden. Wie die restliche Implementierung ist auch sie im Quelltext hinreichend dokumentiert.

Mit wenig Anpassung ließe sich das Modell auch für Programme abseits von Android, welche auch mehrere Einsprungspunkte aufweisen, verwenden: In Verbindung mit der sich aktuell in Entwicklung befindlichen Unterstützung von *JavaScript* durch *WALA* beispielsweise für Browser-Plugins.

Was die unter Verwendung des erstellten Modells generierten *IFC-Analysen* mittels *JoDroid* und *Joana* anbelangt so scheint es, dass einfache Android-Applikationen mit geringfügigen Änderungen an dem Programm selbst bereits recht genau analysierbar sind:

Wie aus den Analysen in Kapitel 6 hervorgeht ist die Abbildung des Lebenszyklus von Android-Anwendungen für die Analyse kleiner Applikationen sowie der Testprogramme aus *DroidBench* bereits recht genau. Typische Android-Anwendungen

weisen unter Umständen ein hohes Maß an Kommunikation mit Komponenten dritter Anwendungen auf. Hier würde das Ergebnis durch die in Kapitel 8 dargestellten Maßnahmen deutlich verbessert; doch auch in dem aktuellen Zustand der Software scheinen solche Anwendungen analysierbar – jedoch mit höherem manuellen Aufwand bezüglich des Ausschlusses fälschlicherweise ausgegebener Sicherheitswarnungen[1].

Eine weitere Einschränkung bei der Analyse typischer Anwendung besteht in dem hohen Bedarf an Rechenzeit und Speicherressourcen, die bereits für die Generierung des Systemabhängigkeitsgraphen erforderlich sind. Auch hier besteht hohes Potential für Optimierungen.

Eine zusätzliche Fragestellung bei der Analyse handelsüblicher Anwendungen, für die kein Quelltext vorliegt, besteht in der Selektion von Datenquellen und -Senken. Der Versuch diese mittels *SuSi* (siehe Unterabschnitt 6.1.4) automatisch zu selektieren stellt zumindest bei Verwendung von *JoDroid* derzeit noch keinen praktikablen Ansatz dar, da sehr viele Warnungen aufgrund androidinterner Datenpfade ausgegeben werden – abhängig von den verwendeten *Stubs* können jedoch auch Pfade übersehen werden. Aus der Liste dieser Warnungen solche herauszufiltern, welche auch bei der Analyse eines leeren Programms auftreten würden, scheint ein gefährlicher Ansatz zu sein, da dadurch auch Pfade des analysierten Programms betroffen sein könnten.

In der aktuellen Fassung erstellt *Joana/JoDroid* – soweit dem Autor bekannt – genauere Analysen, als vergleichbare Werkzeuge. Dies resultiert in einer geringeren Zahl von Fehlalarmen; die Zahl unerkannter Schwachstellen ist sehr gering, Gründe für ihr Verpassen sind bekannt und liegen beispielsweise in der Verwendung von *Reflection* oder der Datenweitergabe über das Dateisystem. Dennoch werden derzeit noch nicht alle Möglichkeiten der Verfeinerung der Analyse voll ausgeschöpft.

[1] Jegliche Kommunikation mit externen Anwendungen muss hier als unsicher eingestuft werden. Die dadurch induzierten Warnungen müssen anschließend manuell überprüft werden.

ANHANG **A**

Liste der registrierten Einsprungspunkte

🖮 com.ibm.wala.dalvik.util.androidEntryPoints

Für einige wichtige Klassen sind Funktionen, die als Einsprungspunkt dienen können mit Positionierungsinformationen fest im Quelltext der im Umfang dieser Arbeit vorgenommenen Anpassungen an *WALA* hinterlegt. Die fraglichen Funktionen sind im Folgenden aufgeführt.

A Einsprungspunkte der Grundkomponenten

In den folgenden Abschnitten sind Funktionen von Android-Komponenten benannt, deren Klassen durch ein Intent zum Start einer Applikation führen können.

A.1 android.app.Application

🖮 com.ibm.wala.dalvik.util.androidEntryPoints.ApplicationEP

Für jede Android-Applikation kann maximal eine Klasse existieren, die *Application* überlädt. Eine solche Klasse kann zur Erstellung einer Umgebung für die gesamte Applikation genutzt werden. Dort definierte Einsprungspunkte werden (bis auf Einsprungspunkte von ContentProviedern) vor den Einsprungspunkten der Android-Komponenten ausgeführt.

onCreate

nach AT_FIRST, ContentProvider.-onCreate

vor Activity.onCreate, Service.onCre-ate

onConfigurationChanged

nach END_OF_LOOP, Activity.on-ConfigurationChanged

vor AT_LAST

onLowMemory

nach END_OF_LOOP, Activity.on-LowMemory

vor AFTER_LOOP, Application.on-ConfigurationChanged

onTrimMemory

Direkt vor Application.onLowMe-mory

A.2 android.content.ContentProvider

🖨 com.ibm.wala.dalvik.util.androidEntryPoints.ProviderEP

Ein *ContentProvider* dient dazu fremden Anwendungen Daten zur Verfügung zu stellen. Die ersten Funktionen im Lebenszyklus von *ContentProvidern* werden bereits vor den Funktionen von *Application* angesprungen.

onCreate

position AT_FIRST

query

nach START_OF_LOOP, Content-Provider.onCreate

insert

nach START_OF_LOOP, Content-Provider.onCreate

onConfigurationChanged

nach START_OF_LOOP, Content-Provider.onCreate

onLowMemory

nach START_OF_LOOP, Content-Provider.onCreate

onTrimMemory

nach START_OF_LOOP, Content-Provider.onCreate

update

nach START_OF_LOOP, Content-Provider.onCreate

A.3 android.app.Activity

🖨 com.ibm.wala.dalvik.util.androidEntryPoints.ActivityEP

Activity-Komponenten haben grundsätzlich eine Graphische Ausgabe. Jede Ansicht hat in der Regel eine eigene Activity.

onCreate

nach AT_FIRST

onStart

nach START_OF_LOOP, Activity.-onCreate

onRestoreInstanceState

nach Activity.onStart

onPostCreate

nach Activity.onRestoreInstanceSta-te, START_OF_LOOP, Activity.onStart

onResume

nach Activity.onPostCreate, Activity.-onRestoreInstanceState

onPostResume

nach Activity.onResume

onNewIntent

nach Activity.onPostCreate, Activity.-onRestoreInstanceState

vor Activity.onResume

onStop

nach AFTER_LOOP

onRestart

nach Activity.onStop

onSaveInstanceState

nach Activity.onPostResume

onPause

nach Activity.onResume, Activity.-onSaveInstanceState, MIDD-LE_OF_LOOP

onDestroy

nach Activity.onPause, At_LAST

onActivityResult

nach MULTIPLE_TIMES_IN_-LOOP

vor Activity.onPause

dispatchPopulateAccessibilityEvent

nach AT_FIRST, Activity.onCreate

onCreateDialog

nach START_OF_LOOP, Activity.-onResume, Activity.onPostCre-ate, Activity.dispatchPopulate-AccessibilityEvent

vor MIDDLE_OF_LOOP, Activity.onStop, Acitvity.onPause, Activity.onSaveInstanceState

onPrepareDialog

nach Activity.onCreateDialog, START_OF_LOOP, Activity.-onResume, Activity.onPostCre-ate, Activity.dispatchPopulate-AccessibilityEvent

vor MIDDLE_OF_LOOP, Activity.onStop, Acitvity.onPause, Activity.onSaveInstanceState

onCreateView

nach START_OF_LOOP, Activity.-onResume, Activity.onPostCre-ate, Activity.dispatchPopulate-AccessibilityEvent

vor MIDDLE_OF_LOOP, Activity.onStop, Acitvity.onPause, Activity.onSaveInstanceState

onAttachFragment

nach Activity.onCreateView, START_OF_LOOP, Activity.-onResume, Activity.onPostCre-ate, Activity.dispatchPopulate-AccessibilityEvent

vor MIDDLE_OF_LOOP, Activity.onStop, Acitvity.onPause, Activity.onSaveInstanceState

onAttachedToWindow

nach Activity.onCreateView, START_OF_LOOP, Activity.-onResume, Activity.onPostCre-ate, Activity.dispatchPopulate-AccessibilityEvent

vor MIDDLE_OF_LOOP, Activity.onStop, Acitvity.onPause, Activity.onSaveInstanceState

onDetachedFromWindow

nach Activity.onAttachedToWindow

vor AFTER_LOOP, Activity.onStop, Activity.onSaveIn-stanceState, Activity.onPause

onContentChanged

nach Activity.onCreateView, START_OF_LOOP, Activity.-onResume, Activity.onPostCre-ate, Activity.dispatchPopulate-AccessibilityEvent

vor MULTIPLE_TIMES_IN_-LOOP, Activity.onStop, Activity.onPause, Activity.onSaveIn-stanceState

