Karlsruhe Institute of Technology

Institut für Programmstrukturen
und Datenorganisation (IPD)

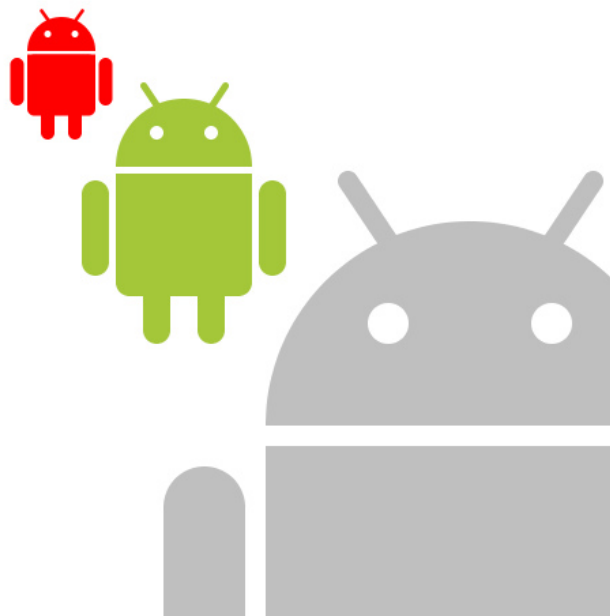
Lehrstuhl Prof. Dr.-Ing. Snelting

Intent Analyse von Android Applikationen

Bachelorarbeit von

Maik Wiesner

an der Fakultät für Informatik



Erstgutachter:	Prof. Dr.-Ing. Gregor Snelting
Zweitgutachter:	Prof. Dr. rer. nat. Bernhard Beckert
Betreuende Mitarbeiter:	Dipl.-Math. Dipl.-Inform. Martin Mohr

Bearbeitungszeit: 14. Juli 2016 – 27. Oktober 2016

Zusammenfassung

Intents sind ein zentraler Bestandteil vieler Android Applikationen, da sie für den Datenaustausch sowohl innerhalb einer App, als auch zwischen Apps verantwortlich sind. Die Kenntnis des Datenflusses ermöglicht das Finden von Sicherheitsrisiken und Datenlecks und ist daher von großem Interesse. Insbesondere ist man an Empfänger und Inhalt der Intents interessiert. Diese Arbeit beschäftigt sich mit einer statischen Analyse der Apps. Es wird eine String-Analyse vorgestellt, welche den Schwerpunkt auf die Betrachtung der Intents setzt, aber auch für allgemeine Analysen beliebiger Java-Applikationen verwendet werden kann. Ferner wird die Implementierung evaluiert und die Effizienz und Aussagekraft bewertet.

[0]Quelle des Titelbilds: `https://flic.kr/p/7Pqjkk`

Inhaltsverzeichnis

1 Einführung **7**

2 Grundlagen und Verwandte Arbeiten **9**
2.1 Grundlagen der String Analyse 9
2.1.1 Beschreibung des Problems 9
2.1.2 Bestimmung der Lösung 13
2.1.3 Fluss-insensitive Analyse 14
2.1.4 Weitere Begrifflichkeiten 15
2.2 Verwandte Arbeiten 19
2.2.1 *BRICS*-String-Analyzer [1] 19
2.2.2 Andere Arbeiten 20

3 Implementierung **21**
3.1 Wichtige Klassen 21
3.2 Gleichungssystem 22
3.3 Umgang mit Schleifen (Widening) 22
3.4 Umgang mit Parametern 24
3.5 Erweiterbarkeit 25
3.6 Annahmen und Einschränkungen 25
3.7 Kennzahlen 26

4 Evaluation **27**
4.1 Betrachtung ausgewählter Beispiele 27
4.1.1 ActivityCommunication2 27
4.1.2 ActivityCommunication3 30
4.2 Statistik 32
4.2.1 Laufzeit 32
4.2.2 App-Eigenschaften 35
4.2.3 Speicherbedarf 38
4.2.4 Präzision 40

5 Fazit **43**

1 Einführung

Typischerweise werden während der Benutzung einer App Informationen und Daten ausgetauscht. Betrachten wir dazu folgendes Szenario. Ein Benutzer erhält eine SMS mit dem Link zu einer Internetseite. Möchte er diese öffnen, so bietet Android ihm eine Auswahl der installierten Apps (bspw. Browser) an, die in der Lage sind, diese zu öffnen. Im Hintergrund wurden dazu alle installierten Apps betrachtet, um diejenigen zu bestimmen, die für die auszuführende Aktion in Frage kommen. Die Information, dass eine Internetseite geöffnet werden soll, hat die eigentliche SMS-App also verlassen und prinzipiell kann jede dieser Apps, die zu öffnende Seite erhalten. Daher stellen solche Kanäle potenzielle Schwachstellen dar, über die ein Angreifer womöglich Zugriff auf sensible Daten erhält.

In Android wird das Senden und Empfangen solcher Informationen mittels Intents realisiert, welche hauptsächlich eine auszuführende Aktion beschreiben. Im Wesentlichen bestehen sie aus drei Teilen:

action Eine String Variable, welche die Aktion des Intents beschreibt. Dies kann das Starten einer anderen App, das Öffnen einer Internetseite, oder das Wählen einer Telefonnummer sein.

data Eine String Variable, welche einen zusätzlichen Parameter der Aktion enthält, wie z.B die Telefonnummer, die gewählt werden soll oder die URL einer zu öffnenden Internetseite.

extras Eine Liste von Key-Value-Paaren, mit denen zusätzliche Parameter übergeben werden können.

Wir sind daran interessiert zu erfahren mit welchem Inhalt und Empfänger Intents von Apps versendet werden, da wir dadurch den Fluss der Daten kennen und Auskunft geben können über z.B eventuell vorhandene Datenlecks und offene Kanäle für Angreifer. Jedoch können wir die App nicht einfach ausführen um diese Fragen zu beantworten, da dies nur einem Black-Box Test entspräche. Es erfolgten Ein- und Ausgaben, aber die für uns relevanten Informationen blieben im Verborgenen. Vielmehr ist für unsere Zwecke eine statische Analyse, d.h eine Inspektion des Codes vonnöten.
Da ein Großteil der relevanten Variablen von Intents Strings sind, lassen sich mit

Hilfe einer String-Analyse mögliche Werte bestimmen. Die grobe Idee dahinter soll ein Beispiel zeigen. Betrachten wir dazu folgende Codezeilen:

```
1 String s;
2 int input = readUserInput(); //Benutzereingabe
3 if (input != 0) {
4         s = "a";
5 } else {
6         s = "b";
7 }
```

Code 1.1: Einfaches Beispiel

Abhängig von der Benutzereingabe wird der Wert von **s** zu a oder b gesetzt. Statisch lässt sich offensichtlich nicht ermitteln welcher Ausführungspfad eingeschlagen wird, da der Wert von **input** erst zur Laufzeit bekannt ist. Das Ergebnis einer String-Analyse für **s** könnte $\mathbf{s} \in \{a, b\}$ lauten. Das Ziel einer solchen Analyse ist es daher den tatsächlich angenommenen Wert eines Strings durch eine geeignete Menge möglicher Werte zu approximieren. Diese sollte möglichst klein sein, um ein präzises Ergebnis zu gewährleisten. Im Beispiel oben liefert jede Obermenge von $\{a, b\}$ ein korrektes Ergebnis. Insbesondere gilt das auch für **ANYSTRING**, welches die Menge aller möglichen String-Werte bezeichnet, gleichzeitig aber auch die unpräziseste Lösung darstellt.

Eine genaue Beschreibung der Vorgehensweise findet sich in Kapitel 2.1. Des weiteren werden in Kapitel 2.2 verwandte Arbeiten betrachtet, welche sich ebenfalls mit String-Analysen beschäftigen, aber für den benötigten Einsatz ungeeignet sind oder Probleme mit sich bringen.

Kapitel 3 geht auf eine Implementierung der vorgestellten String-Analyse mit Hilfe von Wala[2] ein und erläutert die zu Grunde liegenden Konzepte. Abschließend wird die Implementierung im Hinblick auf Effizienz und Präzision evaluiert und bewertet. (Kapitel 4)

2 Grundlagen und Verwandte Arbeiten

In diesem Kapitel wird zunächst die grundlegende Funktionsweise einer String-Analyse erläutert. Danach werden verwandte Arbeiten betrachtet und hervorgehoben warum diese für den vorgesehenen Einsatz ungeeignet sind bzw. worin die Unterschiede zu dieser Arbeit liegen.

2.1 Grundlagen der String Analyse

2.1.1 Beschreibung des Problems

Die String Analyse stellt eine spezielle Form der Datenflussanalyse dar, deren Ziel die Bestimmung möglicher Werte von String-Variablen ist, und wird im folgenden an einem Beispiel erläutert. Betrachten wir dazu wieder das Codefragment aus 1.1.

```
1 String s;
2 int input = readUserInput(); //Benutzereingabe
3 if (input != 0) {
4         s = "a";
5 } else {
6         s = "b";
7 }
```

Code 2.1: Einführungsbeispiel aus 1.1

Zunächst stellen wir fest, dass das Programm verschiedene Ausführungspfade hat, bedingt durch den if-else Block. Wollen wir die Werte bestimmen, die **s** nach Ausführung annehmen kann, müssen wir alle Pfade betrachten. Dabei hilft uns der Kontrollflussgraph (Abbildung 2.1), welcher eben diesen Zweck erfüllt, indem er den Kontrollfluss eines Programms beschreibt.

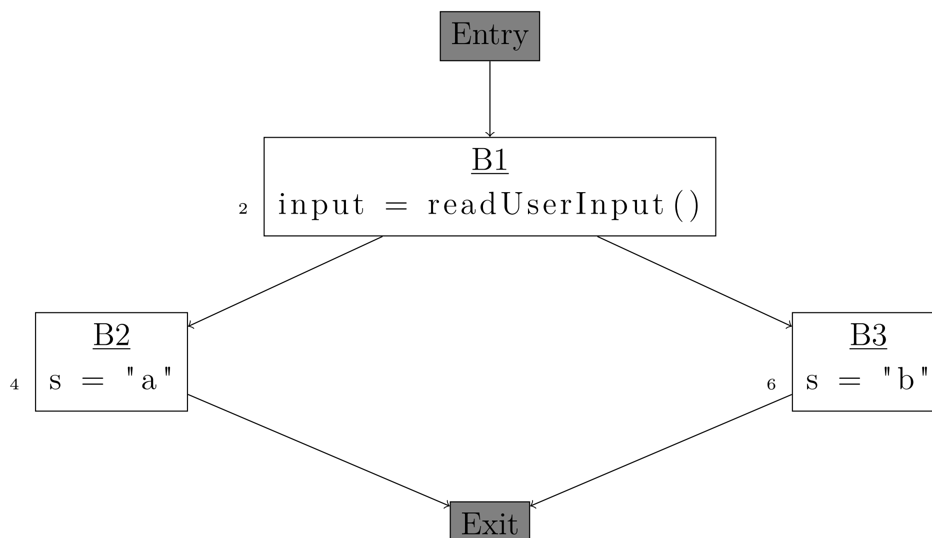


Abbildung 2.1: Kontrollflussgraph

Die Knoten dieses Graphen, auch *Basic Blocks* genannt, stellen Programmabschnitte dar, die nur einen Ausführungspfad besitzen, d.h. keine Verzweigungen oder Sprünge enthalten. Sie bestehen aus einer Folge von Befehlen $b_1, \ldots, b_n$, welche nacheinander abgearbeitet werden.
Zudem besitzt jeder Kontrollflussgraph obligatorische *Entry*- und *Exit* Knoten. Sie beinhalten keine Befehle, sondern repräsentieren Start- und Endpunkt eines Programms. Dadurch hat die Abstraktion im Kontrollflussgraph genau einen Einstiegs- und Ausstiegspunkt, unabhängig von der tatsächlichen Struktur des Programms. Somit beschreibt jeder Pfad von *Entry* zu *Exit* eine mögliche Ausführung.

