Universität Bremen
Fachbereich 3 - Informatik/Mathematik

Masterarbeit

Evaluation und Weiterentwicklung eines Slicers zur Sicherheitsanalyse von Java Anwendungen

Marcel Meissner
Matrikelnummer: [redacted]

vorgelegt am: 5. April 2021

1. Gutachter: Dr. Karsten Sohr
2. Gutachter: Prof. Dr. Ute Bormann

Eidesstattliche Erklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe und dass ich alle Stellen, die ich wörtlich oder sinngemäß aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe. Die Arbeit hat bisher in gleicher oder ähnlicher Form oder auszugsweise noch keiner Prüfungsbehörde vorgelegen.

Bremen, den 5. April 2021

Marcel Meissner

Inhaltsverzeichnis

Inhaltsverzeichnis iv

Abbildungsverzeichnis vii

Programmcodeverzeichnis x

1 Einleitung 1
- 1.1 Motivation . . . 1
- 1.2 Ziele der Arbeit . . . 1
- 1.3 Aufbau der Arbeit . . . 2

2 Grundlagen 3
- 2.1 Programmanalyse & Slicing . . . 3
  - 2.1.1 Intraprozedural vs. Interprozedural . . . 6
  - 2.1.2 Kontrollflussgraph . . . 6
  - 2.1.3 Kontrollabhängigkeit . . . 7
  - 2.1.4 Datenabhängigkeit . . . 9
  - 2.1.5 Programmabhängigkeitsgraph . . . 11
  - 2.1.6 Zwischendarstellung SSA . . . 13
  - 2.1.7 Aufrufgraph . . . 15
  - 2.1.8 Systemabhängigkeitsgraph . . . 17
- 2.2 WALA . . . 18
  - 2.2.1 Das Framework . . . 18
  - 2.2.2 Funktionsweise . . . 19
  - 2.2.3 Ablauf im Slicer . . . 19
  - 2.2.4 WALA-Einstellungsmöglichkeiten durch Parameter . . . 20
    - 2.2.4.1 Datenabhängigkeitsoptionen . . . 20
    - 2.2.4.2 Kontrollabhängigkeitsoptionen . . . 21
- 2.3 Zusammenfassung des Kapitels . . . 22

3 Verwandte Arbeiten 23
- 3.1 Gulmann, Gerken, Detmers . . . 23
- 3.2 Philip Nguyen . . . 24
- 3.3 Alim Kerimov . . . 30

3.4 Michael Cyl . . . 33
3.5 Jan Möhlmann . . . 45
3.6 Aufgetretene Problematiken und Ausblicke der Vorarbeiterin . . . 49
3.7 Verschwundene Funktionalitäten . . . 51
3.8 Ersetzte Funktionalitäten . . . 52
3.9 Zusammenfassung des Kapitels . . . 53

4 Methodik **54**
4.1 Vorgehensweise . . . 54
4.2 Anforderungen . . . 55

5 Anforderungen **58**
5.1 Funktionale Anforderungen . . . 58
5.2 Nicht-funktionale Anforderungen . . . 69
5.3 Einschränkungen . . . 70

6 Implementierung **72**
6.1 Problematiken vor der Implementierung . . . 72
6.2 Make a new Slice . . . 74
6.2.1 Problematik der Ungleichheit . . . 75
6.2.2 Android Version & Java Jar Name . . . 79
6.2.3 Entrypoint Class & Entrypoint Methods . . . 82
6.2.4 Method Class & Method Names . . . 85
6.2.5 Seed Statements & Seed Statement Classes . . . 87
6.2.6 Object & Parameter Tracking . . . 91
6.2.6.1 tracking-Methode . . . 96
6.2.6.2 interprocedural-Methode . . . 97
6.2.6.3 addAliasesToSlice & isValueUsedByInstruction-Methoden . 99
6.2.6.4 intraprocedural-Methode . . . 101
6.2.7 Tracking to slicing criterion . . . 108
6.2.8 Local Variable & Attribute Tracking . . . 110
6.2.8.1 getParameterNamesGroupedBySourceFiles . . . 112
6.2.8.2 MethodVisitorParameterNames . . . 114
6.3 Slices-Tab & View a Slice . . . 117
6.4 Settings-Tab . . . 120
6.5 Sonstige Änderungen . . . 121
6.5.1 Slicer Options . . . 121
6.5.2 Java Parser . . . 121
6.6 Zusammenfassung des Kapitels . . . 128