onApplyThemeResource

Direkt nach Activity.onStart

onCreatePanelView

nach START_OF_LOOP, Activity.-onResume, Activity.onPostCre-ate, Activity.dispatchPopulate-AccessibilityEvent

vor MIDDLE_OF_LOOP, Activity.onStop, Acitvity.onPause, Activity.onSaveInstanceState

onCreatePanelMenu

nach START_OF_LOOP, Activity.-onResume, Activity.onPostCre-ate, Activity.dispatchPopulate-AccessibilityEvent

vor MIDDLE_OF_LOOP, Activity.onStop, Acitvity.onPause, Activity.onSaveInstanceState

onPreparePanel

nach START_OF_LOOP, Activity.-onResume, Activity.onPostCre-ate, Activity.dispatchPopulate-AccessibilityEvent, Activity.on-CreatePanelMenu, Activity.on-CreatePanelView

vor MIDDLE_OF_LOOP, Activity.onStop, Acitvity.onPause, Activity.onSaveInstanceState

onPanelClosed

nach Activity.onCreatePanelMenu, Activity.onCreatePanelView, Activity.onPreparePanel, MIDDLE_OF_LOOP, Activity.onRe-sume, Activity.onPostCreate, Activity.dispatchPopulateAcce-ssibilityEvent

vor AFTER_LOOP

onCreateContextMenu

nach START_OF_LOOP, Activity.-onResume, Activity.onPostCre-ate, Activity.dispatchPopulate-AccessibilityEvent

vor MIDDLE_OF_LOOP, Activity.onStop, Acitvity.onPause, Activity.onSaveInstanceState

onContextItemSelected

nach Activity.onCreateContext-Menu, Activity.onResume, Activity.onPostCreate, Activity.-dispatchPopulateAccessibility-Event, MIDDLE_OF_LOOP

vor Activity.onPanelClosed

onContextMenuClosed

nach Activity.onCreateContext-Menu, Activity.onContextItem-Selected, START_OF_LOOP, Activity.onResume, Activity.on-PostCreate, Activity.dispatchPo-pulateAccessibilityEvent

vor AFTER_LOOP

onCreateOptionsMenu

Direkt nach Activity.onCreateCon-textMenu

onOptionsItemSelected

Direkt nach Activity.onContext-ItemSelected

onPrepareOptionsMenu

nach Activity.onCreateOptions-Menu, START_OF_LOOP, Activity.onResume, Activity.on-PostCreate, Activity.dispatchPo-pulateAccessibilityEvent

vor Activity.onOptionsItemSelected, MIDDLE_OF_LOOP, Activity.onStop, Acitvity.onPause, Activity.onSaveInstanceState

onOptionsMenuClosed

Direkt nach Activity.onContext-MenuClosed

onMenuOpened

nach Activity.onCreateOptions-Menu, Activity.onPrepareOp-tionsMenu, Activity.onCreate-ContextMenu, START_OF_-LOOP, Activity.onResume, Activity.onPostCreate, Activity.-dispatchPopulateAccessibility-Event

vor Activity.onOptionsItemSelected, Activity.onContextItemSele-cted, MIDDLE_OF_LOOP, Activity.onStop, Acitvi-ty.onPause, Activity.onSaveIn-stanceState

onMenuItemSelected

nach Activity.onCreateContext-Menu, Activity.onPrepa-reOptionsMenu, Activi-ty.onMenuOpened, START_-OF_LOOP, Activity.onResume, Activity.onPostCreate, Activity.-dispatchPopulateAccessibility-Event

vor Activity.onOptionsItemSelected, Activity.onContextItemSele-cted, MIDDLE_OF_LOOP, Activity.onStop, Acitvi-ty.onPause, Activity.onSaveIn-stanceState

onTitleChanged

nach START_OF_LOOP, Activity.-onResume, Activity.onPostCre-ate, Activity.dispatchPopulate-AccessibilityEvent

onChildTitleChanged

Direkt nach Activity.onTitleChan-ged

onUserInteraction

nach Activity.onResume,Activity.on-PostCreate, Activity.dispatchPo-pulateAccessibilityEvent, MUL-TIPLE_TIMES_IN_LOOP

dispatchTouchEvent

nach Activity.onUserInteraction, Activity.onResume, Activity.on-PostCreate, Activity.dispatchPo-pulateAccessibilityEvent, MUL-TIPLE_TIMES_IN_LOOP

onTouchEvent

nach Activity.onUserInteraction, Activity.dispatchPopulateAcce-ssibilityEvent, Activity.dispatch-TouchEvent

dispatchGenericMotionEvent

nach Activity.onUserInteraction, Activity.onResume, Activity.on-PostCreate, Activity.dispatchPo-pulateAccessibilityEvent, MUL-TIPLE_TIMES_IN_LOOP

onGenericMotionEvent

nach Activity.dispatchGenericMo-tionEvent

dispatchTrackballEvent

nach Activity.dispatchPopulateAcce-ssibilityEvent, Activity.onUser-Interaction, Activity.onGeneric-MotionEvent Activity.onResu-me,Activity.onPostCreate, MUL-TIPLE_TIMES_IN_LOOP

onTrackballEvent

nach Activity.dispatchTrackball-Event, Activity.onUserInterac-tion, Activity.onGenericMotion-Event

dispatchKeyEvent

nach Activity.onUserInteraction, Activity.onTrackballEvent,

Activity.onResume, Activity.on-PostCreate, Activity.dispatchPo-pulateAccessibilityEvent, MUL-TIPLE_TIMES_IN_LOOP

dispatchKeyShortcutEvent

nach Activity.dispatchKeyEvent, Activity.onUserInteraction, Activity.onTrackballEvent, Activity.-onResume, Activity.onPostCre-ate, Activity.dispatchPopula-teAccessibilityEvent, MULTI-PLE_TIMES_IN_LOOP

onKeyDown

nach Activity.onUserInteraction, Activtiy.dispatchKeyEvent, Activity.onTrackballEvent

onKeyLongPress

nach Activtiy.dispatchKeyEvent, Activity.onKeyDown

onKeyMultiple

nach Activtiy.dispatchKeyEvent, Activity.onKeyDown

onKeyShortcut

nach Activtiy.dispatchKeyEvent, Activity.dispatchKeyShortcut-Event, Activity.onKeyDown

onKeyUp

nach Activity.dispatchKeyEvent, Activity.onKeyDown, Activity.onKeyLongPress, Activity.onKeyMultiple, Activity.onKeyShortcut

onBackPressed

nach Activity.dispatchKeyEvent, Activity.onKeyDown, Activity.onKeyUp

onCreateNavigateUpStack

nach START_OF_LOOP

vor Activity.onPause, Activity.on-SaveInstanceState,AFTER_-LOOP

onPrepareNavigateUpTaskStack

nach Activity.onCreateNavigateUp-Stack

vor Activity.onPause, Activity.on-SaveInstanceState, END_OF_-LOOP

onNavigateUpFromChild

nach Activity.onCreateNavigateUp-Stack

vor Activity.onCreateNavigateUp-Stack, Activity.onPrepareNa-vigateUpTaskStack, Activity.-onSaveInstanceState, Activity.onPause

onNavigateUp

nach Activity.onNavigateUpFrom-Child

vor Activity.onCreateNavigateUp-Stack, Activity.onPrepareNa-vigateUpTaskStack, Activity.-onSaveInstanceState, Activity.onPause

onSearchRequested

nach MULTIPLE_TIMES_IN_-LOOP, Activity.onKeyUp, Activity.onTrackballEvent, Activity.onOptionsItemSelected, Activity.onContextItemSele-cted, Activity.onMenuItemSe-lected

onActionModeStarted

position MULTIPLE_TIMES_-IN_LOOP

onActionModeFinished

nach Activity.onActionModeStarted

onWindowStartingActionMode

nach MULTIPLE_TIMES_IN_-LOOP

vor Activity.onActionModeStarted

onConfigurationChanged

nach Activity.onStop

vor Activity.onRestart

onCreateDescription

nach Activity.onSaveInstanceState

vor Activity.onPause

onCreateThumbnail

Direkt vor Activity.onCreateDesc -ription

onProvideAssistData

nach FILL

vor MIDDLE_OF_LOOP

onRetainNonConfigurationInstance

nach Activity.onStop

vor Activity.onDestroy

onLowMemory

nach END_OF_LOOP

vor AFTER_LOOP, Activity.onCon -figurationChanged

onTrimMemory

Direkt vor Activity.onLowMemory

onUserLeaveHint

Direkt vor Activity.onPause

onWindowAttributesChanged

nach MULTIPLE_TIMES_IN_ -LOOP

vor END_OF_LOOP

onWindowFocusChanged

Direkt nach Activity.onResume

A.4 Fragment

🖮 com.ibm.wala.dalvik.util.androidEntryPoints.FragmentEP

Durch *Fragments* können Teilbereiche einer graphischen Ausgabe gekapselt und wiederverwendet werden. Ein Fragment alleine kann nicht über ein Intent gestartet werden sondern ist immer Teil einer *Activity*.