Für jeden Befehl b definieren wir $\mathbf{IN}_b$ und $\mathbf{OUT}_b$. Dies sind Mengen, welche den Zustand des Programms vor Ausführung von b (IN) bzw. nach Ausführung von b (OUT) repräsentieren. In unserem Fall enthalten sie String-Variablen (oder auch andere relevante Variablen) und deren Werte.
$\mathbf{OUT}_b$ ist abhängig von $\mathbf{IN}_b$ und der Semantik von b. Hierfür führen wir *Transferfunktionen* f_b ein. Diese beschreiben die Semantik eines Befehls b, indem sie zu einer gegebenen IN-Menge die entsprechende OUT-Menge ausgeben. Hier und auch im Folgenden bezeichne IN_b^* bzw. OUT_b^* die Menge aller möglicher IN- bzw. OUT-Mengen von b.

$$f_b : \mathrm{IN}_b^* \to \mathrm{OUT}_b^* \tag{2.1}$$

$\mathbf{OUT}_b$ lässt sich damit definieren durch

$$\mathrm{OUT}_b := f_b(\mathrm{IN}_b) \tag{2.2}$$

Da die Ausführungsreihenfolge der Befehle $b_1, \ldots, b_n$ innerhalb eines Basic Blocks B eindeutig ist, können wir ihn auch als einen einzigen Befehl $B = b_n \circ \cdots \circ b_1$ interpretieren, womit eine Definition der IN- und OUT-Mengen für Basic Blocks möglich

ist. Hierbei nutzen wir aus, das für aufeinanderfolgende Befehle der Zusammenhang $\text{OUT}_{b_i} = \text{IN}_{b_{i+1}} (1 \leq i < n)$ gilt.

$$f_{B=b_n \circ \cdots \circ b_1} : \text{IN}^*_{b_1} \to \text{OUT}^*_{b_n} \tag{2.3}$$

$$\begin{aligned} f_{B=b_n \circ \cdots \circ b_1}(\text{IN}_{b_1}) :&= f_{b_n}(f_{b_{n-1}}(\ldots(f_{b_1}(\text{IN}_{b_1})))) && (2.4)\\ &= (f_{b_n} \circ \cdots \circ f_{b_1})(\text{IN}_{b_1}) && (2.5)\\ &= f_B(\text{IN}_{b_1}) && (2.6) \end{aligned}$$

Somit ergibt sich die Transferfunktion f_B von B durch Komposition der Transferfunktionen der einzelnen Befehle von B. Da zudem der Programmzustand vor Eintritt eines Basic Blocks dem Zustand vor Ausführung des ersten Befehls des Blocks entspricht, lassen sich die IN- und OUT-Mengen für Basic Blocks definieren durch

$$\begin{aligned} \mathbf{IN}_B &:= \text{IN}_{b_1} && (2.7)\\ \mathbf{OUT}_B &:= f_B(IN_B) && (2.8) \end{aligned}$$

Im folgenden bezeichne f_i die Transferfunktionen der Zeile i im obigen Beispiel (Code 2.1).

$$\begin{aligned} f_2 &= \text{IN} \mapsto \text{IN} \cup \ \{\textbf{input} = \text{readUserInput}()\} && (2.9)\\ f_4 &= \text{IN} \mapsto \text{IN} \cup \ \{\mathbf{s} = a\} && (2.10)\\ f_6 &= \text{IN} \mapsto \text{IN} \cup \ \{\mathbf{s} = b\} && (2.11) \end{aligned}$$

Wir brauchen Transferfunktionen nur für jene Befehle, welche die IN- und OUT-Mengen tatsächlich beeinflussen, d.h. jene die Variablen definieren bzw. verändern. Dies muss nicht zwangsweise eine Zuweisung sein, sondern kann auch durch Seiteneffekte hervorgerufen werden. Ein Beispiel wäre der Aufruf von *append* eines StringBuilders.

Es gilt noch zu klären, wie IN- und OUT Mengen von verschiedenen Basic Blocks zusammenhängen. Beispielsweise ist leicht einzusehen, dass $\text{IN}_{B1} = \text{OUT}_{Entry}$ gilt, da *B1* der einzige unmittelbare Nachfolger von *Entry* ist. Im Allgemeinen ist dieser Zusammenhang jedoch nicht so trivial. Betrachten wir dazu *Exit*. Dieser Block hat zwei unmittelbare Vorgänger, nämlich *B2* und *B3*. Abhängig vom gewählten Pfad gilt entweder $\text{IN}_{Exit} = \text{OUT}_{B2}$ oder $\text{IN}_{Exit} = \text{OUT}_{B3}$. Da wir an einer konservativen Lösung[1] interessiert sind, müssen wir beide Fälle in der IN-Menge von *Exit* berücksichtigen. In unserem Fall entspricht das der Vereinigung der jeweiligen OUT-Mengen,

[1] Eine Lösung ist konservativ, wenn sie in jedem Fall die korrekte Lösung enthält, d.h. sie berücksichtigt jeden theoretisch möglichen Pfad des Kontrollflussgraphen, unabhängig davon, ob er tatsächlich eingenommen wird.

d.h. $\text{IN}_{Exit} = \text{OUT}_{B2} \cup \text{OUT}_{B3}$.
Verallgemeinern lässt sich das durch

$$\text{IN}_B = \bigcup_{\tilde{B} \text{ ist unmittelbarer Vorgänger von B}} \text{OUT}_{\tilde{B}} \tag{2.12}$$

Der Operator, der verschiedene Pfade zusammenfasst, wird als **Meet-Operator** bezeichnet. Hier ist es die Vereinigung.

Somit sind, bis auf IN_{Entry}, alle IN- und OUT-Mengen definiert. Da dies der Startpunkt des Programms ist, kann diese Menge keine Variablen oder Werte enthalten. Wir definieren

$$\text{IN}_{Entry} := \emptyset \tag{2.13}$$

Die dazugehörige OUT-Menge ergibt sich durch die Transferfunktion, welche sowohl für *Entry*, als auch für *Exit* der Identitätsfunktion entspricht, da diese Blöcke keine Anweisungen enthalten, und die IN-Mengen somit nicht verändern.

Das Ergebnis, an dem wir letztendlich interessiert sind, ist OUT_{Exit}, da diese Menge den Zustand der Variablen nach Betrachtung aller möglichen Ausführungspfade beschreibt. Wir können die String-Analyse zusammenfassend wie folgt formulieren.

String-Analyse: Berechnung möglicher Werte von String-Variablen

Gegeben:

(1) Kontrollflussgraph $G = (V_{BB}, E)$ mit

$$V_{BB} = \{B \mid B \text{ ist Basic Block}\} \cup \{\text{Entry}, \text{Exit}\}$$
$$E = \{(B_i, B_j) \mid B_i \text{ ist unmittelbarer Vorgänger von } B_j\}$$

(2) Transferfunktion f_B für jeden Basic Block $B \in V_{BB}$, wobei

$$f_{\text{Entry}}(\text{IN}) = f_{\text{Exit}}(\text{IN}) = \text{IN}$$

Gesucht: OUT_{Exit}, wobei gelten muss

$$\text{IN}_{Entry} = \emptyset \tag{2.14}$$
$$\text{OUT}_B = f_B(\text{IN}_B), \quad \forall B \in V_{BB} \tag{2.15}$$
$$\text{IN}_B = \bigcup_{(\tilde{B},B)\in E} \text{OUT}_{\tilde{B}}, \quad \forall B \in V_{BB} \setminus \{\text{Entry}\} \tag{2.16}$$

Bemerkung: Auch wenn nur OUT_{Exit} gesucht ist, so liegen aufgrund der Bedingungen 2.14 - 2.16 mit OUT_{Exit} auch alle anderen OUT-Mengen vor.

2.1.2 Bestimmung der Lösung

Für unser obiges Beispiel ergibt sich folgendes Gleichungssystem.

$$\text{IN}_{\text{Entry}} = \emptyset \tag{2.17}$$
$$\text{OUT}_{\text{Entry}} = \text{IN}_{\text{Entry}} \tag{2.18}$$
$$\text{IN}_{B_1} = \text{OUT}_{\text{Entry}} \tag{2.19}$$
$$\text{OUT}_{B_1} = \text{IN}_{B_1} \cup \{\textbf{input} = \textbf{ANY_INT}\} \tag{2.20}$$
$$\text{IN}_{B_2} = \text{OUT}_{B_1} \tag{2.21}$$
$$\text{OUT}_{B_2} = \text{IN}_{B_2} \cup \{\mathbf{s} = a\} \tag{2.22}$$
$$\text{IN}_{B3} = \text{OUT}_{B_1} \tag{2.23}$$
$$\text{OUT}_{B_3} = \text{IN}_{B_3} \cup \{\mathbf{s} = b\} \tag{2.24}$$
$$\text{IN}_{\text{Exit}} = \text{OUT}_{B_2} \cup \text{OUT}_{B_3} \tag{2.25}$$
$$\text{OUT}_{\text{Exit}} = \text{IN}_{\text{Exit}} \tag{2.26}$$

Hier lässt sich die Lösung recht einfach bestimmen, da die Bedingung $\text{IN}_{\text{Entry}} = \emptyset$ (Gleichung 2.17), die anderen Mengen eindeutig festlegt. Im Allgemeinen ist dies aber nicht der Fall. Betrachten wir hierzu die Codezeilen in 2.2.

```
1 String s = "a";
2 for (int i = 0; i < n; i++) {
3         s = "a";
4 }
```

Code 2.2: Methode mit einer Schleife

Der dazugehörige Kontrollflussgraph (Abbildung 2.2) hat einen Zykel.