7 Evaluation **129**
7.1 Eingliederung alter Beispiele . . . 129
7.1.1 Arbeit von Nguyen . . . 131
7.1.2 Arbeit von Cyl . . . 133
7.1.3 Arbeit von Kerimov . . . 135
7.2 Zusätzliche JUnit-Tests . . . 138
7.2.1 Code-Konvention Tests . . . 138
7.2.2 Einstiegspunkte-Tests . . . 139
7.2.3 Methoden für das Seed-StatementTests . . . 141
7.2.4 Tracking Tests . . . 142
7.2.4.1 Objekt-Tracking . . . 142
7.2.4.2 Parameter-Tracking . . . 147
7.2.4.3 Lokale Variablen und Attribut-Tracking . . . 147
7.3 Benchmark Tests für Cipher und Signature . . . 148
7.4 OSCI Tests . . . 153
7.5 Zusammenfassung des Kapitels . . . 165

8 Fazit **166**

Literaturverzeichnis **167**

A Anhang **170**
A.1 Programmcode . . . 170

Abbildungsverzeichnis

2.1 Abstraktes Beispiel des Slicings 4
2.2 Beispielprogramm für das Slicen 5
2.3 Beispielprogramm für das Slicen - Backward 5
2.4 Beispielprogramm für das Slicen - Forward 5
2.5 Schlüsselwörter und deren Kontrollflüsse 6
2.6 CFG für das Beispielprogramm 7
2.7 CFG mit Kontrollabhängigkeiten 8
2.8 CDG für das Beispielprogramm 9
2.9 CFG mit Datenabhängigkeiten 10
2.10 PDG für das Beispielprogramm 11
2.11 Backward-Slicing am PDG für das Beispielprogramm 12
2.12 Forward-Slicing am PDG für das Beispielprogramm 13
2.13 Beispielprogramm ohne SSA-Form 14
2.14 Beispielprogramm in SSA-Form 14
2.15 Ein CFG und der umgewandelte CFG in SSA-Form 14
2.16 Ein Programmfragment (a) und sein Aufrufgraph (b). Die rechteckigen Knoten entsprechen den Methoden, die runden Formen entsprechen den Aufrufanweisungen. 15
2.17 print-Methoden, geklont im Aufrufgraphen 16
2.18 SDG mit einen Beispielprogramm 18

3.1 Konfigurationsdatei 25
3.2 Programmablauf *findEntryPoints()* 26
3.3 TF07 ohne Objektverfolgung 28
3.4 TF07 mit Objektverfolgung 28
3.5 TF05 mit Objektverfolgung 29
3.6 Alte Paketstruktur des Slicers 31
3.7 Neue Paketstruktur des Slicers 31
3.8 Startbildschirm Android-Slicer 34
3.9 Slices-Tab Android-Slicer 34
3.10 Ansicht zur Erstellung einen neuen Slices 35
3.11 Ausschnitt von AlarmManagerService.java 36
3.12 Quellcodeansicht von AlarmManagerService 36

3.13 Ansicht eines fertigen Slices - Teil 1 . . . 38
3.14 Ansicht eines fertigen Slices - Teil 2 . . . 39
3.15 Ansicht eines fertigen Slices (Show-Diff und View Side by Side) . . . 40
3.16 Settings-Tab Android-Slicer . . . 41
3.17 Slicer-Options Android-Slicer . . . 41
3.18 CFA-Options Android-Slicer . . . 42
3.19 NEW Slices-Tab Android-Slicer . . . 45
3.20 NEW Ansicht zur Erstellung eines neuen Slices . . . 46
3.21 NEW Ansicht zur Erstellung eines neuen Slices . . . 46