onAttach

nach AT_FIRST, ContentProvider. -onCreate, Application.onCreate

vor START_OF_LOOP

onCreate

nach AT_FIRST, Fragment.onAt -tach

vor START_OF_LOOP

onCreateView

nach AT_FIRST, Fragment.onCre -ate

vor START_OF_LOOP

onActivityCreated

nach AT_FIRST, Fragment.onCre -ate, Fragment.onCreateView

vor START_OF_LOOP

onViewStateRestored

nach AT_FIRST, Fragment.onCre -ateView,Fragment.onActivity -Created

vor START_OF_LOOP

onStart

nach START_OF_LOOP, Activity. -onStart, Fragment.onViewState -Restored

vor MIDDLE_OF_LOOP

onResume

nach START_OF_LOOP, Fragment.-onStart, Activity.onPostCreate, Activity.onRestoreInstanceSta-te, Activity.onResume

vor MIDDLE_OF_LOOP

onPause

nach END_OF_LOOP, Activity.on-Pause

vor AFTER_LOOP

onStop

nach Fragment.onPause, Activity.on-Stop

onDestroyView

nach Fragment.onStop, Activity.on-Stop

onDestroy

nach Fragment.onDestroyView, Activity.onStop

onDetach

nach AT_LAST, Fragment.onDe-stroy

vor Activity.onDestroy

onActivityResult

nach MULTIPLE_TIMES_IN_-LOOP, Fragment.onPause, Activity.onPause

onConfigurationChanged

Direkt nach Activity.onConfigura-tionChanged

onContextItemSelected

Direkt nach Activity.onContext-ItemSelected

onCreateAnimator

Direkt nach Fragment.onResume

onCreateContextMenu

Direkt nach Activity.onCreateCon-textMenu

onCreateOptionsMenu

Direkt nach Activity.onCreateOp-tionsMenu

onDestroyOptionsMenu

nach END_OF_LOOP

onHiddenChanged

nach MIDDLE_OF_LOOP

onInflate

nach AT_FIRST, Application.onCre-ate, Activity.onCreate

vor Fragment.onAttach

onLowMemory

Direkt nach Activity.onLowMemo-ry

onOptionsItemSelected

Direkt nach Activity.onOptions-ItemSelected

onOptionsMenuClosed

Direkt nach Activity.onOptions-MenuClosed

onPrepareOptionsMenu

nach START_OF_LOOP, Fragment.-onCreateOptionsMenu, Fragment.onResume

vor Fragment.onOptionsItemSele-cted, Fragment.onOptionsMenu-Closed, AFTER_LOOP

onSaveInstanceState

nach Fragment.onStop, Fragment.on-DestroyView

vor Fragment.onDestroy, AT_LAST

onTrimMemory

Direkt vor Activity.onTrimMemory

onViewCreated

nach Fragment.onCreateView

vor Fragment.onActivityCreated, Fragment.onViewStateRestored

A.5 android.app.Service und BroadcastReceiver

com.ibm.wala.dalvik.util.androidEntryPoints.ServiceEP

Services sind Rechenaufgaben zugeordnet, welche im Hintergrund ausgeführt werden.

onCreate

nach AT_FIRST, ContentProvider.-onCreate

onStart

nach AT_FIRST, Srevice.onCreate, Service.onStartCommand

onStartCommand

nach AT_FIRST, Srevice.onCreate

vor

onBind

nach Service.onCreate, BEFORE_-LOOP

onUnbind

nach Service.onBind, END_OF_-LOOP

onRebind

nach Service.onBind, Service.onUn-bind

onTaskRemoved

nach Service.onUnbind, Service.on-Start, Service.onBind, Service.-onStartCommand, AT_LAST

vor Service.onDestroy

onConfigurationChanged

nach Service.onCreate

vor AFTER_LOOP

onLowMemory

nach END_OF_LOOP

vor AFTER_LOOP

onTrimMemory

nach END_OF_LOOP

vor AFTER_LOOP

onDestroy

nach Service.onUnbind, Service.on-Start, Service.onBind, Service.-onStartCommand, AT_LAST

onHandleIntent

nach Service.onCreate

vor Service.onStart

AbstractInputMethodService. onCreateInputMethodInterface

Direkt nach Service.onCreate

AbstractInputMethodService. onCreateInputMethodSession-Interface

nach AbstractInputMethodService.-onCreateInputMethodInterface

AbstractInputMethodService. onGenericMotionEvent

Direkt nach Activity.onGenericMo-tionEvent

AbstractInputMethodService. onTrackballEvent

Position Activity.onTrackballEvent

AccessibilityService. onAccessibilityEvent

nach AbstractInputMethodService.-onTrackballEvent

AccessibilityService.
onInterrupt

nach AccessibilityService.onAccessi-bilityEvent

DreamService.
onActionModeFinished

Position Activity.onActionModeFi-nished

DreamService.
onActionModeStarted

Position Activity.onActionMode-Started

DreamService.
onAttachedToWindow

Position Acitivity.onAttachedTo-Window

DreamService.
onContentChanged

Position Activity.onContentChan-ged

DreamService.
onCreatePanelMenu

Position Activity.onCreatePanel-Menu

DreamService.
onCreatePanelView

Position Activity.onCreatePanel-View

DreamService.
onDetachedFromWindow

Position Activity.onDetachedFrom-Window

DreamService.
onDreamingStarted

nach Service.onStart

DreamService.
onDreamingStopped

nach DreamService.onDreaming-Started, Service.onBind, Service.-onStartCommand

vor Service.onDestroy, Service.onUn-bind

DreamService.
onMenuItemSelected

Position Activity.onMenuItemSele-cted

DreamService.
onMenuOpened

Position Activity.onMenuOpened

DreamService.
onPanelClosed

Position Activity.onPanelClosed

DreamService.
onPreparePanel

Position Activity.onPreparePanel

DreamService.
onSearchRequested

Position Activity.onSearchReques-ted

DreamService.
onWindowAttributesChanged

Position Activity.onWindowAttri-butesChanged

DreamService.
onWindowFocusChanged

Position Activity.onWindowFocus-Changed

DreamService.
onWindowStartingActionMode

Position Activity.onWindowStart-ingActionMode

HostApduService.onDeactivated

Direkt vor Activity.onPause

MediaRouteProviderService.
onCreateMediaRouteProvider

Position Service.onCreate

**NotificationListenerService.
onNotificationPosted**

Position MULTIPLE_TIMES_IN
_LOOP

**NotificationListenerService.
onNotificationRemoved**

nach NotificationListenerService.on
-NotificationPosted

**PrintService.
onConnected**

nach Service.onStart

**PrintService.
onCreatePrinterDiscoverySes
-sion**

nach Service.onStart

vor PrintService.onConnected

**PrintService.
onDisconnected**

nach PrintService.onConnected

vor Service.onDestroy

**PrintService.
onPrintJobQueued**

nach PrintService.onConnected

vor PrintService.onDisconnected

**PrintService.
onRequestCancelPrintJob**

nach PrintService.onPrintJobQue
-ued

vor PrintService.onDisconnected

**RecognitionService.
onCancel**

nach MULTIPLE_TIMES_IN_
-LOOP

vor END_OF_LOOP

**RecognitionService.
onStartListening**

nach MULTIPLE_TIMES_IN_
-LOOP

vor RecognitionService.onCancel

**RecognitionService.
onStopListening**

nach RecognitionService.onCancel

vor END_OF_LOOP

**RemoteViewsService
.onGetViewFactory**

nach Service.onStart

**SettingInjectorService.
onGetEnabled**

nach Service.onStart

**SettingInjectorService.
onGetSummary**

nach Service.onStart

**TextToSpeechService.
onGetFeaturesForLanguage**

nach Service.onStart

**TextToSpeechService.
onGetLanguage**

Direkt vor TextToSpeechSer
-vice.onGetFeaturesForLanguage

**TextToSpeechService.
onIsLanguageAvailable**

Direkt vor TextToSpeechService.
-onLoadLanguage

**TextToSpeechService.
onLoadLanguage**

Direkt vor TextToSpeechService.
-onGetLanguage

**TextToSpeechService.
onStop**

Direkt vor Activity.onStop

**TextToSpeechService.
onSynthesizeText**

nach TextToSpeechService.onGet
-Language, TextToSpeechService.
-onLoadLanguage, MULTIPLE_
-TIMES_IN_LOOP