Als Gleichung für IN_{B_2} ergibt sich

$$\text{IN}_{B_2} = \{\mathbf{s} = a\} \cup f_{B_2}(\text{IN}_{B_2}) \tag{2.27}$$
$$= \{\mathbf{s} = a\} \cup (\{\mathbf{s} = a\} \cup \text{IN}_{B_2}) \tag{2.28}$$
$$= \{\mathbf{s} = a\} \cup \text{IN}_{B_2} \tag{2.29}$$

Mit $f'_{B_2}(\text{IN}) := \{\mathbf{s} = a\} \cup f_{B_2}(\text{IN})$ erhalten wir eine Fixpunktgleichung.

$$\text{IN}_{B_2} = f'_{B_2}(\text{IN}_{B_2}) \tag{2.30}$$

Somit ist IN_{B_2} ein Fixpunkt von f'_{B_2}. Dass dieser nicht eindeutig ist, sieht man leicht ein, wenn man Gleichung 2.29 betrachtet, welche äquivalent ist zu Gleichung[2] 2.31

$$\text{IN}_{B_2} \supseteq \{\mathbf{s} = a\} \tag{2.31}$$

[2] Wir verwenden den Begriff Gleichung, auch wenn es sich tatsächlich um eine Ungleichung handelt.

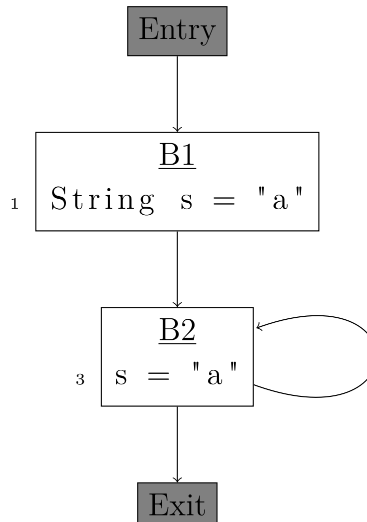


Abbildung 2.2: Kontrollflussgraph von 3.1

Wünschenswert ist eine minimale Lösung, d.h. eine Lösung die Bestandteil jeder anderen Lösung ist, da wir den Wertebereich der Variablen so präzise wie möglich bestimmen wollen.
Für sich alleine betrachtet wäre $\text{IN}_{B_2} := \{\mathbf{s} = a\}$ der kleinste Fixpunkt. Jedoch müssen für die kleinste Lösung des gesamten Gleichungssystems alle Gleichungen und alle Mengen berücksichtigt werden, d.h. wir suchen den kleinsten Fixpunkt der Funktion

$$f^* : \text{IN}_1^* \times \cdots \times \text{IN}_n^* \to \text{OUT}_1^* \times \cdots \times \text{IN}_n^* \quad , \tag{2.32}$$

welche die IN-Mengen der n Basic Blocks auf die jeweiligen OUT-Mengen abbildet. Dieser kann mit Hilfe von Algorithmus 1 bestimmt werden. Hierbei sind die IN-Mengen initial leer und es werden solange Elemente hinzugefügt, bis keine Änderung mehr eintritt.

2.1.3 Fluss-insensitive Analyse

Android Applikationen haben keinen bestimmten Einstiegspunkt, wie z.B die *main*-Methode bei einer klassisches Java-Anwendung. Daher ignorieren wir den Fluss, d.h. den Programmverlauf der Methoden-übergreifend eingenommen wird und betrachten stattdessen jede Methode für sich. Wie im vorherigen Abschnitt erläutert, extrahieren wir aus dem Kontrollflussgraphen einer Methode gewisse Gleichungen. Die Gesamtheit aller Gleichungen aller Methoden ergibt unser Gleichungssystem. Lösen wir dieses mit Hilfe der Fixpunktiteration entspricht das Ergebnis der Ausführung aller Methoden in beliebiger Reihenfolge.
Betrachten wir beispielsweise die beiden Methoden in Abbildung 2.3, so wäre das Ergebnis der Analyse $sb \supseteq \{\varepsilon, a, b, ab, ba\}$, da wir nicht wissen, ob die Methoden

Algorithmus 1 Kleinster Fixpunkt
(leicht angepasste Version von [3, Algorithmus 9.11])

Eingabe: Kontrollflussgraph $G = (V, E)$ und Transferfunktionen f_B für alle $B \in V$
Ausgabe: Kleinster Fixpunkt $\text{IN} = (\text{IN}_{B_1}, \ldots, \text{IN}_{B_n})$

1: **for** jeder Basic Block B {
2: $\text{IN}_B := \emptyset$
3: } *// end for*
4: $\text{OUT}_{\text{Entry}} := f_{\text{Entry}(\text{IN}_B)}$
5: **while** bel. OUT-Menge hat sich geändert {
6: **for** jeder Basic Block B außer *Entry* {
7: $\text{IN}_B := \bigcup_{\tilde{B} \text{ ist Vorgänger von } B} \text{OUT}_{\tilde{B}}$
8: $\text{OUT}_B := f_B(\text{IN}_B)$
9: } *// end for*
10: } *// end while*

Abbildung 2.3: *append*-Aufrufe auf einem Feld

```
public void a() {
sb.append("a");
}
```

```
public void b() {
sb.append("b");
}
```

überhaupt aufgerufen werden, und falls doch, in welcher Reihenfolge. Für eine praktische Implementierung müssen gewisse Annahmen getroffen werden. So erhalten wir das eben genannte Ergebnis nur, wenn wir annehmen, dass jede Methode maximal einmal aufgerufen wird.
Felder stellen in diesem Kontext eine besondere Herausforderung dar.

2.1.4 Weitere Begrifflichkeiten

SSA-Form

Die *Static-Single-Assignment*-Form ist eine Zwischensprache, mit der Eigenschaft, dass jede Variable höchstens eine Definitionsstelle besitzt. Dazu ein Beispiel.

Jede Definitionsstelle erhält eine neue, eindeutige SSA-Variable. In der Literatur findet sich oft die Notation verschiedene SSA-Variablen einer ursprünglichen Variable durch Indizes zu unterscheiden.

```
x = a + b
x = x − c
y = x − d
y = 2 ∗ y
```

Code 2.3: ursprünglicher Code

```
x1 = a + b
x2 = x1 − c
y1 = x2 − d
y2 = 2 ∗ y1
```

Code 2.4: SSA-Form

Abbildung 2.4: SSA-Form Beispiel mit arithmetischen Operationen

Eine weitere Besonderheit der SSA-Form ist die Phi-Funktion, die genutzt wird um Definitionen mehrerer Kontrollflusspfade zusammenzufassen. Sie liefert immer die SSA-Variable zurück, deren Kontrollflusspfad eingenommen wurde. Ist beispielsweise die Bedingung der *if*-Klausel in der SSA-Form 2.6 in Abbildung 2.5 wahr, so wird y der Wert der Variablen x_1 zugewiesen.

```
if (...) {
    x = a
} else {
    x = b
}
y = x
```

Code 2.5: ursprünglicher Code

```
if (...) {
    x1 = a
} else {
    x2 = b
}
y = φ(x1 ,x2)
```

Code 2.6: SSA-Form

Abbildung 2.5: Bedeutung der Phi-Funktion

Bemerkung: Die SSA-Form wird auch von Wala genutzt.

Pi-Knoten

Betrachten wir einfache, teilweise rekursive Zuweisungen (innerhalb einer Methode), so liegen aufgrund der SSA-Form stets nicht rekursive Gleichungen vor, wie in Abbildung 2.7 verdeutlicht. Die Konkatenation zweier Mengen ist hierbei Elementweise zu verstehen.

Ändern sich die Werte der Variablen allerdings nicht durch Zuweisungen, sondern durch Seiteneffekte, wie es beispielsweise bei StringBuildern der Fall ist, so gilt das nicht mehr. Da es keine weiteren Definitionsstellen gibt, werden keine neuen SSA-Variablen benötigt. Die Folge sind rekursive Gleichungen wie in Abb. 2.9.

Eine Fixpunktiteration würde hier nicht terminieren, da nie ein Fixpunkt erreicht werden würde. Jedoch wissen wir, dass die Konkatenation nur ein mal ausgeführt

```
String s = "a";
String t = "bc";
s = s + t;
s = s + "d";
```

Code 2.7: Menge von Zuweisungen

```
String s1 = "a";
String t1 = "b";
s2 = s1 + t1;
s3 = s2 + "d";
```

Code 2.8: SSA-Form

$s_1 \supseteq \{a\}$
$t_1 \supseteq \{b\}$
$s_2 \supseteq s_1 \cdot t_1$
$s_3 \supseteq s_2 \cdot \{d\}$

Abbildung 2.6: Gleichungen

Abbildung 2.7: Nicht rekursive Gleichungen

```
sb.append("a");
sb.append("b");
sb.append("c");
```

Code 2.9: Zuweisung per Seiteneffekt

$sb \supseteq sb \cdot$ "a"
$sb \supseteq sb \cdot$ "b"
$sb \supseteq sb \cdot$ "c"

Abbildung 2.8: Resultierende Gleichungen

Abbildung 2.9: rekursive Gleichungen

wird, und nicht beliebig oft. Ignorieren wir daher den Aspekt der Terminierung und nehmen an, die String-Analyse würde ein bereits ausgeführtes *append* während der Fixpunktiteration nicht nochmals durchführen. Dann ergeben sich die in Tabelle 2.1 aufgelisteten Mengen, welche ein weiteres Problem aufzeigen.

Tabelle 2.1: Zwischenergebnisse der Fixpunktiteration

	Initiale Menge	1.Iteration	2.Iteration	3.Iteration
sb	ε	ε,a,b,c	ε,a, ba, ca, ab, b, cb, ac, bc, c	$\ldots bac$, cab, abc, bca

Hierbei handelt es sich um die Zwischen- und Endergebnisse, die durch Ausführung der Anweisungen in beliebiger Reihenfolge entstehen. Da wir allein aus den Gleichungen nicht auf die Ausführungsreihenfolge der Anweisungen schließen können, enthält der Fixpunkt alle Kombinationen nach Ausführung einzelner und beliebig vieler Instruktionen. Zwar ist der exakte Wert *abc* in der letzten Menge enthalten, jedoch ist das Resultat dennoch sehr unpräzise.

Eine Möglichkeit, mit der sich sowohl die rekursiven Gleichungen, als auch das Problem der Reihenfolge lösen lassen, sind *Pi-Knoten*. Diese können nach bestimmten Anweisungen (z.B *append*) eingefügt werden und sichern den Zustand einer SSA-Variable, indem sie diese kopieren und umbenennen. Nachfolgende Instruktionen verwenden den Pi-Knoten anstatt der ursprünglichen Variablen. Die SSA-Form von 2.9 mit eingefügten Pi-Knoten ist in 2.10 aufgeführt.

```
1 sb₁.append("a")
2 sb₂ = pi[sb₁]
3 sb₂.append("b")
4 sb₃ = pi [sb₂]
5 sb₃.append("c")
6 sb₄ = pi[sb₃]
```

Code 2.10: Pi-Knoten

Letztendlich entsteht eine Kette von Pi-Knoten

$$sb_1 \rightarrow sb_2 \rightarrow sb_3 \rightarrow sb_4$$

Der Wert von sb_4 lässt sich nun durch Rückverfolgen der Kette und Konkatenation der entsprechenden Komponenten berechnen.