6.1 Falsches Slice-Ergebnis . . . 74
6.2 Ansicht zur Erstellung eines neuen Slices Teil 1 - Marcel . . . 75
6.3 Ansicht zur Erstellung eines neuen Slices Teil 2 - Marcel . . . 75
6.4 Ungleiche Benennung der gleichen Einstellungsmöglichkeiten . . . 76
6.5 Javar Jar Name . . . 80
6.6 Zip Datei hochladen . . . 81
6.7 Entrypoint Class und Entrypoint Methods . . . 83
6.8 Programmablauf *getJavaEntrypoints()* . . . 85
6.9 Method Class und Method Names . . . 85
6.10 Beispiel einer erzeugten Methode durch WALA . . . 90
6.11 Slice-Ergebnis - ClassDMoreMethods - alte Objekt-Tracking Methoden . . . 92
6.12 Slice-Ergebnis - ClassDMoreMethods - nur interprozedural . . . 93
6.13 Slice-Ergebnis - ClassDMoreMethods - interprozedural und intraprozedural 93
6.14 Slice-Ergebnis - InterproceduralMoreObjects - ohne rekursiven Aufruf . . . 94
6.15 Slice-Ergebnis - InterproceduralMoreObjects - mit rekursiven Aufruf . . . 94
6.16 Slice-Ergebnis - ClinicExampleBean - ohne Tracking nach dem Slicing . . . 95
6.17 Slice-Ergebnis - ClinicExampleBean - miz Tracking nach dem Slicing . . . 96
6.18 Slice-Ergebnis - ObjectTrackingPutMoreObjectNewInFullConstructor - interprozedural . . . 102
6.19 Slice-Ergebnis - ObjectTrackingPutMoreObjectNewInFullConstructor - intraprozedural . . . 103
6.20 Slice-Ergebnis - interprocedural - ohne Methoden-Einschränkung . . . 109
6.21 Slice-Ergebnis - interprocedural - mit Methoden-Einschränkung . . . 110
6.22 Slice-Ergebnis - AttributeNoAssignment . . . 111
6.23 Slice-Ergebnis - AttributeNoAssignment - mit Attribut-Tracking . . . 111
6.24 Beispiel - IAlarmManager . . . 117
6.25 Slices-Tab - Marcel . . . 118
6.26 Ansicht eines fertigen Slices - Teil 1 - Marcel . . . 118
6.27 Ansicht eines fertigen Slices - Teil 2 - Marcel . . . 119
6.28 Ansicht eines fertigen Slices - Teil 3 - Marcel . . . 119
6.29 Settings-Tab - Java_Directory_Path . . . 120

6.30 Exclusion_List - Bearbeiten - Teil 1 . . . 121
6.31 Exclusion_List - Bearbeiten - Teil 2 . . . 121
6.32 Slice-Ergebnis - Try-Catch-Blöcke - alt . . . 123
6.33 Slice-Ergebnis - Try-Catch-Blöcke - alt . . . 123
6.34 Slice-Ergebnis - Import-Anweisungen - neu . . . 124
6.35 Slice-Ergebnis - Klassendeklaration - alt . . . 125
6.36 Slice-Ergebnis - Klassendeklaration - neu . . . 125
6.37 Slice-Ergebnis - If-Else-Block - alt . . . 126
6.38 Slice-Ergebnis - If-Else-Block - neu . . . 126
6.39 Slice-Ergebnis - Static-Block - alt . . . 127
6.40 Slice-Ergebnis - Static-Block - neu . . . 127