VpnService.onRevoke

nach END_OF_LOOP

vor Service.onDestroy

WallpaperService.onCreateEngine

nach Service.onCreate

vor Service.onStart

InputMethodService. onAppPrivateCommand

Position MULTIPLE_TIMES_IN _LOOP

InputMethodService. onBindInput

nach Activity.onResume

InputMethodService. onComputeInsets

nach Service.onStart

InputMethodService. onConfigureWindow

nach InputMethodService.onCompu -teInsets

InputMethodService. onCreateCandidatesView

nach Service.onStart

vor InputMethodService.onCompu -teInsets

InputMethodService. onCreateExtractTextView

nach InputMethodService. onCre -ateCandidatesView

InputMethodService. onCreateInputView

Position InputMethodService. on -CreateExtractTextView

InputMethodService. onDisplayCompletions

Position MULTIPLE_TIMES_IN _LOOP

InputMethodService. onEvaluateFullscreenMode

Position MULTIPLE_TIMES_IN _LOOP

InputMethodService. onEvaluateInputViewShown

Position MULTIPLE_TIMES_IN _LOOP

InputMethodService. onExtractTextContextMenu -Item

Position MULTIPLE_TIMES_IN _LOOP

InputMethodService. onExtractedCursorMovement

Position MULTIPLE_TIMES_IN _LOOP

InputMethodService. onExtractedSelectionChanged

Direkt nach InputMethodService. -onExtractedCursorMovement

InputMethodService. onExtractedTextClicked

Direkt nach InputMethodService. -onExtractedCursorMovement

InputMethodService. onExtractingInputChanged

nach InputMethodService.onExtra -ctedTextClicked, InputMethod -Service.onExtractedSelection -Changed, MULTIPLE_TIM -ES_IN_LOOP

InputMethodService. onFinishCandidatesView

nach InputMethodService.onEx -tractingInputChanged, In -putMethodService.onFinishIn -put, InputMethodService.on -StartCandidatesView

InputMethodService. onFinishInput

nach InputMethodService.onExtrac-tingInputChanged

InputMethodService.onFinishInputView

nach InputMethodService.onFinish-Input

InputMethodService.onInitializeInterface

Position MULTIPLE_TIMES_IN_LOOP

InputMethodService.onKeyDown

Direkt vor Activity.onKeyDown

InputMethodService.onKeyLongPress

Direkt vor Activity.onKeyLong-Press

InputMethodService.onKeyMultiple

Direkt vor Activity.onKeyMultiple

InputMethodService.onKeyUp

Direkt vor Activity.onKeyUp

InputMethodService.onShowInputRequested

Position MULTIPLE_TIMES_IN_LOOP

InputMethodService.onStartCandidatesView

Position InputMethodService.on-CreateExtractTextView

InputMethodService.onStartInput

nach AbstractInputMethodService.-onCreateInputMethodSession-Interface, Activity.onResume

InputMethodService.onStartInputView

Position InputMethodService.on-StartInput

InputMethodService.onUnbindInput

Position Service.onUnbind

InputMethodService.onUpdateCursor

Direkt vor AbstractInputMethod-Service.onGenericMotionEvent

InputMethodService.onUpdateExtractedText

Direkt nach InputMethodService.-onExtractedTextClicked

InputMethodService.onUpdateExtractingViews

nach InputMethodService.onExtrac-tingInputChanged

InputMethodService.onUpdateExtractingVisibility

nach InputMethodService.onUpdate-ExtractingViews

InputMethodService.onUpdateSelection

nach InputMethodService.onExtrac-tedSelectionChanged

InputMethodService.onViewClicked

nach AbstractInputMethodSer-vice.onGenericMotionEvent, AbstractInputMethodSer-vice.onTrackball Event, Input-MethodService.onKey Up

InputMethodService.onWindowHidden

nach InputMethodService.onWin-dowShown

vor Service.onDestroy, Activity.on-Pause

InputMethodService.onWindowShown

nach InputMethodService.onConfi-gureWindow, InputMethodSer-vice.onCreateCandidatesView

BroadcastReceiver.onReceive

nach AT_FIRST, ContentProvider.-onCreate

ANHANG B

Weitere Klassen für Einsprungspunkte

com.ibm.wala.dalvik.util.AndroidEntryPointLocator

Die folgenden Funktionen verursachen keinen Start einer Android Anwendung. Bei der Umsetzung des Modells wurden sie jedoch als wichtig genug erachtet, um für sie ebenfalls Platzierungsinformationen zu hinterlegen.

Diese Informationen werden lediglich dann berücksichtigt, wenn bei der Suche nach Einsprungspunkten das Flag **INCLUDE_CALLBACKS** gesetzt ist.

Loader

com.ibm.wala.dalvik.util.androidEntryPoints.LoaderCB

Die *Loader*-Klasse kann herangezogen werden, um asynchron Daten aus dem Dateisystem zu laden.

onCreateLoader

Position AT_FIRST

onLoadFinished

nach onCreateLoader

onLoaderReset

nach onCreateLoader

Location

com.ibm.wala.dalvik.util.androidEntryPoints.LocationEP

Im Folgenden sind Klassen aufgeführt, welche im Zusammenhang mit der Änderung der physikalischen Position des Gerätes stehen.

LocationListener.onLocationChanged

Position MIDDLE_OF_LOOP

LocationListener.onProviderDisabled

Position MIDDLE_OF_LOOP

**LocationListener.
onProviderEnabled**

Position MIDDLE_OF_LOOP

**LocationListener.
onStatusChanged**

Position MIDDLE_OF_LOOP

**GpsStatus$Listener.
onGpsStatusChanged**

Position MIDDLE_OF_LOOP

**GpsStatus$NmeaListener.
onNmeaReceived**

Position MIDDLE_OF_LOOP

ANHANG C

Standartwerte des Instanziierungsverhaltens

🖮 com.ibm.wala.dalvik.ipa.callgraph.androidModel.parameters.DefaultInstantiationBehavior

Im Verlauf der Generierung des Lebenszyklusmodelles werden Instanziierungsverhalten für auftretende Parameter anhand ihrer Typen vorgeschlagen. In der folgenden Tabelle sind die Randbedingungen für die Vorschläge aufgeführt.

		Instanziierungsverhalten
Paket	Ljava/lang	CREATE
Typ	Ljava/lang/Object	CREATE
Typ	[Ljava/lang/Object	CREATE
Typ	Landroid/os/Bundle	REUSE
Typ	Landroid/app/Activity	REUSE
Typ	Landroid/app/Service	REUSE
Typ	Landroid/view/Menu	CREATE
Typ	Landroid/view/ContextMenu	CREATE
Typ	Landroid/view/MenuItem	CREATE
Typ	Landroid/view/ActionMode	CREATE
Typ	Landroid/util/AttributeSet	CREATE
Typ	Landroid/view/ActionMode$Callback	CREATE
Typ	Landroid/view/KeyEvent	CREATE
Paket	Landroid/support/v4/view	REUSE

Die für Typen angegebenen Werte werden jeweils für von ihnen abgeleitete Typen übernommen, sofern auf dem Ableitungspfad keine spätere feste Angabe vorhanden ist.

ANHANG D

Funktionen zum Start von Intents

🖮 com.ibm.wala.dalvik.ipa.callgraph.propagation.cfa.IntentStarters

Weitere Komponenten werden in Android mittels eines *Intent* gestartet, welches an eine artspezifische Start-Funktion übergeben wird. Diese Funktionen seien im Folgenden aufgelistet:

a.c.ContextWrapper.bindService

Zielart Service

Eingenschaft Unbekanntes Intent, Explizit

a.c.ContextWrapper.sendBroadcast

Zielart BroadcastReceiver

Eingenschaft Broadcast

a.c.ContextWrapper.sendBroadcastAsUser

Zielart BroadcastReceiver

Eingenschaft Broadcast

a.c.ContextWrapper.sendOrderedBroadcast

Zielart BroadcastReceiver

Eingenschaft Broadcast

a.c.ContextWrapper.sendOrderedBroadcastAsUser

Zielart BroadcastReceiver

Eingenschaft Broadcast

a.c.ContextWrapper.sendStickyBroadcast

Zielart BroadcastReceiver

Eingenschaft Broadcast

a.c.ContextWrapper.sendStickyBroadcastAsUser

Zielart BroadcastReceiver

Eingenschaft Broadcast

a.c.ContextWrapper.sendStickyOrderedBroadcast

Zielart BroadcastReceiver

Eingenschaft Broadcast

a.c.ContextWrapper.sendStickyOrderedBroadcastAs-User

Zielart BroadcastReceiver

Eingenschaft Broadcast

a.c.ContextWrapper.startActivities

Zielart Activity

Eingenschaft Unbekanntes Intent

a.c.ContextWrapper.startActivity

Zielart Activity

Eingenschaft Unbekanntes Intent

a.c.ContextWrapper.startIntentSender

Zielart Activity

Eingenschaft Unbekanntes Intent, ”Quench Permissions”