Mit Hilfe der Pi-Knoten lassen sich Gleichungen aufstellen, welche nicht rekursiv sind.

$$sb_2 \supseteq sb_1 \cdot \{a\}$$
$$sb_3 \supseteq sb_2 \cdot \{b\}$$
$$sb_4 \supseteq sb_3 \cdot \{c\}$$

Es sei noch angemerkt, dass dies nur für Anweisungen innerhalb einer Methode möglich ist, da hier die Reihenfolge der Anweisungen tatsächlich bekannt ist. Ist **sb** beispielsweise ein Feld und **sb**.*append(...)* wird in verschieden Methoden aufgerufen, so ist die Menge aller möglichen Kombinationen (wie in Tabelle 2.1) das bestmögliche Resultat.

2.2 Verwandte Arbeiten

2.2.1 *BRICS*-String-Analyzer [1]

Die hier beschriebene Vorgehensweise stellt eine sehr mächtige und präzise String-Analyse dar, welche bei dem Versuch, diese umzusetzen jedoch einige Probleme mit sich brachte. Zunächst soll allerdings grob die Vorgehensweise skizziert werden.

Ausgangspunkt ist auch hier der Kontrollflussgraph des zu analysierenden Programms. Anstatt eine Fixpunktiteration durchzuführen, wie in 2.1 beschrieben, wird eine kontextfreie Grammatik G aus dem Graphen extrahiert. In einem nächsten Schritt wird die von G beschriebene Sprache $L(G)$ durch eine reguläre Sprache $L_{\text{Reg}} \supseteq L(G)$ approximiert, die von einem endlichen Automaten A akzeptiert wird. Mit Hilfe von A lässt sich dann ein regulärer Ausdruck *Reg* bestimmen, welcher L_{Reg} beschreibt.

Es existiert auch eine Implementierung der Analyse in Java mit zwei entscheidenden Nachteilen.

1. Für die Erstellung des Kontrollflussgraphen wird *Soot*[4] verwendet.

2. Die Analyse ist nicht auf Android ausgelegt, sodass die Interpretation wichtiger Methoden (u.A. auch von Intents) hinzugefügt werden müsste.

Um diese Probleme zu beheben, verwendet die Implementierung ein Front- und ein Backend. Das Frontend transformiert den Kontrollflussgraphen in eine Zwischensprache, welche dann vom Backend für die eigentliche Analyse verwendet wird. Der Versuch das Frontend an Wala und Android anzupassen scheiterte, da die so resultierende Analyse für eine praktische Verwendung unbrauchbar war. Sie lieferte für jeden String **ANYSTRING** als Ergebnis. Für eine erfolgreiche Implementierung ist eine genaue Kenntnis der verwendeten Zwischensprache notwendig, um die Informationen die das verwendete Framework (Soot, Wala, etc.) liefert in Gänze an das Backend weiterzugeben. Es hat sich herausgestellt, dass diese zu komplex war, als dass ein Anpassen des bestehenden Frontends ohne eine komplette Neuimplementierung möglich wäre. Aufgrund dessen wurde dieser Ansatz verworfen.

2.2.2 Andere Arbeiten

Das Problem der meisten bestehenden String-Analysen ist, dass diese für die Analyse von Intents zu umfangreich und komplex sind, da sie entweder allgemein gehalten sind, und Aspekte im Zentrum haben, die bei der Betrachtung von Intents eher zweitrangig sind, oder zu konkret und auf einen bestimmten Anwendungsfall zugeschnitten sind. Hinzu kommt, dass sofern bestehende Implementierungen existieren, diese nicht Wala verwenden.

Die unter [5] vorgestellte Analyse konzentriert sich auf einen Widening-Operator, der benötigt wird, falls der iterative Algorithmus (Alg. 1) nicht terminieren würde. Dies ist beispielsweise der Fall, wenn eine Konkatenation innerhalb einer Schleife ausgeführt wird, unter der Annahme, dass beliebig viele Iterationen möglich sind. Ein solcher String würde mit jedem Iterationsschritt länger werden und es entstünden rekursive Gleichungen wie in 2.9. Ein Widening-Operator erzwingt eine Terminierung, indem er beispielsweise den Grenzwert einer konvergierenden Iterationenfolge bestimmt (siehe auch Abschnitt 3.3). Das vorgestellte Widening ist für unsere Zwecke jedoch zu komplex, da es uns genügt bei Konkatenationen in einer Schleife den Kleen'schen Abschluss zu bilden.
Eine andere Analyse, unter [6] zu finden, betrachtet die Länge von Strings, um mögliche Speicherüberläufe zu finden. Für unsere Analyse ist das nicht relevant.

3 Implementierung

Die in Kapitel 2 vorgestellte String-Analyse wurde in Java implementiert, wobei das Killdall-Framework von Wala[1] verwendet wurde.

3.1 Wichtige Klassen

An dieser Stelle werden die wichtigsten Klassen aufgeführt, da diese die Basis für genauere Erläuterungen bilden.

StringAnalysis Startpunkt der Analyse und Schnittstelle für den Benutzer. Der Konstruktor erhält eine Klassenhierarchie oder eine apk-Datei, aus welcher die Klassenhierarchie generiert wird. Standardmäßig werden die deklarierten Felder und Methoden aller Klassen des *ApplicationScope* für die Analyse herangezogen. Der Benutzer kann weitere Methoden und Felder hinzufügen.

Superprovider Verwaltet das Gleichungssystem und dessen Variablen. Bietet Schnittstellen um Gleichungen und Variablen einzufügen bzw. zu verändern.

NodeVariable Verwaltet die SSA-Variablen eines Basic Blocks und wird von den Transferfunktionen benutzt. Hauptaufgabe ist das Delegieren an den **Superprovider**.

FieldOrLocal Repräsentiert eine String Variable und verwaltet die dazugehörigen Gleichungen (siehe Abschnitt 3.2).

MethodVariable Repräsentiert eine Methode. Enthält die dazugehörige IR und **FieldOrLocal**-Variablen für den Rückgabewert, erhaltene Parameter (durch Aufrufe anderer Methoden) und lokale Variablen.

IntentVar Repräsentiert einen Intent, bestehend aus den Komponenten *action, data, extra* und *class*, welche selbst den Typ **FieldOrLocal** haben.

[1] https://github.com/joana-team/wala
Commit-Hash: 793312ef300ef5cbd300f4812ff2f61579757a8a

3.2 Gleichungssystem

Das Gleichungssystem selbst ist nur implizit gegeben. Jede Variable (**FieldOrLocal**) v enthält eine Menge $M = \{v_1, \dots v_n\}$ weiterer Variablen, wodurch die Gleichungen $v \supseteq v_1, v \supseteq v_2, \dots, v \supseteq v_n$ repräsentiert werden. Die v_i haben wiederum ihrerseits eine Menge an Gleichungen. Letztendlich ergibt sich ein Graph, dessen Kanten die einzelnen Gleichungen darstellen, wie in Abb. 3.1 beispielhaft illustriert. v ist dabei die Wurzel[2] und wird vom **Superprovider** verwaltet. Wurzelelemente sind *Hotspots*, da dies die Variablen sind, deren Werte uns interessieren.
Zu beachten ist, dass aufgrund der Transitivität von $\supseteq$ der Pfad $v \to v_1 \to v_3$ der Gleichung $v \supseteq v_3$ entspricht. Dadurch können wir Zyklen ignorieren, da diese nur bereits bekannte Gleichungen darstellen. So beschreibt z.B die Kante (v_3, v_1) auf v bezogen die Gleichung $v \supseteq v_1$, welche bereits durch die Kante (v, v_1) repräsentiert wird. Zwar fällt durch ignorieren der Kante (v_3, v_1) auch die Gleichung $v_3 \supseteq v$ weg, jedoch hat dies auf die Werte von v keinen Einfluss. Wäre v_3 ebenfalls ein Hotspot, würden wir in einem gesonderten Durchgang den Graphen mit v_3 als Wurzel betrachten.

Die Werte von v lassen sich durch Traversierung des Graphen berechnen. Hierzu werden, ähnlich der Tiefensuche, ausgehend von v alle Pfade durchlaufen bis ein Zielknoten erreicht wird. Zu den Zielknoten zählen alle Knoten mit Ausgangsgrad 0 und Knoten, dessen Nachfolger einen Zykel schließen, d.h. zu denen ein kürzerer Pfad von v ausgehend existiert. Wurde ein solcher Knoten erreicht, können dessen Werte rekursiv dem Wertebereich des Vorgängers hinzugefügt werden. Schließlich erhält v alle Werte.

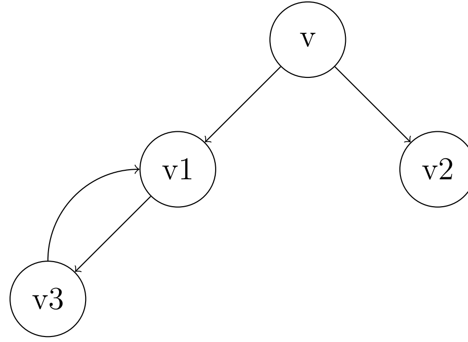
Zur Berechnung der Ergebnisse muss der **Superprovider** daher nur die Hotspots kennen und deren Werte bestimmen. Dadurch ist auch sichergestellt, dass nur die Gleichungen betrachtet werden, die zur Berechnung tatsächlich benötigt werden.

3.3 Umgang mit Schleifen (Widening)

Die Konkatenation innerhalb von Schleifen stellt ein eigenständiges Problem dar, da wir beliebig viele Iterationen annehmen müssen. Betrachten wir dazu Code 3.1 und die dazugehörige SSA-Form (Code 3.2), welche die in Kapitel 2.1.4 vorgestellten Pi-Knoten enthält. Da zwei mögliche Definitionen von *sb* (Zeile 1 und 7) den Beginn der Schleife erreichen, erhalten wir einen Phi-Knoten sb_ϕ, für den sich folgende

[2]Der Begriff Wurzel meint hier nur, dass dieser Knoten Ausgangspunkt unserer Betrachtungen ist, und darf nicht mit der graphentheoretischen Definition bei Betrachtung von Bäumen verwechselt werden, zumal der Graph im Allgemeinen kein Baum ist.