7.1 Beispiel einer Testmethode zum Testfall TF1" von Cyl . . . 130
7.2 Packageaufbau der Testklassen . . . 130
7.3 Slice-Ergebnis - SimpleTest-tf05 - alt . . . 132
7.4 Slice-Ergebnis - SimpleTest-tf05 - neu . . . 133
7.5 Slice-Ergebnis - PackageManagerService-tf10 - neu . . . 134
7.6 Slice-Ergebnis - JcaCipherService - neu - Teil 1.1 . . . 136
7.7 Slice-Ergebnis - JcaCipherService - neu - Teil 1.2 . . . 136
7.8 Slice-Ergebnis - JcaCipherService - neu - Teil 2 . . . 137
7.9 Slice-Ergebnis - JcaCipherService - neu - Teil 3 . . . 138
7.10 Log vom Slice-Ergebnis zum Einstiegspunkt-Test . . . 140
7.11 Log vom Slice-Ergebnis zum MethodNodes-Test . . . 142
7.12 Slice-Ergebnis von ObjectTrackingPutMoreStaticObject-Klasse . . . 145
7.13 Debug Teilansicht des Heapgraphen . . . 146
7.14 Vereinfachte und verkleinerte Teilansicht des Heapgraphen . . . 147
7.15 Slice-Ergebnis - Diff-Ansicht der Klasse Simple - tf02_main . . . 149
7.16 Slice-Ergebnis - AllInOneClass - kein Tracking . . . 151
7.17 Slice-Ergebnis - AllInOneClass - mit Tracking - Teil 1 . . . 152
7.18 Slice-Ergebnis - AllInOneClass - mit Tracking - Teil 2 . . . 152
7.19 Datenflüsse zwischen OneWayMessage_PassiveRecipient und der doFinal-Methode . . . 154
7.20 Slice-Ergebnis - SymCipherOutputStream - ohne Tracking . . . 155
7.21 Slice-Ergebnis - SOAPMessageEncrypted - nur der Konstruktor . . . 158
7.22 Slice-Ergebnis - DialogHandler - nur static-Block . . . 159

Programmcodeverzeichnis

6.1 Fehlermeldung beim Erzeugen eines Aufrufgraphen (Quelle: Eigene Darstellung) . . . 72
6.2 SliceService.java - *process*-Methode alt (Quelle: Eigene Darstellung) 73
6.3 SliceService.java - *process*-Methode neu (Quelle: Eigene Darstellung) 73
6.4 getEntrypoint-Algorithmus (Quelle: Eigene Darstellung) 76
6.5 getAndroidEntrypoint-Algorithmus (Quelle: Eigene Darstellung) 77
6.6 Java Algorithmus (Quelle: Eigene Darstellung) 78
6.7 Android Algorithmus (Quelle: Eigene Darstellung) 78
6.8 getAndroidEntrypoint Algorithmus Ausschnitt - alt (Quelle: Eigene Darstellung) . . . 84
6.9 getAndroidEntrypoint Algorithmus Ausschnitt - neu (Quelle: Eigene Darstellung) . . . 84
6.10 findMethodNodes - neu (Quelle: Eigene Darstellung) 86
6.11 Beispiel - Seed Statement Classes (Quelle: Eigene Darstellung) 88
6.12 findSeedStatements - alt (Quelle: Eigene Darstellung) 88
6.13 findSeedStatements - neu (Quelle: Eigene Darstellung) 89
6.14 InterproceduralComplex - Klasse (Quelle: Eigene Darstellung) 91
6.15 ClassDMoreMethods - Klasse (Quelle: Eigene Darstellung) 92
6.16 Tracking Methode (Quelle: Eigene Darstellung) 97
6.17 interprocedural-Methode-Teilalgorithmus (Quelle: Eigene Darstellung) . . . 98
6.18 addAliasesToSlice-Methode (Quelle: Eigene Darstellung) 99
6.19 isValueUsedByInstruction-Methode (Quelle: Eigene Darstellung) 100
6.20 Teil Quellcode aus der ObjectTrackingPutMoreObjectNewInFullConstructor - Klasse (Quelle: Eigene Darstellung) 101
6.21 trackingToSliceCriterion Algorithmus (Quelle: Eigene Darstellung) 108
6.22 *getParameterNamesGroupedBySourceFiles Methode* (Quelle: Eigene Darstellung) 112
6.23 visit-Methoden der MethodVisitorParameterNames-Klasse (Quelle: Eigene Darstellung) 115