a.c.ContextWrapper.startService

Zielart Service

Eingenschaft Unbekanntes Intent

a.c.Context.bindService

Zielart Service

Eingenschaft Unbekanntes Intent, Explizit

a.c.Context.sendBroadcast

Zielart BroadcastReceiver

Eingenschaft Broadcast

a.c.Context.sendBroadcastAsUser

Zielart BroadcastReceiver

Eingenschaft Broadcast

a.c.Context.sendOrderedBroadcast

Zielart BroadcastReceiver

Eingenschaft Broadcast

a.c.Context.sendOrderedBroadcastAsUser

Zielart BroadcastReceiver

Eingenschaft Broadcast

a.c.Context.sendStickyBroadcast

Zielart BroadcastReceiver

Eingenschaft Broadcast

a.c.Context.sendStickyBroadcastAsUser

Zielart BroadcastReceiver

Eingenschaft Broadcast

a.c.Context.sendStickyOrderedBroadcast

Zielart BroadcastReceiver

Eingenschaft Broadcast

a.c.Context.sendStickyOrderedBroadcastAs-User

Zielart BroadcastReceiver

Eingenschaft Broadcast

a.c.Context.startActivities

Zielart Activity

Eingenschaft Unbekanntes Intent

a.c.Context.startActivity

Zielart Activity

Eingenschaft Unbekanntes Intent

a.c.Context.startIntentSender

Zielart Activity

Eingenschaft Unbekanntes Intent, ”Quench Permissions”

a.c.Context.startService

Zielart Service

Eingenschaft Unbekanntes Intent

a.a.Activity.startActivities

Zielart Activity

Eingenschaft Unbekanntes Intent

a.a.Activity.startActivity

Zielart Activity

Eingenschaft Unbekanntes Intent

a.a.Activity.startActivityForResult

Zielart Activity

Eingenschaft Unbekanntes Intent, Result-CallBack

a.a.Activity.startActivityFromChild

Zielart Activity

Eingenschaft Unbekanntes Intent

a.a.Activity.startActivityFromFragment

Zielart Activity

Eingenschaft Unbekanntes Intent

a.a.Activity.startActivityIfNeeded

Zielart Activity

Eingenschaft Unbekanntes Intent

a.a.Activity.startIntentSender

Zielart Activity

Eingenschaft Unbekanntes Intent, "Quench Permissions"

a.a.Activity.startIntentSenderForResult

Zielart Activity

Eingenschaft Unbekanntes Intent, "Quench Permissions", Result-CallBack

a.a.Activity.startIntentSenderFromChild

Zielart Activity

Eingenschaft Unbekanntes Intent, "Quench Permissions", Result-CallBack

a.a.Activity.startNextMatchingActivity

Zielart Activity

Eingenschaft Unbekanntes Intent

a.a.Activity.startActivity

Zielart Activity

Eingenschaft Unbekanntes Intent

a.s.v4.app.FragmentActivity.startActivityForResult

Zielart Activity

Eingenschaft Unbekanntes Intent, Result-CallBack

a.s.v4.app.FragmentActivity.startActivityFromFragment

Zielart Activity

Eingenschaft Unbekanntes Intent

ANHANG E

JoDroids Einsprungspunktdatei

🖮 edu.kit.joana.wala.jodroid.entrypointsFile.Tags

JoDroid hinterlegt die gefundenen Einsprungspunkte in einer *XML*-Datei mit der Endung *ntrP*. Im Folgenden seien die in dieser Datei erlaubten Tags aufgeführt.

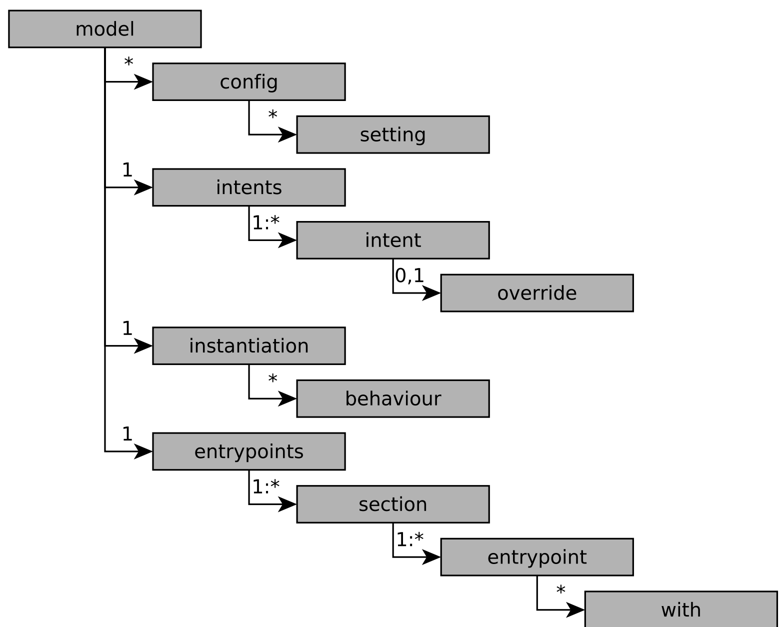


Abbildung E.1: *Dokumentstruktur einer ntrP-Datei*

Der Wurzelkonten der Datei wird durch ein *model*-Tag gebildet. Unter diesem

können sich beliebig viele *config*-Knoten befinden. Weiterhin muss je genau ein Knoten der Typen *intents*, *instantiation* und *entrypoints* unter diesem Knoten vorhanden sein.

Das Config-Tag

Einträge unter diesem Tag dienen derzeit nur der Information, Änderungen werden nicht eingelesen. Das Tag sollte ein Attribut "*of*" aufweisen, welches die Klasse referenziert, aus der die Konfiguration geladen wurde.

Es ist mit dem *setting*-Tag lediglich ein Typ von Unterknoten erlaubt. Von diesen können beliebig viele vorhanden sein. Jedes *setting*-Tag muss ein Attribut *name*, zur Referenzierung des Feldes, und ein Attribut *value* aufweisen.

Das Intents-Tag

Dieses Tag dient zur Aufnahme der im Folgenden behandelten *intent*-Tags. Von diesen sollte mindestens ein Knoten existieren. Das *intents*-Tag selbst hat keine Attribute.

Das Intent-Tag

Ein Tag diesen Typs dient zur Aufnahme von Informationen, wie Start-Funktionen (siehe Anhang D) durch den *IntentContextInterpreter* (siehe Unterabschnitt 5.2.4) aufgelöst werden sollen. Folgende Attribute können in einem Intent-Tag definiert werden:

name Hierbei handelt es sich um einen String in Java-Bytecode-Notation, durch welchen entweder direkt die Zielklasse oder ein abstrakter Name zu dem Intent angegeben werden kann. Dieses Feld ist somit mit dem *action*-Tag einer *AndroidManifest.xml*-Datei vergleichbar.

resolves Steuert die erstellte Wrapper-Funktion (siehe Unterabschnitt 4.6.5). Erlaubte Werte lauten auf:

INTERNAL_TARGET Diese Einstellung kann gewählt werden, wenn das Ziel des Intents im *AnalysisScope* liegt. Der *IntentContextInterpreter* wird hier ein Modell generieren, welches Lediglich Einsprungspunkte der Zielklasse enthält.

Wenn sich unter dem *Intent*-Tag kein *override*-Tag befindet, so muss der in *name* angegebene Wert auf den Klassennamen lauten, zu dem der Intent aufgelöst werden soll.

EXTERNAL_TARGET Diese Einstellung weist darauf hin, dass das Ziel des Intents durch eine externe Applikation zur Verfügung gestellt wird. Im Zuge der Modellierung wird der Aufruf zu einer artspezifischen Platzhalterfunktion gemäß Unterabschnitt 3.7.1 aufgelöst. In der späteren IFC-Analyse sollten diese Funktionen als "unsichere Senke" markiert werden.

STANDARD_ACTION Diese Einstellung wird für Intents verwendet, die auf einen von Android vorgegebenen Namen lauten. In aller Regel wird man sie über ein *Override* (siehe unten) auf ein internes Ziel umleiten.

Ist kein *Override* vorhanden wird ein der Intent derzeit wie ein Intent mit Angabe *UNKNOWN_ TARGET* behandelt.

UNKNOWN_TARGET Wenn das Ziel des Intents unbekannt ist, so wird ein zu der *Start-Funktion* passendes *Mini-Modell* (wie in Unterabschnitt 4.6.4 beschrieben) angesprungen. Anschließend wird eine weitere Funktion wie bei *EXTERNAL_ TARGET* angesprungen.

Derzeit kann nur ein Ziel je Intent aufgelöst werden, auch implizite Intents sind aktuell nicht möglich. Ist dies gewünscht, so muss auf das Verhalten von *UNKNOWN_ TARGET* zurückgefallen werden.

Auch können derzeit keine URI-Informationen zu den Intents hinterlegt werden.

Das Override-Tag

Über dieses Tag kann angegeben werden, dass ein Intent zu einem anderen Intent aufgelöst werden soll. Das Ziel des *Overrides* kann selbst wieder mit einem *Override* versehen sein; in diesem Fall wird die "Kette" weiter verfolgt, bis ein Ziel gefunden wurde. Ein Schutz vor der Konstruktion einer Endlosschleife existiert hier nicht.

Sinnvoll ist die Verwendung dieses Tags vor allem zur Umleitung von Intents, welche durch das System definiert werden (mit dem *resolves*-Wert von *STANDARD_ ACTION*). Auch für *INTERNAL_ TARGET*-Intents kann es verwendet werden.

Für die Überschreibung von Intents mit den Einstellungen *EXTERNAL_ TARGET* und *UNKNOWN_ TARGET* ist das Verhalten undefiniert.