Abbildung 3.1: Repräsentation der Gleichungen für v



rekursive Gleichung ergibt.

$$\begin{aligned} sb_\phi &= \phi(sb_p, sb_l) && (3.1)\\ &= sb_p \cup sb_l && (3.2)\\ &= sb_p \cup \pi(sb') && (3.3)\\ &= sb_p \cup \pi(\pi(sb_\phi)) && (3.4) \end{aligned}$$

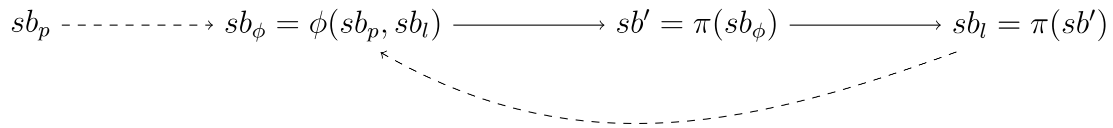
Dabei ist $\pi(\pi(sb_\phi))$ der Wert, der durch Konkatenation von *loop* an die bisherigen Werte von sb_ϕ entsteht (Ausführung der beiden *append* Befehle). Eine Fixpunktiteration würde die Menge $\{prefix, prefixloop, prefixlooploop, \dots\}$ berechnen, ohne zu terminieren. Um dies zu verhindern ersetzen wir die Menge durch den regulären Ausdruck $prefix \cdot \{loop\}^*$, der den Fixpunkt beschreibt, welcher von der Fixpunktiteration nur asymptotisch erreicht wird.
Der resultierende reguläre Ausdruck hat immer die Gestalt $p \cdot l^*$, wobei p die Werte derjenigen Variable sind, die außerhalb der Schleife definiert wurde, und l die Werte beschreibt die innerhalb der Schleife beliebig oft konkateniert werden können. In unserem Fall also sb_p und sb_l. Um dies zu veranschaulichen stellen wir die Abhängigkeiten grafisch dar. (Abb. 3.2)

Der abgebildete Graph beschreibt die Abhängigkeiten für Pi- und Phi-Knoten. Für jede der beiden SSA-Variablen von $\phi(sb_p, sb_l)$ führen wir eine Tiefensuche auf diesem Graphen durch und stellen fest, dass sb_l Teil eines Zykels ist. Somit sind die Werte von sb_p die Präfix-Komponente p des regulären Ausdrucks. Um l zu bestimmen, betrachten wir sb_l, d.h. die Variable die Teil des Zykels ist. Die Grundidee ist anzunehmen die Schleife würde genau eine Iteration durchführen, d.h. die Kante (sb_l, sb_ϕ) zu ignorieren. Definieren wir nun $sb_\phi := \{\varepsilon\}$, erhalten wir

$$sb_l = \pi(\pi(\{\varepsilon\})) = \pi(\{lo\}) = \{loop\} =: l \qquad (3.5)$$

Es sei erwähnt, dass die Semantik von $\pi(\cdot)$ immer von der zugrunde liegenden Anweisung abhängt. In unserer Implementierung ist das jedoch immer ein Aufruf von *append*.

Abbildung 3.2: Abhängigkeiten der Variablen
Zu einem Phi-Knoten führende Kanten sind gestrichelt und die zu einem Pi-Knoten führenden durchgehend




```
1 StringBuilder sb = new StringBuilder("prefix");
2 for (...) {
3 sb.append("lo");
4 sb.append("op");
5 }
6 String s = sb.toString();
```

Code 3.1: Konkatenation innerhalb einer Schleife

```
1 StringBuilder sbp = new StringBuilder("prefix");
2 for (...) {
3       sbϕ = ϕ(sbp, sbl)
4       sbϕ.append("lo");
5       sb' = π(sbϕ)
6       sb'.append("op");
7       sbl = π(sb')
8 }
9 String s = sbϕ.toString();
```

Code 3.2: Vereinfachte SSA-Form

3.4 Umgang mit Parametern

Die Implementierung ist in der Lage die Parameter eines Methodenaufrufs zu berücksichtigen. So wird z.B für jeden Aufruf $m(a, b, c)$ einer Methode m das Tripel (a, b, c) gespeichert. Enthält m die Anweisung $v = a$ für eine Variable v, so wird der Wertebereich von v alle Werte von a beinhalten mit denen m aufgerufen wurde, wobei a keine Konstante sein muss.
Zukünftige Erweiterungen könnten bei der Interpretierung weiterer Methoden (z.B *replace(a,b)* bei Strings) die Kombination der Parameter berücksichtigen um genauere Ergebnisse zu liefern.

3.5 Erweiterbarkeit

Soll die Analyse weitere Methoden interpretieren können, so sind die drei folgenden Schritte durchzuführen. Dabei sei erwähnt, dass der softwaretechnische Aspekt der Erweiterbarkeit nicht im Vordergrund stand.

1. Erben der Klasse *Invoke*. Diese beschreibt einen allgemeinen Methodenaufruf ohne Interpretation. Im Konstruktor werden die Arrays **params** und **types** gesetzt, welche die übergebenen Parameter und deren Typ (Konstante, lokale Variable, etc.) enthalten.

2. Überschreiben der Methode *evaluate(NodeVariable lhs, NodeVariable rhs)*. Die entstandenen Gleichung lassen sich über die entsprechende Methodenaufrufe von *lhs* dem globalen Gleichungssystem hinzufügen.

3. Eintragen der erstellten Klasse in der Methode *visitInvoke()* der Klasse *SSA-TransferfunctionFactory*

3.6 Annahmen und Einschränkungen

Um die Komplexität der Analyse eingrenzen zu können wurden folgende Annahmen und Einschränkungen gemacht.

- Bei der Analyse werden Werte des Typs *String, StringBuilder, StringBuffer* und *int* berücksichtigt.

- Bei Android Apps werden zusätzlich die Typen *Intent* und *Uri* beachtet.

- Es werden keine Exceptions geworfen, d.h. der Pfad im Kontrollflusspfad, der den Eintritt einer Exception beschreibt, wird nicht berücksichtigt.

- Es werden keine Pointer verfolgt. Der Aufruf von $o.m()$ hängt nur vom Typ O von o ab, nicht von o selbst, d.h. es wird angenommen, dass die Methode m der Klasse O aufgerufen wurde, selbst wenn o tatsächlich Instanz einer Unterklasse von O ist die m überschreibt.
  Zudem wird o dabei nicht berücksichtigt, d.h. es wird nicht zwischen einem Aufruf von $o_1.m()$ und $o_2.m()$ unterschieden. Ausnahmen bilden Instanzen der Klassen *StringBuilder, StringBuffer* und *Intent*

- Der Wert **null** wird ignoriert, d.h. die Zuweisung String s = **null** hat auf die Analyse keinen Einfluss.

3.7 Kennzahlen

Abschließend sind noch einige Kennzahlen der Implementierung aufgeführt.

Anzahl Klassen 48

Anzahl Methoden 344

Anzahl Packages 8

Gesamtzahl Zeilen Code 4322

4 Evaluation

In diesem Kapitel wird die Implementierung der String-Analyse bewertet. Dabei stehen vor allem Aspekte wie die Laufzeit und die Genauigkeit der berechneten String-Werte im Vordergrund.
Zunächst werden ausgewählte Apps betrachtet und mit Hilfe des Quellcodes das Ergebnis der String-Analyse mit dem exakten Ergebnis verglichen. Die darauf folgenden Abschnitte listen Tabellen und Grafiken auf, welche auf einer Analyse mehrerer hundert Apps basieren.

4.1 Betrachtung ausgewählter Beispiele

4.1.1 ActivityCommunication2

Die erste App, auf die näher eingegangen wird, ist *ActivityCommunication2* von DroidBench [7]. Sie beinhaltet die drei Klassen *InFlowActivity*, *IsolateActivity* und *OutFlowActivity*. Wir betrachten zunächst die Methode 4.1, welche der Übersichtlichkeit halber an einigen Stellen vereinfacht wurde.

```
1  @Override
2  protected void onCreate(Bundle savedInstanceState) {
3  super.onCreate(savedInstanceState);
4  setContentView(R.layout.activity_main);
5
6  TelephonyManager telephonyManager = ...
7  String imei = telephonyManager.getDeviceId(); //IMEI auslesen
8  Intent i = new Intent(
9            "ignore.edu.mit.icc_action_string_operations.ACTION"
10                   .substring(7));
11 //IMEI mit Key "DroidBench" dem Intent hinzufügen
12 i.putExtra("DroidBench", imei);
13 startActivity(i);
14 }
```

Code 4.1: onCreate Methode in OutFlowActivity.java

Die Methode versendet einen Intent, welcher die IMEI des Geräts im *extra*-Attribut enthält, mit „**DroidBench**“ als Key. In den beiden anderen Klassen wird per *getIntent()* ein Intent aufgefangen und durch einen Aufruf von *getExtraString(*„**DroidBench**“*)* der entsprechende Wert ausgelesen.

Die String-Analyse findet einen versendeten Intent mit folgenden Werten:

action = { edu.mit.icc .action_string_operations.ACTION }
extra = { DroidBench }

action enthält den effektiven Wert nach Aufruf von *substring(7)* und **extra** enthält alle verwendeten Key's.
Die Analyse verfolgt nur gesendete Intents, da dies ausreicht um den Fluss der Daten nachzuvollziehen. Tatsächlich ist es, ohne weiteres, nur dann möglich den Empfänger eines Intents anzugeben, wenn dieser explizit gesetzt wird. In allen anderen Fällen kann jede App, die die action des Intents kennt, diesen abfangen. Auch in diesem Fall können wir allein durch den Aufruf von *getExtraString(*„**DroidBench**“*)* in *InFlowActivity* und *IsolateActivity* nicht darauf schließen, dass der ausgelesene Wert der IMEI entspricht. Dazu müsste noch die *Manifest*-Datei analysiert werden (4.2), welche zeigt, dass nur *InFlowActivity* ein möglicher Empfänger ist, da hier der entsprechende Intent-Filter gesetzt ist. (Zeile 7 und Zeile 16, rot hinterlegt)
Auf diese Weise ermittelt Android auch intern den Empfänger eines Intents.

Eine solche Analyse würde allerdings den Rahmen sprengen und wird daher an dieser Stelle nicht weiter verfolgt. Eine Arbeit, die sich damit beschäftigt ist [8].

```
1  <activity
2       android:name="edu.mit.icc_action_string_operations.
            InFlowActivity"
3           android:label="@string/app_name" >
4           <intent-filter>
5                   <action android:name=
6                   "edu.mit.icc_action_string_operations.ACTION" />
7                   <category android:name="android.intent.
                        category.DEFAULT" />
8           </intent-filter>
9  </activity>
10
11 <activity
12          android:name="edu.mit.icc_action_string_operations.
                IsolateActivity"
13          android:label="@string/app_name" >
14          <intent-filter>
15                  <action android:name=
16                  "edu.mit.icc_action_string_operations.EDIT" />
17                  <category android:name="android.intent.
                        category.DEFAULT" />
18          </intent-filter>
19 </activity>
```

Code 4.2: Ausschnitt der Manifest.xml

4.1.2 ActivityCommunication3

Eine weitere App, ebenfalls von DroidBench [7], ist *ActivityCommuniction3*. Diese ist der vorhergegangenen sehr ähnlich, da sie ebenfalls aus den drei Klassen *InflowActivity*, *OutflowActivity* und *IsolateActivity* besteht. Auch hier wird in *OutFlowActivity* ein Intent versendet, und in den beiden anderen Klassen einer aufgefangen. Der entscheidende Unterschied liegt jedoch darin, wie der Intent erstellt wird (Code 4.3).

```
1  @Override
2  protected void onCreate(Bundle savedInstanceState) {
        ⋮
7  String imei = ...
8  ComponentName comp = new ComponentName(getPackageName(),
       InFlowActivity.class.getName());
9  Intent i = new Intent().setComponent(comp);
10 i.putExtra("DroidBench", imei);
11
12 startActivity(i);
13 }
```

Code 4.3: onCreate Methode in OutFlowActivit.java

Die Analyse liefert zwei gefundene Intents mit den Werten in Tabelle 4.1. Ein Vergleich mit dem Code zeigt, dass das Ergebnis nicht korrekt ist, da tatsächlich nur ein Intent versendet wird. Um zu verstehen weshalb die Analyse zwei Intents findet, muss die dazugehörige *IR* betrachtet werden (Code 4.4).