7.1 EntryPoint - Klasse (Quelle: Eigene Darstellung) 140
7.2 ObjectTrackingPutMoreStaticObject - Klasse (Quelle: Eigene Darstellung) 143

7.3 Slice-Ergebnis - SymCipherOutputStream - mit Tracking (Quelle: Eigene Darstellung) . . . 155
7.4 Slice-Ergebnis - ContentContainer - ohne Tracking (Quelle: Eigene Darstellung) . . . 160
7.5 Slice-Ergebnis - ContentContainer - mit Tracking (Quelle: Eigene Darstellung) 162

A.1 *interprocedural*-Methode (Quelle: Eigene Darstellung) . . . 170
A.2 *intraprocedural*-Methode (Quelle: Eigene Darstellung) . . . 171
A.3 *MethodVisitorParameterNames*-Klasse (Quelle: Eigene Darstellung) . . . 175
A.4 *MethodVisitorSourceLines*-Klasse (Quelle: Eigene Darstellung) . . . 178
A.5 *Cipher AllInOneClass*-Klasse (Quelle: Eigene Darstellung) . . . 184

Kapitel 1

Einleitung

1.1 Motivation

In dieser Arbeit wird auf einen existenten Java- beziehungsweise Android-Slicer aufgebaut, welcher in mehreren Bachelor- und Masterarbeiten bei Dr. Karsten Sohr weiterentwickelt wurde. Ursprünglich wurde dieser Slicer am Technologie-Zentrum Informatik und Informationstechnik (TZI) durch andere Abschlussarbeiten entwickelt. Aufgrund der Weiterentwicklung des Slicers sind einige Funktionalitäten nicht mehr vorhanden und/oder ersetzt worden. Dies geschah teilweise durch paralleles Arbeiten an unterschiedlichen Umsetzungen des Slicers oder deshalb, weil bestimmte Funktionalitäten zu diesen Zeitpunkt nicht mehr gebraucht worden sind. Als Beispiel dafür wurde der ursprüngliche Java-Slicer zu einem Android-Slicer durch Michael Cyl umgebaut [Nguyen, 2018].
Da Android unter anderem andere Einstiegspunkte als ein Java-Programm hat, musste der Java-Slicer verändert werden und "verlor" auf diesem Wege einige Java-spezifische Funktionalitäten. Jan Möhlmann erweiterte diesen Android-Slicer so, dass dieser auch wieder Java-Programme slicen konnte[Möhlmann, 2020]. Ein Beispiel für das Verschwinden und spätere Ersetzen einer Funktionalität ist hier die Objektverfolgung der zu suchenden Methode (Seed-Statement). Detailliert wird auf diese Punkte im Kapitel 3 "Verwandte Arbeiten" eingegangen. In diesem Kapitel werden die einzelnen Arbeiten von (Philip Nguyen, Alim Kerimov, Michael Cyl und Jan Möhlmann) genauer fokussiert. Der Slicer soll später an Bibliotheken benutzt werden können, welche unter anderem auch die Kryptografie benutzen. Damit gewährleistet werden kann, dass der Slicer für solch einen Einsatz tauglich ist, muss dieser auf seine Funktionalitäten evaluiert werden.

1.2 Ziele der Arbeit

Wie im Kapitel Motivation bereits angedeutet, wird ein Ziel dieser Arbeit darauf ausgerichtet sein, die Arbeiten der vier Personen genauer in den Blick zu nehmen, um verschwundene oder ersetzte Funktionalitäten zu identifizieren. Die unterschiedlichen einzelnen Slicer werden dabei zu einem zusammengefasst, indem die verschwundenen Funktionalitäten wieder