Als Ziel der Umleitung sind allerdings wieder alle *resolves*-Werte denkbar.

Die Attribute zu dem *overrides*-Tag verhalten sich analog zu denen des *intent*-Tags. Der Wert von *resolves* sollte allerdings mit dem *resolves*-Wert des Ziels übereinstimmen.

Derzeit ist lediglich ein *Override* je Intent vorgesehen. Soll die Umleitung auf mehrere Ziele erfolgen sollte somit ein *UNKNOWN_ TARGET*-Ziel verwendet werden.

Das Instantiation-Tag

Dieses Tag dient zur Aufnahme von *behaviour*-Tags. In einem Dokument sollte es jeweils genau einmal existieren. Dieses Tag besitzt ein einzelnes Attribut:

default Kann zu einem Typ weder über das Paket noch über Ableitung ein Verhalten festgestellt werden, so wird der Wert von *default* zurückgegeben. Mögliche Werte sind im *of*-Attribut des *behaviour*-Tags beschrieben.

Da in Android alle Referenztypen von *Object* erben sollte dieser Wert (zumindest wenn eine *DefaultInstantiationBehavior* oder *LoadedInstantiationBehavior*, welche Vererbung beachten, zum Einsatz kommt) von untergeordneter Rolle sein.

Das Behaviour-Tag
Über *behaviour*-Tags, welches als Kind-Element des *instantiation*-Tags auftauchen, lässt sich das Instanziierungsverhalten zu einem Typ gemäß Abschnitt 3.2 steuern.

Das Behaviour-Tag hat die folgenden Attribute:

type Beinhaltet den Typ, zu dem das Verhalten festgelegt werden soll in Java-Bytecode-Notation. Alternativ kann das Attribut *package* gewählt werden, um das Verhalten für ein komplettes Paket festzulegen.

package Dient zu Verhaltensfestlegung für ein komplettes Paket. Befindet sich in diesem Paket ein Typ, für den ein Verhaltenseintrag mit einem *type*-Attribut vorliegt, so wird dieses Verhalten bevorzugt.

of Gibt das Verhalten gemäß Abschnitt 3.2 an. Mögliche Werte lauten derzeit auf:

- **CREATE** Bewirkt, dass bei jedem Auftauchen einer Variable diesen Typs eine neue Instanz erstellt wird.
- **REUSE** Bewirkt, dass eine Variable diesen Typs zu einem Parameter des Modells wird. Diese Instanz wird für alle Vorkommen des Typs herangezogen. Hat ein Einsprungspunkt einen zuweisungskompatiblen Rückgabewert, so wird dieser über eine Φ-Funktion in die verwendete Instanz eingemischt.

exactness Dieser Wert wird durch eine *DefualtInstantiationBehavior* generiert. Er gibt an, auf welchem Weg der *of*-Wert zu einem Typ gefunden wurde. Der Wert von *exactness* hat keinerlei Auswirkung auf das spätere Modell. Mögliche Werte lauten auf:

- **EXACT** Dieser Wert besagt, dass zu dem gegebenen Typ ein Verhalten explizit in der *IInstantiationBehavior* hinterlegt wurde.
- **INHERITED** Der *of*-Wert wurde durch Nachschlagen der Vererbungsbeziehungen in der Klassenhierarchie von einer Oberklasse übernommen.
- **PACKAGE** Die *IInstantiationBehavior* enthält einen Paket-Eintrag, der auf diesen Typ zutrifft.
- **PREFIX** Der Wert wurde durch einen Text-Vergleich der Anfänge der Typnamen generiert.

to *(derzeit nicht berücksichtigt)* Bietet die Möglichkeit der Angabe eines anderen Verhaltenswertes abhängig von dem Typ der Klasse, auf deren Einsprungspunkt aufgerufen werden soll.

call *(derzeit nicht berücksichtigt)* Bietet die Möglichkeit der Angabe eines anderen Verhaltenswertes abhängig von dem Einsprungspunkt, der aufgerufen werden soll.

Die Implementierung der Auswertung der Parameter *to* und *call* ist vorbereitet, jedoch no nicht komplett umgesetzt.

Das Entrypoints-Tag
Dieses Tag hat keinerlei Attribute. Es dient zur Aufnahme ein oder mehrere *Section*-Tags.

Das Section-Tag
Durch dieses Tag wird eine *Sektion* innerhalb des Modells zur Aufnahme von Einsprungspunkten gemäß Abschnitt 3.3 definiert. Dieses Tag hat das folgende Attribut:

name Der Name des *Labels* (siehe Abschnitt 3.3), welches die *Sektion* einleitet. Mögliche Werte der in *AndroidEntryPoint.ExecutionOrder* definierten Klasse sind:

- **AT_FIRST** Diese Einsprungspunkte werden bei Start der Applikation ausgeführt.
- **BEFORE_LOOP** Eine Hilfsposition ohne direkte Verhaltensimplikation.
- **START_OF_LOOP** Hier finden sich hauptsächlich Einsprungspunkte, die bei Sichtbar- Werden der Applikation aufgerufen werden.
- **MIDDLE_OF_LOOP** Eine Hilfsposition ohne direkte Verhaltensimplikation.
- **MULTIPLE_TIMES_IN_LOOP** Einsprungspunkte zu Statusänderungen und für die Benutzerinteraktion befinden sich meist in dieser Sektion.
- **END_OF_LOOP** Funktionen, die Aufgerufen werden wenn die Anwendung sich noch im Vordergrund befindet, jedoch eher dem Ende des Lebenszyklus angehörig sind, finden sich in dieser Sektion.
- **AFTER_LOOP** Unter anderem finden sich hier Funktionen, die dem in-den-Hintergrund- Treten einer Komponente zugeordnet sind.
- **AT_LAST** Beinhaltet größtenteils Funktionen, die kurz bevor die Anwendung aus dem Speicher entfernt wird aufgerufen werden.]

Anders als die Namen der Labels vermuten lassen befinden sich an diesen Stellen nicht zwangsläufig Schleifen! Ob dies der Fall ist hängt von der Verwendeten Modellstruktur (siehe Abschnitt 3.4) ab.

Welche Einsprungspunkte in der Regel in welchen Sektionen liegen lässt sich anhand von Anhang A ablesen.

Das Entrypoint-Tag
In einem solchen Tag wird jeweils ein Sprungziel angegeben. Sie können als Kindelement eines Section-Tags auftauchen.

Einsprungspunkte werden in der Modellierung in der Reihenfolge aufgerufen, in der sie in der Datei angegeben sind.

type Diese optionale Eigenschaft dient der Beschreibung des Komponententyps der implementierenden Klasse. Die Werte haben keinen Einfluss auf die Modellierung und sollen lediglich das Verständnis der jeweiligen Zugehörigkeit der Einsprungspunkte erleichtern.

Mögliche Werte lauten in der aktuellen Implementierung zunächst auf: *ACTIVITY*, *SERVICE*, *APPLICATION*, *PROVIDER* und *BROADCAST_RECEIVER*. Weitere finden sich in: *ABSTRACT_INPUT_METHOD_SERVICE*, *ACCESSIBILITY_SERVICE*, *BINDER*, *CONTEXT*, *DREAM_SERVICE*, *FRAGMENT*, *GPS_LISTENER*, *GPS_ NMEA_LISTENER*, *HOST_APDU_SER-VICE*, *HTTP*, *INPUT_METHOD_SERVICE*, *INTENT_SERVICE*, *LOADER_CB*, *LOCATION_LISTENER*, *LOCATION_MGR*, *MEDIA_ROUTE_-PROVIDER_ SERVICE*, *NOTIFICATION_LISTENER_SERVICE*, *OFF_-HOST_APDU_ SERVICE*, *PRINT_SERVICE*, *RECOGNITION_SERVICE*, *REMOTE_VIEWS_SERVICE*, *RESOLVER*, *SETTING_INJECTOR_SER-VICE*, *SMS*, *SMS_GSM*, *SPELL_CHECKER_SERVICE*, *TELEPHONY*, *TEXT_TO_ SPEECH_SERVICE*, *VPN_SERVICE* und *WALLPAPER_SER-VICE*. Der letztmögliche Wert *UNKNOWN* findet Verwendung, wenn ein Einsprungspunkt anhand einer Heuristik selektiert wurde.

call In dieser Pflicht-Eigenschaft wird der *Selector* (bestehend aus Paket, Klasse, Name und Signatur) der Funktion, die an dieser Position im Modell aufgerufen werden soll angegeben.

Das With-Tag
Dieses mögliche Kind-Tag des *Entrypoint-Tags* findet Verwendung, wenn ein Einsprungspunkt einer Oberklasse aufgerufen werden soll. Solche Tags kommen demnach nur dann in der Datei vor, wenn die Einsprungspunkte mit aktivierter Option **WITH_SUPER** gesucht wurden.