Tabelle 4.1: Inhalt der gefundenen Intents

	Intent 1	Intent 2
Quelle	OutFlowActivity.java.onCreate()	*unbekannt*
action	$\emptyset$	$\emptyset$
data	$\emptyset$	$\emptyset$
extra	$\emptyset$	{"DroidBench" }

Zunächst wird ein neuer Intent mittels **new**-Befehl instanziiert und in *v19* gespeichert. Das Problem ist der Aufruf von setComponent(...) unmittelbar danach, da diese Methode eine Referenz auf den erzeugten Intent zurückgibt. Zwar ist das referenzierte Objekt dasselbe, jedoch wird aufgrund der SSA-Form dieser eine neue SSA-Variable zugewiesen (*v22*). Somit enthält **i** im Code oben nicht *v19*, sondern *v22*, eine der Analyse unbekannte Variable, da die Information *v22 = v19* aufgrund fehlender Interpretation von setComponent(...) nicht vorhanden ist.

Dieses Problem tritt immer dann auf, wenn eine Methode, welche von der Analyse nicht interpretiert werden kann, eine Referenz des Objekts zurückgibt, auf dem sie aufgerufen wurde und diese weiter verwendet wird.

```
v19 = new Intent()
invokespecial v19.init() //Konstruktoraufruf
v22 = invokevirtual v19.setComponent(...)
v25 = invokevirtual v22.putExtra("DroidBench", v11)
invokevirtual startActivity(v22)
return
```

Code 4.4: Vereinfachte IR von OnCreate()

4.2 Statistik

Alle hier aufgeführten Statistiken beziehen sich auf einen Rechner mit folgenden Spezifikationen:

CPU Intel Core i7-4790 CPU @ 3.60GHz × 8

Betriebssystem Ubuntu 16.04 LTS 64 Bit

Java-Version 1.8.0_101

Der JVM wurden dabei 6 GB Heap zugeteilt. Zu beachten ist, dass insbesondere Werte wie die Laufzeit, stark von Java-internen Prozeduren (Garbage Collection, Just-In-Time Compilation, etc.) abhängen, auf die an dieser Stelle nicht weiter eingegangen wird. Speziell in Tabelle 4.2 ist das die Ursache der großen Streuung der Messdaten, welche zu einer hohen Standardabweichung führt. Aus Zeitgründen war es nicht möglich näher auf dieses Problem einzugehen, zumal dies eine nicht-triviale Angelegenheit ist. Eine genauere Beschreibung und mögliche Lösungsansätze finden sich unter [9]. Zukünftige Maßnahmen und Experimente könnten diesen Aspekt berücksichtigen.

4.2.1 Laufzeit

Tabelle 4.2 listet die unten aufgeführten Informationen für 100 Apps auf, wobei das Benchmark Programm für jeweils 50 Apps gesondert ausgeführt wurde.
Bei einigen Apps wurde beim erstellen des Gleichungssystem Wala-seitig eine Exception geworfen. Ursache ist ein Bug bei der Erstellung der *TypeInference*. Diese Apps sind durch ein E in der Spalte t_b gekennzeichnet und es wird nur die Anzahl der Klassen angegeben.

C Anzahl der Klassen der apk

I Anzahl gefundener Intents

t_b Zeit zum Aufstellen des Gleichungssystems in ms. Der angegebene Wert entspricht dem Durchschnittswert aus 30 Durchläufen innerhalb eines Benchmark-Durchlaufs.

t_s Zeit zum lösen des Gleichungssystems in ms. Hierbei wurden nur die Intents gelöst. Auch hier ist es der Durchschnittswert aus 30 Iterationen innerhalb eines Benchmark Durchlaufs.

σ_b / σ_s Standardabweichung der entsprechenden Messdaten in t_b bzw. t_s.

Tabelle 4.2: Laufzeitanalyse

Name	C	I	t_b	σ_b	t_s	σ_s
ch.fixme.status_17	38	5	67,87	14,69	0,67	0,48
com.lightbox.android.camera_2	163	16	158,23	57,68	0,9	0,4
com.anysoftkeyboard.languagepack .danish_2	10	0	0,2	0,41	0,03	0,18
com.eolwral.osmonitor_81	255	7	204,97	6,58	2,63	0,49
apps.babycaretimer_6	89	21	94,4	0,81	1,83	0,38
com.example.CosyDVR_10	28	5	23,67	0,48	0,5	0,51
com.tum.yahtzee_2	33	1	17,07	0,25	0,07	0,25
com.casimirlab.simpleDeadlines_18	64	12	37,23	0,5	1,03	0,18
com.lonepulse.travisjr_2	584	13	318,53	1,66	56,1	49,86
com.jakebasile.android.hearingsaver_13	19	2	5,67	0,48	0,03	0,18
com.doplgangr.secrecy_40	453	11	275,2	3,18	10,47	0,97
com.webworxshop.swallowcatcher_7	391	17	620,7	11,86	17,07	6,82
com.android.shellms_4	13	1	8,63	0,49	0,1	0,31
com.kpz.pomodorotasks.activity_8	62	8	32,13	0,43	0,17	0,38
com.blogspot.tonyatkins.freespeech_124	331	43	501,5	9,11	5,27	0,45
aws.apps.androidDrawables_8	281	14	355,6	0,67	2,5	0,51
com.google.android.gms_1300	75	8	23,63	0,49	0,07	0,25
com.ultrafunk.network_info_12	24	13	17,23	0,43	0,1	0,31
com.easytarget.micopi_19	37	7	33,53	0,51	0,3	0,47
com.anysoftkeyboard.languagepack .georgian.fdroid_5	24	2	1,3	0,47	0	0
com.asksven.betterwifionoff_43	580	60	648	1,39	16,43	0,5
com.stwalkerster.android.apps .strobelight_3	13	0	1,43	0,5	0	0
com.blntsoft.emailpopup_14	39	9	27,57	0,5	0,6	0,5
com.replica.replicaisland_14	257	22	382,03	4,62	2,97	0,32
com.moonpi.tapunlock_13	56	19	37,1	0,31	0,23	0,43
com.netthreads.android.noiz2_12	106	6	110,63	0,61	2,67	0,48
com.newsblur_89	412	46	322,9	1,06	5,63	0,56
com.ubergeek42.WeechatAndroid_12	584	28	1.042,53	16,02	4,83	0,38
com.omegavesko.holocounter_1	18	1	4,43	0,5	0,1	0,31
com.gueei.applocker_3	406	11	179,23	5,7	0,5	0,51
com.java.SmokeReducer_1	16	5	7,33	0,48	0,07	0,25
com.unwind.networkmonitor_1	19	0	4,67	0,48	0,1	0,31
com.daviancorp.android .monsterhunter3udatabase_5	383	40	459,43	0,77	8,57	0,68
damo.three.ie_17	734	11	940,7	97,54	24,9	0,71
com.ringdroid_20600	89	8	176,53	0,86	4,87	0,57

Name	C	I	t_b	σ_b	t_s	σ_s
com.nucc.hackwinds_10	1025	-	E	-	-	-
com.wordpress.sarfraznawaz .callerdetails_1	23	0	14,5	0,51	0,67	0,48
com.glTron_4	46	1	99,4	0,81	0,17	0,38
com.gmail.jerickson314.sdscanner_12	21	0	8,57	0,5	0,03	0,18
com.ivanvolosyuk.sharetobrowser_7	30	4	10,9	0,31	0,17	0,38
com.easwareapps.f2lflap2lock_adfree_3	24	10	9	0	0,1	0,31
com.googlecode.droidwall_157	40	4	62,47	1,83	1.519,87	637,78
com.xatik.app.droiddraw.client_4	309	26	385,7	1,02	2,27	0,45
br.usp.ime.retrobreaker_4	35	3	32,93	0,37	0,97	0,18
com.harleensahni.android.mbr_20140127	37	11	19,23	0,9	0,2	0,41
com.achep.widget.jellyclock_6	16	3	2,57	0,57	0	0
com.manuelmaly.hn_20	526	19	261,43	1,01	2,6	0,56
com.leafdigital.kanji.android_2	64	20	66,5	0,57	2,6	0,5
com.googlecode.chartdroid_18	38	0	10,87	0,35	0	0
com.hobbyone.HashDroid_18	54	6	219,43	0,73	0,1	0,31
org.wroot.android.goldeneye_1	13	1	19,1	13,73	0,17	0,38
fr.ybo.transportsbordeaux_291	550	55	330,27	45,68	21,57	6,18
kdk.android.simplydo_3	57	2	31,1	0,31	0,5	0,51
jonas.tool.saveForOffline_12	283	17	246,3	1,26	3,7	0,65
org.peterbaldwin.client.android .vlcremote_58	108	10	45	0,69	0,23	0,43
monakhv.android.samlib_23	420	33	469,07	10,23	4,37	0,49
ivl.android.moneybalance_3	48	0	26,97	0,32	0,2	0,41
org.wiktionary_1	113	20	174,17	4,29	4,2	0,41
jp.ksksue.app.terminal_11	32	5	38,3	0,47	0,2	0,41
org.chorem.android.saymytexts_7	121	22	152	0,87	12,03	0,32
org.jtb.alogcat_43	42	5	22,23	2,06	0,5	0,51
org.madore.android.unicodeMap_4	53	2	41,13	0,35	0	0
org.projectmaxs.module.contactsread_28	106	14	47,77	0,63	0,77	0,43
org.totschnig.myexpenses_193	1255	-	E	-	-	-
org.beide.droidgain_1	9	1	2,6	0,5	0,03	0,18
us.feras.mdv.demo_1	44	4	24,07	0,45	2,5	0,51
org.ligi.passandroid_248	3184	-	E	-	-	-
org.aja.flightmode_3	1	2	1,33	0,48	0	0
dk.andsen.asqlitemanager_17	105	33	204,1	0,76	7,7	0,53
de.mreiter.countit_2	23	1	9,83	0,38	0,1	0,31
se.erikofsweden.findmyphone_11	39	3	49,33	0,48	1,97	0,18
org.tamanegi.wallpaper .multipicture.dnt_48	136	24	163,03	0,61	0,73	0,45
pe.moe.nori_11	164	11	94,03	0,49	0,43	0,5
org.microg.nlp.backend.apple_1100	368	-	E	-	-	-

Name	C	I	t_b	σ_b	t_s	σ_s
org.jamienicol.episodes_9	388	2	352,43	3,9	3,77	0,43
org.projectmaxs.module.wifiaccess_28	102	14	50,57	0,5	0,83	0,38
org.openintents.about_8	45	13	26	0,37	0,4	0,5
free.yhc.netmbuddy_31	379	25	227,83	0,59	13,3	0,47
net.bytten.xkcdviewer_32	66	12	25,43	0,57	0,17	0,38
eu.woju.android.packages.hud_11	14	0	2,7	0,47	0	0
de.markusfisch.android.wavelines_3	36	1	21,6	0,5	0,13	0,35
net.phunehehe.foocam_4	62	0	64,8	1,13	0,7	0,47
fr.tvbarthel.apps.simpleweatherforcast_8	154	8	122,93	0,45	1,4	0,5
me.echeung.cdflabs_11	252	3	180,67	0,76	2,17	0,65
org.cyanogenmod.great.freedom_9	6	0	0,03	0,18	0	0
net.gaast.giggity_41	102	14	99,5	0,63	1	0
se.johanhil.duckduckgo_1	12	2	2,63	0,49	0,1	0,31
org.twelf.cmtheme_2	6	0	0,07	0,25	0	0
fr.tvbarthel.apps.simplethermometer_9	43	5	13,53	0,51	0,03	0,18
org.herrlado.ask.languagepack .lithuanian_9	23	2	1,4	0,5	0	0
org.tbrk.mnemododo_23	63	1	84,77	0,68	36,4	1,57
org.tasks_348	3813	99	3.116,57	48,75	72,03	18,05
org.openbmap_12	1182	39	1.810,63	28,34	38,23	0,73
free.yhc.feeder_57	335	27	217,57	0,5	10,03	0,18
youten.redo.ble.ibeacondetector_3	134	0	163,43	0,57	3,43	0,5
giraffine.dimmer_36	75	25	71	0,45	1,2	0,41
uk.ac.cam.cl.dtg.android.barcodebox_4	32	15	16,53	0,51	0,33	0,48
jp.takke.cpustats_8	21	6	30,67	0,48	0,13	0,35
fr.gaulupeau.apps.InThePoche_11	34	4	11	0	3,37	0,49
de.nico.ha_manager_22	44	2	16,47	0,57	0,23	0,43

Da alle Werte in einem vertretbaren Rahmen liegen, lässt sich festhalten, dass die Implementierung effizient ist und für praktische Anwendungen verwendet werden kann.

4.2.2 App-Eigenschaften

In Tabelle 4.3 sind Kennzahlen von 100 Apps aufgelistet. Für diejenigen, die eine Exception warfen, sind keine Zahlen angegeben.

M Anzahl der Methoden. Hierbei wurden nur nicht-abstrakte Methoden, sowie statische Initialisierer aus dem *ApplicationScope* berücksichtigt.

F Anzahl der Felder

I Anzahl der **Bytecode** Instruktionen. Diese müssen in keinem Zusammenhang mit der Anzahl Quellcodezeilen stehen, da komplexe (einzeilige) Quellcode-Instruktionen mehrere Bytecode Anweisungen ergeben können und der Quellcode Kommentare enthalten kann.

G Anzahl aufgestellter Gleichungen.

Tabelle 4.3: App-Eigenschaften

Name	M	F	I	G
ch.fixme.status_17	175	93	5054	740
com.lightbox.android.camera_2	1008	341	15777	5422
com.anysoftkeyboard.languagepack.danish_2	11	0	21	0
com.eolwral.osmonitor_81	1451	165	22776	8408
apps.babycaretimer_6	436	146	10460	2084
org.wroot.android.goldeneye_1	42	11	1026	108
fr.ybo.transportsbordeaux_291	2487	261	34164	13795
kdk.android.simplydo_3	265	54	4169	1042
jonas.tool.saveForOffline_12	1562	129	27723	14688
com.example.CosyDVR_10	110	16	2463	293
org.peterbaldwin.client.android.vlcremote_58	478	103	5091	3408
monakhv.android.samlib_23	3195	578	43060	12366
com.tum.yahtzee_2	126	14	1997	532
ivl.android.moneybalance_3	176	28	2774	537
org.wiktionary_1	-	-	-	-
com.casimirlab.simpleDeadlines_18	220	103	4105	477
com.lonepulse.travisjr_2	2291	265	32884	12765
com.jakebasile.android.hearingsaver_13	57	11	756	118
com.doplgangr.secrecy_40	2136	429	28128	10590
jp.ksksue.app.terminal_11	140	93	3216	755
org.chorem.android.saymytexts_7	594	120	12865	2884
com.webworxshop.swallowcatcher_7	2661	423	55841	42470
com.android.shellms_4	23	3	490	129
org.jtb.alogcat_43	199	42	2753	727
com.kpz.pomodorotasks.activity_8	294	48	3481	869
com.blogspot.tonyatkins.freespeech_124	3217	363	48759	23480
org.madore.android.unicodeMap_4	183	34	5909	3700
aws.apps.androidDrawables_8	2340	333	30188	8964
com.google.android.gms_1300	247	63	2592	636
org.projectmaxs.module.contactsread_28	544	132	5385	1712
com.ultrafunk.network_info_12	115	9	1903	408
org.totschnig.myexpenses_193	-	-	-	-

Name	M	F	I	G
org.beide.droidgain_1	22	3	359	41
com.easytarget.micopi_19	123	55	3373	520
com.anysoftkeyboard.languagepack.georgian.fdroid_5	31	39	308	6
com.asksven.betterwifionoff_43	4012	586	58076	19766
com.stwalkerster.android.apps.strobelight_3	26	3	232	20
us.feras.mdv.demo_1	189	19	2585	1488
org.ligi.passandroid_248	24713	2081	302162	200247
org.aja.flightmode_3	4	0	84	7
dk.andsen.asqlitemanager_17	571	97	18142	5622
com.blntsoft.emailpopup_14	115	39	2865	426
de.mreiter.countit_2	52	5	790	116
se.erikofsweden.findmyphone_11	260	66	5026	1439
com.replica.replicaisland_14	1777	472	52286	27006
com.moonpi.tapunlock_13	213	29	5395	678
org.tamanegi.wallpaper.multipicture.dnt_48	780	194	14047	6113
pe.moe.nori_11	829	131	11424	3123
com.netthreads.android.noiz2_12	620	303	11963	4094
org.microg.nlp.backend.apple_1100	2416	323	36161	14744
org.jamienicol.episodes_9	2680	443	40134	19387
com.newsblur_89	2086	245	36054	13649
org.projectmaxs.module.wifiaccess_28	531	133	5696	1867
org.openintents.about_8	144	81	2811	936
free.yhc.netmbuddy_31	1926	308	26520	13920
com.ubergeek42.WeechatAndroid_12	3895	591	66062	20831
net.bytten.xkcdviewer_32	263	17	2931	747
com.omegavesko.holocounter_1	45	0	361	44
eu.woju.android.packages.hud_11	44	6	458	86
com.gueei.applocker_3	1804	76	18828	10968
com.java.SmokeReducer_1	52	7	849	71
de.markusfisch.android.wavelines_3	98	18	1952	308
net.phunehehe.foocam_4	267	51	5234	1072
com.unwind.networkmonitor_1	38	10	446	58
fr.tvbarthel.apps.simpleweatherforcast_8	982	186	12638	5358
com.daviancorp.android.monsterhunter3udatabase_5	3245	660	43424	12202
me.echeung.cdflabs_11	1368	87	21367	13232
damo.three.ie_17	6258	485	104962	55799
org.cyanogenmod.great.freedom_9	6	5	12	0
net.gaast.giggity_41	487	75	10793	2333
se.johanhil.duckduckgo_1	28	10	318	41
com.ringdroid_20600	439	136	12180	3232
com.nucc.hackwinds_10	5218	890	69755	36175
org.twelf.cmtheme_2	6	5	12	0

Name	M	F	I	G
fr.tvbarthel.apps.simplethermometer_9	188	87	1561	522
com.wordpress.sarfraznawaz.callerdetails_1	52	1	1079	260
org.herrlado.ask.languagepack.lithuanian_9	30	39	306	6
org.tbrk.mnemododo_23	427	100	7262	1928
org.tasks_348	26740	2327	299128	200613
com.glTron_4	248	85	9278	1478
org.openbmap_12	9542	1675	159406	61192
free.yhc.feeder_57	1665	227	24151	10710
com.gmail.jerickson314.sdscanner_12	90	9	1030	529
youten.redo.ble.ibeacondetector_3	1257	103	15407	8386
com.ivanvolosyuk.sharetobrowser_7	97	13	1111	279
com.easwareapps.f2lflap2lock_adfree_3	60	5	1215	78
giraffine.dimmer_36	408	104	8176	1622
com.googlecode.droidwall_157	155	40	3761	1463
com.xatik.app.droiddraw.client_4	2476	347	34395	9405
br.usp.ime.retrobreaker_4	201	19	4256	854
com.harleensahni.android.mbr_20140127	106	26	1890	259
uk.ac.cam.cl.dtg.android.barcodebox_4	92	29	1770	239
com.achep.widget.jellyclock_6	27	0	340	21
com.manuelmaly.hn_20	2238	302	29704	15792
jp.takke.cpustats_8	97	12	4500	1804
com.leafdigital.kanji.android_2	247	66	5065	1520
com.googlecode.chartdroid_18	74	87	901	102
com.hobbyone.HashDroid_18	284	140	17955	7177
fr.gaulupeau.apps.InThePoche_11	79	35	1347	162
de.nico.ha_manager_22	127	12	1884	340