Durch einen *With-Tag* wird angegeben, welche konkreten Typen für einen Parameter herangezogen werden sollen.

param Über dieses Attribut wird angegeben, für welches Argument der Funktion konkrete Typen angegeben werden sollen. Mögliche Werte lauten auf

Text 'this' Gibt an, dass für den impliziten *this-Pointer* ein anderer Typ angegeben werden soll.

Eine Nummer Gibt die Stelle in der Signatur der Funktion an, für die ein anderer konkreter Typ gewählt werden soll. Die Nummerierung beginnt dabei bei eins. Ein möglicher impliziter this-Pointer wird nicht mitgezählt.

type Der konkrete Typ in Java-Bytecode-Syntax, der Verwendung finden soll. Offenbar muss der angegebene Typ zuweisungskompatibel zu dem jeweiligen Typ der Signatur der Funktion sein.

Für einen Einsprungspunkt mit *With-Tag* werden mehrere einzelne Aufrufe im Modell generiert[1]. Werden für mehrere Parameterstellen konkrete Typen angegeben, so wird das Kartesische Produkt gebildet.

[1]Die Alternative bestünde darin Instanzen über ein Φ-Funktion zu kombinieren

ANHANG F

Einstellmöglichkeiten des Modells

🖮 com.ibm.wala.dalvik.util.AndroidEntryPointManager

Dieses Kapitel beschreibt alle Modellbezogenen Einstellungen. Die Beschreibung findet sich auch in der *JavaDoc*-Dokumentation wieder.

Es ist zu beachten, dass die angenommenen Standartwerte nicht grundsätzlich die konservativste Einstellung wiederspiegeln.

setDoBootSequence (boolean)

Ist diese Einstellung aktiv wird zusätzlicher Code zur Erzeugung eines Android-Kontexts in das Modell eingefügt. Dieser Kontext wird hauptsächlich bei der Kommunikation über Applikationsgrenzen hinweg interessant.

Ist diese Einstellung nicht aktiv, so können einige Aufrufe – beispielsweise die Abfrage des *ActivityManagers* nicht richtig aufgelöst werden.

Ist diese Einstellung aktiv, so müssen die verwendeten Stubs die Funktion *Activity.attach* beinhalten.

Durch Deaktivierung der Einstellung lässt sich unter Umständen ein kleinerer Systemabhängigkeitsgraph generieren wobei die negativen Auswirkungen gering sind.

Standardwert: aktiviert

setDoFlatComponents (boolean)

Wenn diese Einstellung aktiv ist, so wird für jede Komponente eine Instanz in der *AndroidModelClass* generiert. Immer wenn eine neue Instanz einer Komponente durch das Modell generiert werden soll wird diese globale Instanz genutzt.

Android hat eine selten verwendete Flat-Einstellung für den Aufruf von Komponenten. Ist diese in einer Applikation vorhanden, so sollte auch diese Einstellung aktiviert werden. Das Ergebnis ist danach konservativer.

Standardwert: deaktiviert

setInstantiationBehavior (IInstantiationBehavior)

Das instanziierungsverhalten steuert, wie eine Instanz von jeweiligen Typen erstellt werden soll. Sie ist in Abschnitt 4.2 näher beschrieben.

Derzeit implementierte Instanziierungsverhalten sind:

DefaultInstantiationBehavior Verfügt über fest codierte Grundeinstellungen – siehe Anhang C.

LoadedInstantiationBehavior Ein leeres Verhalten, das zunächst befüllt werden muss.

Standartwert: *DefaultInstantiationBehavior*.

setModelBehavior (Class abstractAndroidModel)

Steuert Sonderbehandlungen (beispielsweise Schleifen), die in das Modell eingefügt werden.

Derzeit implementierte Modellstrukturen sind (siehe Unterabschnitt 4.5.5):

SequentialAndroidModel Fügt keinerlei Sonderbehandlungen ein.

LoopAndroidModel Komponenten werden einmalig gestartet. Es existiert eine Schleife für das Sichtbar und Unsichtbar werden von Activitys, eine weitere für Benutzerinteraktion.

LoopKillAndroidModel Hat eine weitere Schleife für den Neustart einer Komponente aufgrund von Speicherknappheit.

Standartwert: *LoopAndroidModel*

setAllowIntentRerouting(boolean)

Steuert das Verhalten bei Auftritt von Funktionen, durch die das Ziel eines Intents verändert wird (beispielsweise *Intent.setAction*, *Intent.setClass*, *Intent.fillIn*).

Ist diese Einstellung deaktiviert, so wird das Intent grundsätzlich als unauflösbar markiert, woraufhin bei dessen Aufruf konservativ approximiert wird.

Bei aktiviertem Intentrerouting wird die Einstellung des Aufrufes einer solchen Funktion übernommen, es sei denn:

- Das Intent war bereits auflösbar. Es wird dann unauflösbar.
- Der für das Intent verwendete Konstruktor deutet auf ein explizites Intent hin, das jedoch von WALA nicht auflösbar war. Es verbleibt unauflösbar.
- Eine andere *action* wird auf einem expliziten Intent gesetzt. Die Änderung wird dann ignoriert.
- Es ist der zweite Aufruf einer solchen Funktion auf dem selben impliziten Intent. Es wird dann unauflösbar.

Standartwert: aktiviert

setOverride (Intent, Intent)

Steuert zu welchem Ziel ein Intent aufgelöst werden soll und damit, welches Modell bei Start eines solchen Intents aufgerufen werden soll. Das Ziel selbst ist offensichtlich wieder als Intent angegeben.

Für ein Intent existieren die Einstellungen:

IntentType.INTERNAL_TARGET Das Intent wird zu der in *action* angegebenen (in der Analyse auflösbaren) Klasse aufgelöst.

IntentType.EXTERNAL_TARGET Es wird ein *ExternalTargetModel* erzeugt, welches auf alle dem Intent beigefügten Parameter zugreift und daraus einen Rückgabewert erzeugt.

IntentType.UNKNOWN_TARGET Es werden alle Komponenten passenden Typs in der Applikation (beispielsweise alle Activities) gewählt. Anschließend wird ein *ExternalTargetModel* angesprungen.

IntentType.STANDARD_ACTION Wird für Actions verwendet, die durch Android vordefiniert sind. Ist keine weitere Überschreibung vorhanden wird mit ihnen wie mit *UNKNOWN_ TARGET* verfahren.

IntentType.BROADCAST Mit Broadcasts wird wie mit *UNKNOWN_ -TARGET* verfahren.

IntentType.SYSTEM_SERVICE Diese Einstellung wird lediglich intern für das Vorkommen des Aufrufes *getSystemService(String)* verwendet: Das *SystemServiceModel* erzeugt lediglich eine Instanz des passenden Service und gibt diese zurück.

IntentType.IGNORE Der Aufruf wird ignoriert. Diese Einstellung ist beispielsweise für *actions* sinnvoll, die durch die Umgebung aufgerufen werden und die keinen Schaden anrichten können – beispielsweise *DIAL*.

Die Funktion *setOverride* übernimmt Einstellungen lediglich dann, wenn sie genauer als die vorherige Einstellung sind.

Derzeit ist lediglich ein Ziel pro Intent vorgesehen. Sind mehrere Ziele gewünscht, so muss ein Intent mit der Einstellung *UNKNOWN_ TARGET* gewählt werden.

setOverrideForce (Intent, Intent)

Ist analog zu der vorherigen Einstellung erlaubt aber eine Veringerung der Präzision.

setPackage (String)

Setzt das Paket der analysierten Anwendung. Dies hat Auswirkungen auf die Auflösung von Intents. Wird eine *AndroidManifest.xml*-Datei gelesen, so wird das Paket automatisch gesetzt.

Wurde kein Paket gesetzt, so wird anhand der Einsprungspunkte geraten.

ANHANG G

Analyseeinstellungen von Insecure Bank

Hier seien die Einstellungen der Analyse von *Insecure Bank* in Unterabschnitt 6.4.3 in vollem Umfang dargestellt.