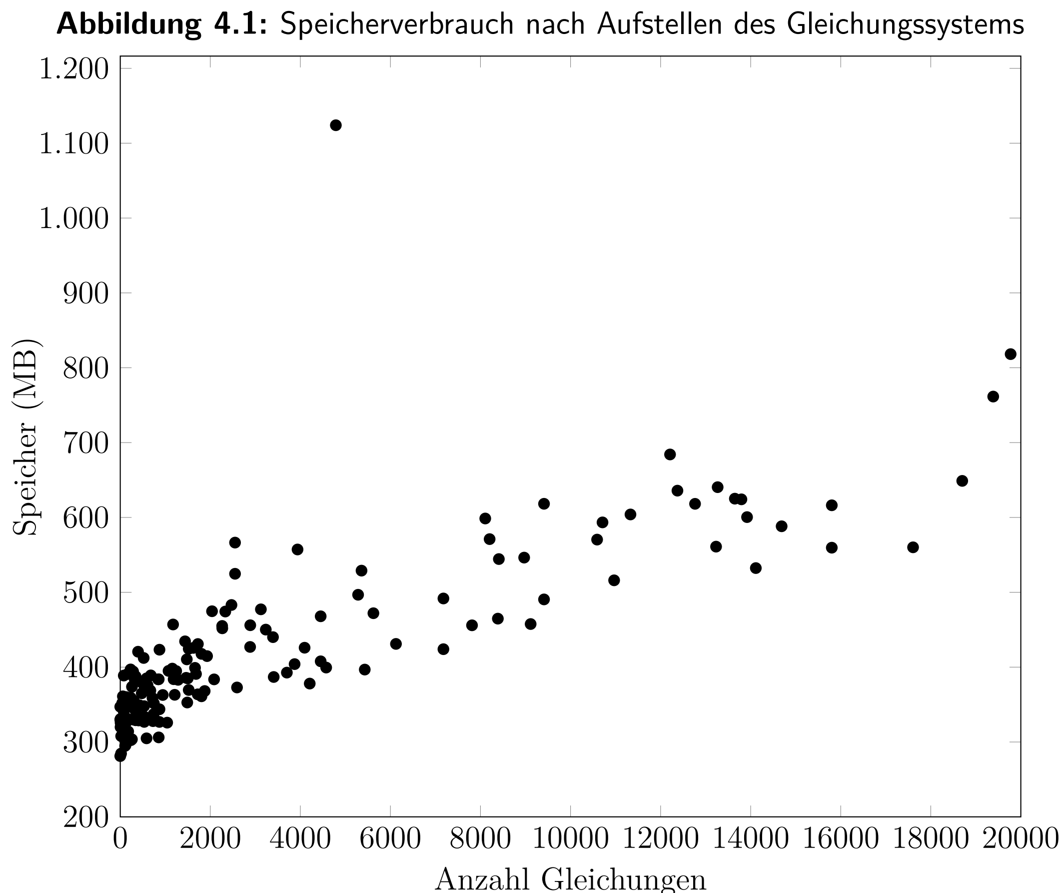
4.2.3 Speicherbedarf

Abbildung 4.1 zeigt den benötigten Speicherbedarf nach Aufstellen des Gleichungssystems in Abhängigkeit der Anzahl aufgestellter Gleichungen. Dieser Statistik lagen 200 Apps zugrunde, von denen jeweils 100 in einem Durchlauf des Benchmark-Programms analysiert wurden. Eingetragen ist der durchschnittliche Speicherbedarf nach 20 Iterationen. An dieser Stelle sei nochmals erwähnt, dass Java-interne Prozeduren, wie die Garbage Collection, in das Resultat mit einfließen.

Das Ergebnis entspricht der Erwartung, dass der benötigte Speicher linear mit der Anzahl aufgestellter Gleichungen wächst. Ein Großteil der Apps benötigt etwa 200-300 MB Speicher, bei relativ wenigen Gleichungen. Neben dem Speicher den Wala benötigt, müssen die Gleichungen selbst, sowie Methoden mit zugehöriger IR, Felder und Intents repräsentiert werden.

Eine App[1] fällt allerdings besonders auf, da sie mit 4788 Gleichungen und 1124 MB Speicher deutlich mehr benötigt, als Apps mit einer vergleichbaren Anzahl an Gleichungen. Bei genauerer Betrachtung zeigt sich jedoch, dass diese App mit 48.343 Instruktion auch viel mehr Anweisungen hat, als Apps mit ähnlich vielen Gleichungen. (vgl. Tabelle 4.3) Dementsprechend groß fallen dann auch die IR's aus. Nicht jede Anweisung resultiert in einer Gleichung. Tatsächlich hängen die Gleichungen nicht unmittelbar von den Instruktionen ab. Beispielsweise würde eine Methode, welche nur weitere Methoden aufruft, gar keine Gleichungen erzeugen.

Abbildung 4.1: Speicherverbrauch nach Aufstellen des Gleichungssystems



[1] org.vono.narau_6.apk

4.2.4 Präzision

Die Analyse der Präzision wurde sowohl für Methoden (4.2), als auch für Intents (4.3) durchgeführt, indem der Anteil der Variablen bestimmt wurde, deren Ergebnis unbefriedigend ist. Hierunter fallen Variablen mit dem Resultat **ANYSTRING**, aber auch Variablen mit mindestens 30 möglichen Werten (**ZU VIELE**). Mit Hilfe eines Boxplots lassen sich die Resultate sehr gut veranschaulichen, da sie alle relevanten statistischen Kenngrößen beinhalten. Die beiden äußersten Werte sind das 0,025-Quantil und das 0,975-Quantil. In der Mitte befindet sich ein Kasten, dessen Ränder dem unteren bzw. oberen Quartil entsprechen. Zusätzlich kennzeichnet die Mitte des Kastens den Median.
Bei den Intents ist der Kasten nicht sichtbar, da sowohl das untere Quartil, als auch das obere Quartil 0 ist.

Die Ergebnisse zeigen auf, dass die String-Analyse für allgemeine Analysen (z.B Reflection) eher durchschnittliche Ergebnisse liefert, was sicherlich auch daran liegt, dass einige Methoden von der Analyse nicht interpretiert werden können, und entsprechende Variablen dadurch zu **ANYSTRING** gesetzt werden.
In Bezug auf Intents jedoch, liefert die String-Analyse recht annehmbare Ergebnisse. Ein Grund dafür, ist die Tatsache, dass die meisten Werte Konstanten sind bzw. Werte, die nicht durch komplizierte Berechnungen entstehen. Mögliche Empfänger der Intents müssen beispielsweise die Key's der *extra*-Werte kennen, um auf diese zuzugreifen. Eine komplizierte oder zufällige Berechnung der Key's würde dem im Wege stehen.

Abbildung 4.2: Boxplot der Anystring-Analyse von Methoden: Betrachtet wurden pro App max. 50 Methoden mit einem String als Rückgabewert

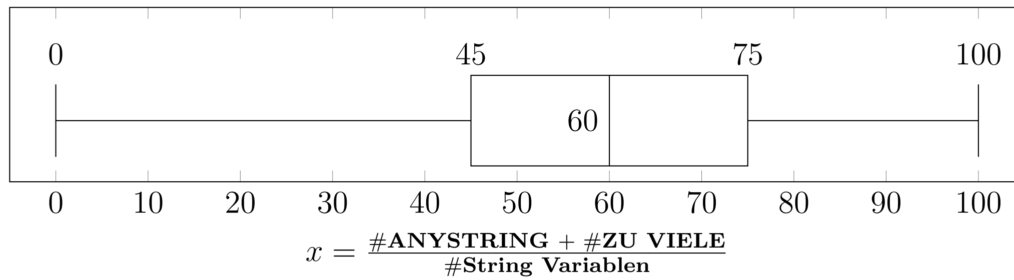
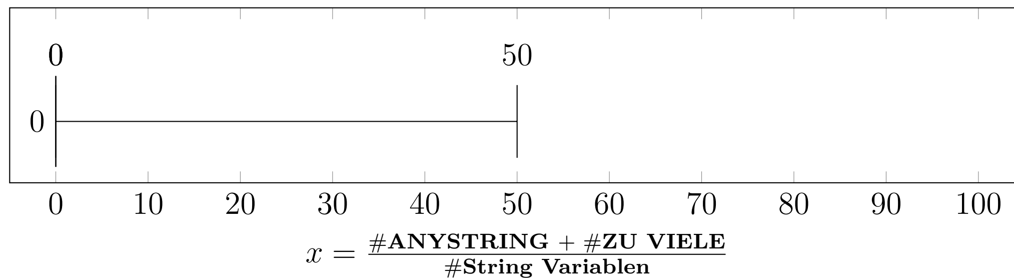


Abbildung 4.3: Boxplot der Anystring-Analyse von Intents: Betrachtet wurden die Komponenten der Intents (*action*,*data*, *extra*, *class*). Variablen mit einem leeren Wertebereich wurden ignoriert, da diese nicht eindeutig als Erfolg bzw. Misserfolg klassifiziert werden können. Bei einem Intent muss nicht jede Komponente gesetzt sein.



5 Fazit

Es wurde ein String-Analyse vorgestellt und implementiert, mit der sich Android-Apps im Hinblick auf Intents untersuchen lassen. Die Evaluation zeigte die Effizienz dieser sowohl in Bezug auf die Laufzeit, als auch auf die Genauigkeit der berechneten Ergebnisse.

Dennoch sind noch an vielen Stellen Optimierungen möglich. So ist die Anzahl der Methoden, die interpretiert werden können, im Moment noch überschaubar, womit sich eine Verbesserung der Resultate durch das Hinzufügen weiterer erzielen lässt.
Zudem handelt es sich bei der Implementierung um einen Prototyp, welcher größtenteils experimentell entstanden ist. Eine formalere Beschreibung des Problems, sowie ein softwaretechnischer Entwurf würden die Erweiterbarkeit vereinfachen und es ließen sich Aspekte berücksichtigen, die im Rahmen einer Bachelorarbeit ignoriert werden müssen.(z.B Pointer-Analyse)

Letztendlich kann diese Arbeit als Basis einer komplexeren und mächtigeren Analyse dienen, welche durch ihre Resultate das Potenzial und den Informationsgewinn, bezogen auf den Fluss der Daten, bei einer Analyse von Intents aufzeigt.

Literaturverzeichnis

[1] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Proc. 10th International Static Analysis Symposium (SAS)*, vol. 2694 of *LNCS*, pp. 1–18, Springer-Verlag, June 2003. Available from `http://www.brics.dk/JSA/`.

[2] "WALA." `http://wala.sourceforge.net/wiki/index.php/Main_Page`.

[3] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman, *Compiler: Prinzipien, Techniken und Werkzeuge (2. Auflage).* Pearson-Studium, 2008.

[4] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pp. 13–, IBM Press, 1999.

[5] T.-H. Choi, O. Lee, H. Kim, and K.-G. Doh, "A practical string analyzer by the widening approach," in *Proceedings of the 4th Asian Conference on Programming Languages and Systems*, APLAS'06, (Berlin, Heidelberg), pp. 374–388, Springer-Verlag, 2006.

[6] F. Yu, T. Bultan, and O. H. Ibarra, "Symbolic string verification: Combining string analysis and size analysis," in *in Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS*, pp. 322–336.

[7] "DroidBench." `https://github.com/secure-software-engineering/DroidBench.git`.

[8] T. Blaschke, "Automatische Modellierung des Lebenszyklus von Android-Anwendungen," Apr. 2014.

[9] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *SIGPLAN Not.*, vol. 42, pp. 57–76, Oct. 2007.

Erklärung

Hiermit erkläre ich, Maik Wiesner, dass ich die vorliegende Bachelorarbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

____________________ ____________________________________

Ort, Datum Unterschrift

Danke

Ich danke vor allem meinem Betreuer, Martin Mohr, für die tatkräftige Unterstützung bei Problemen jeglicher Art, seien sie praktischer oder theoretischer Natur. Zusätzlicher Dank geht an Simon Bischof für Ratschläge und Tipps.