Einstellungen zur Generierung des Android-Modells

doBootSequence	false
flatComponents	false
instantiationBehavior	DefaultInstantiationBehavior
abstractAndroidModel	LoopAndroidModel
allowIntentRerouting	true

Einstellungen zur Generierung des SDG

accessPath	false
computeInterference	false
computeSummary	true
debugAccessPath	false
debugCallGraphDotOutput	false
debugManyGraphsDotOutput	false
debugStaticInitializers	false
keepPhiNodes	true
localKillingDefs	true
noBasePointerDependency	true
prunecg	DEFAULT_PRUNE_CG
additionalContextInterpreter	IntentContextInterpreter-Instanz
additionalContextSelector	IntentContextSelector-Instanz
debugAccessPathOutputDir	
entry	intentInsecureBankActivityEncap ()
exceptions	IGNORE_ALL
ext	Instanz einer ”leeren” Klasse

fieldPropagation	OBJ_GRAPH
ignoreStaticFields	Main.IGNORE_STATIC_FIELDS
immutableNoOut	Main.IMMUTABLE_NO_OUT
immutableStubs	Main.IMMUTABLE_STUBS
methodTargetSelector	
objSensFilter	
pruningPolicy	
pts	OBJECT_SENSITIVE
sideEffects	
staticInitializers	SIMPLE

Klassen Index

Der folgende Index führt Klassen auf, die in diesem Dokument referenziert werden. Für Informationen über alle Klassen, die im Umfang dieser Arbeit geschrieben wurde bietet sich eine Suche in der *JavaDoc*-Dokumentation an.

com.ibm.wala.cfg.InducedCFG ... 77

com.ibm.wala.classLoader

- .IMethod ... 33
- .JavaLanguage.JavaInstructionFactory ... 34

com.ibm.wala.dalvik.ipa.callgraph.androidModel

- .parameters
  - .AndroidModelParameterManager ... 52
  - .DefaultInstantiationBehavior ... 50, 133
  - .IInstantiationBehavior ... 50
  - .Instantiator ... 51, 75
  - .LoadedInstantiationBehavior ... 50
- .stubs
  - .AndroidBoot ... 57
  - .AndroidStartComponentTool ... 67
  - .Overrides ... 63

com.ibm.wala.dalvik.ipa.callgraph.impl

- .AbstractAndroidModel ... 59
- .AndroidEntryPoint ... 77
  - .ExecutionOrder ... 52
  - .IExecutionOrder ... 52
- .DexFakeRootMethod.ReuseParameters ... 57

com.ibm.wala.dalvik.ipa.callgraph.propagation.cfa

- .Intent ... 65
- .IntentContext ... 64, 78
- .IntentContextInterpreter ... 64, 78
- .IntentContextSelector ... 64, 78
- .IntentStarters ... 63, 64, 135

com.ibm.wala.dalvik.ipa.summaries

- .SummarizedMethodWithNames ... 75

.VolatileMethodSummary 74
com.ibm.wala.dalvik.util
.AndroidAnalysisScope 84
.AndroidEntryPointLocator 47, 48, 50, 77, 131
.AndroidEntryPointLocator.LocatorFlags 85
.AndroidEntryPointManager 47, 49, 65, 85, 147
.AndroidManifestXMLReader 49, 77
.AndroidSettingFactory 65
.androidEntryPoints 48, 117
.androidEntryPoints.ActivityEP 118
.androidEntryPoints.ApplicationEP 117
.androidEntryPoints.FragmentEP 123
.androidEntryPoints.LoaderCB 131
.androidEntryPoints.LocationEP 131
.androidEntryPoints.ProviderEP 118
.androidEntryPoints.ServiceEP 125
com.ibm.wala.ipa
.callgraph 33
.callgraph.AnalysisOptions 84
.callgraph.AnalysisScope 31, 84
.callgraph.CGNode 33
.cha.ClassHierarchy 32
com.ibm.wala.ssa
.IR 34
.SSAGotoInstruction 77
com.ibm.wala.util
.PrimitiveAssignability 75
.ssa 69
.ssa.IInstantiator 51, 75
.ssa.ParameterAccessor 73
.ssa.SSAValue 71
.ssa.SSAValueManager 52, 73
.ssa.TypeSafeInstructionFactory 72

edu.kit.joana.SuSi2Joana 80
edu.kit.joana.wala.core
.PDGNodeCreationVisitor 79
edu.kit.joana.wala.jodroid
.CliProgressMonitor 79
.entrypoints 54
.entrypointsFile.Tags 139

joana.ui.ifc
.sdg.graphviewer 37
.wala.console 36

Literaturverzeichnis

[1] Agesen, O. Constraint-based type inference and parametric polymorphism. In *Static Analysis*. Springer, 1994, pp. 78–100. 2.2.2

[2] Agesen, O. The cartesian product algorithm. In *ECOOP'95 — Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995*, M. Tokoro and R. Pareschi, Eds., vol. 952 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1995, pp. 2–26. 2.2.2

[3] Allen, F. E. Control flow analysis. *SIGPLAN Not. 5*, 7 (July 1970), 1–19. 2.1.2

[4] Allen, J. R., Kennedy, K., Porterfield, C., and Warren, J. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (New York, NY, USA, 1983), POPL '83, ACM, pp. 177–189. 2.1.3

[5] Andersen, L. O. *Program analysis and specialization for the C programming language*. PhD thesis, 1994. 2.2.2

[6] Adroid developer reference manual. https://developer.android.com/reference/packages.html. 2.3.3, 2.3.4, 4.5.1

[7] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (Oct. 1991), 451–490. 2.2.4

[8] Bytecode for the Dalvik VM. http://source.android.com/tech/dalvik/dalvik-bytecode.html. 2.3.7

[9] Adroid developer guide. https://developer.android.com/guide. 2.3.4, 3

[10] Din66001, Sept. 1966. Programmablaufplan. 2.1.1

[11] Dolby, J., and Sridharan, M. Static and dynamic program analysis using wala. In *Proceedings of the 2010 ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation* (June 2010). 1.2

[12] Droidbench – benchmarks. http://sseblog.ec-spride.de/tools/droidbench/. 6.4.1

[13] FERRANTE, J., OTTENSTEIN, K. J., AND WARREN, J. D. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst. 9*, 3 (July 1987), 319–349. 2.1.5

[14] FINK, S., DOLBY, J., AND COLBY, L. Semi-automatic j2ee transaction configuration. Tech. rep., Technical Report RC23326, IBM, 2004. 2.2.2

[15] Flowdroid – taint analysis. http://sseblog.ec-spride.de/tools/flowdroid/. 5.4.3, 6.3, 6.4.1

[16] FRITZ, C. Flowdroid: A precise and scalable data flow analysis for android. Master's thesis, Technische Universität Darmstadt, 2013. 6.3, 4

[17] FUCHS, A. P., CHAUDHURI, A., AND FOSTER, J. S. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa* (2009). 2.4.3

[18] GIFFHORN, D., AND HAMMER, C. Precise analysis of java programs using joana. In *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on* (sept. 2008), pp. 267 –268. 1.2, 2.5

[19] GROVE, D., AND CHAMBERS, C. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst. 23*, 6 (Nov. 2001), 685–746. 2.2.2

[20] GROVE, D., DEFOUW, G., DEAN, J., AND CHAMBERS, C. Call graph construction in object-oriented languages. In *Proceedings of the 12th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications* (New York, NY, USA, 1997), OOPSLA '97, ACM, pp. 108–124. 2.2.2

[21] HIND, M. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (New York, NY, USA, 2001), PASTE '01, ACM, pp. 54–61. 2.2.2

[22] HORWITZ, S., REPS, T., AND BINKLEY, D. Interprocedural slicing using dependence graphs. *SIGPLAN Not. 39*, 4 (Apr. 2004), 229–243. 2.1.3, 2.1.3, 2.1.5, 2.1.6, 2.1.6, 2.1.7, 2.1.8

[23] Insecurebank. http://www.paladion.net/downloadapp.html. 6.4.3

[24] LENGAUER, T., AND TARJAN, R. E. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst. 1*, 1 (Jan. 1979), 121–141. 2.1.2

[25] MITCHELL, J. Type inference and type containment. In *Semantics of Data Types*, G. Kahn, D. MacQueen, and G. Plotkin, Eds., vol. 173 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1984, pp. 257–277. 2.2.3

[26] OTTENSTEIN, K. J., AND OTTENSTEIN, L. M. The program dependence graph in a software development environment. *SIGPLAN Not. 19*, 5 (Apr. 1984), 177–184. 2.1.5, 2.1.6

[27] PODGURSKI, A., AND CLARKE, L. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *Software Engineering, IEEE Transactions on 16*, 9 (sep 1990), 965 –979. 2.1.3, 2.1.4

[28] RYDER, B. Constructing the call graph of a program. *Software Engineering, IEEE Transactions on SE-5*, 3 (may 1979), 216 – 226. 2.2.1

[29] SHIVERS, O. Control flow analysis in scheme. Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation. 2.2.2

[30] SHIVERS, O. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Pittsburgh, PA 15213, USA, 1991. 2.2.2

[31] Smali disassembler for dalvik bytecode. https://code.google.com/p/smali/. 2.4.3

[32] Soot: a java optimization framework. http://www.sable.mcgill.ca/soot/. 6.3

[33] STEENSGAARD, B. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1996), POPL '96, ACM, pp. 32–41. 2.2.2

[34] Susi – sources and sinks. http://sseblog.ec-spride.de/tools/susi/. 5.4.3, 6.1.4

[35] TUMBLESON, C. E. A. android-apktool. https://code.google.com/p/android-apktool/. 2.3.8

[36] Javadoc api dokumentation walas. http://wala.sourceforge.net/javadocs/. 2.4.1

[37] Offizielles wala wiki bei sourceforge. http://wala.sourceforge.net/. 2.4

[38] WEISER, M. Program slicing. In *Proceedings of the 5th international conference on Software engineering* (Piscataway, NJ, USA, 1981), ICSE '81, IEEE Press, pp. 439–449. 2.1.6, 2.1.6, 2.1.8