eingebaut werden, sofern diese wirklich genötigt werden. Zusätzlich wird untersucht, ob es noch Funktionalitäten gibt, welche noch nicht umgesetzt worden sind. Die aufgetretenen Problematiken der Vorarbeiten werden präzise aufgelistet und es wird versucht, diese durch mögliche Ansätze zu lösen. Durch eigene Tests, Benchmark Tests und das Umbauen von vorhandenen Beispielen aus früheren Arbeiten zu Tests soll die Funktionalität des Slicers evaluiert werden. Insbesondere sollen dadurch die verschwundenen und nicht realisierten Funktionalitäten implementiert und verifiziert werden. Zum Schluss wird der Slicer anhand einer öffentlichen Bibliothek evaluiert, um auch hier die Funktionalität des Slicers zu verifizieren.

1.3 Aufbau der Arbeit

Im Kapitel 1 "Einleitung" wurden die Motivation und die Ziele dieser Arbeit konkretisiert. Das Grundlagen Kapitel 2 liefert die wichtigen Hintergrundinformationen zum Slicing (Abschnitt 2.1) und über die WALA-Bibliothek (Abschnitt 2.2). Diese Informationen sind für das Verstehen des vorhandenen Slicers und der Arbeiten der Vorarbeiten unerlässlich.
Die Abschlussarbeiten der Vorarbeiten werden im Kapitel 3 "Verwandte Arbeiten" genauer aufgeführt und beschrieben. Zusätzlich wird in diesem Kapitel die verschwundenen (Abschnitt 3.7) und ersetzten Funktionalitäten (Abschnitt 3.8) aufgelistet.
Im Kapitel 4 wird detailliert auf die Vorgehensweise der Implementierung und der Evaluation eingegangen. Es wird darauf ausgeführt, welche methodischen Ansätze gewählt worden sind.
Auf Basis der gewonnenen Informationen der Vorarbeiten werden in Kapitel 5 "Anforderungen" die neuen und überarbeiteten Anforderungen aufgelistet.
Das Kapitel 6 "Implementierung" beschreibt die Änderungen und die neuen Algorithmen, welche für die Anforderungen umgesetzt wurden.
Die Evaluation, welche im Kapitel 7 vorgenommen wird, überprüft die Korrektheit der Änderungen an den Slicer mithilfe von unterschiedlichen Testfällen.
Zum Schluss werden im Kapitel 8 "Fazit" die Arbeit und deren Ergebnisse resümierend zusammengefasst. Insbesondere wird auch ein Ausblick auf spätere Arbeiten formuliert.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen für das Slicing dargelegt, zusätzlich wird auf das WALA-Framework eingegangen. Die Grundlagen werden an dieser Stelle kürzer gehalten und nur das Nötigste für das Verständnis dieser Arbeit wird erklärt, weil der Fokus dieser Arbeit auf dem Zusammenführen der Vorarbeiten liegt. Deswegen wird für vertiefende Informationen an den entsprechenden Stellen auf die entsprechenden Arbeiten, sowie auf die Vorarbeiten verwiesen.

2.1 Programmanalyse & Slicing

Bei der Programmanalyse wird zwischen zwei Arten differenziert. Zu einem gibt es die statische Analyse und zum anderen die dynamische Analyse. Bei der statischen Analyse wird der Quellcode des Programms untersucht, welcher dann meistens abstrahiert wird. Dabei wird das Programm nicht ausgeführt. Die dynamische Analyse entspricht hingegen dem "Gegenteil". Das Programm wird bei seiner Ausführung untersucht. Die Ausführung muss dabei öfters vorgenommen werden, denn das Programm wird mit unterschiedlichen Parameters ausgeführt. Laut [Ernst, 2003] gibt es aber zwischen den beiden Analyseverfahren mehr Gemeinsamkeiten, als auf den ersten Blick vermutet wird. Beide Arten haben ihre Vor- und Nachteile, daher könnte eine Kombination aus beiden Verfahren vorteilhaft sein. [Ernst, 2003]
Das Slicing-Tool, welches durch die Vorarbeiten entwickelt wurde, beruht auf der statischen Analyse, und zwar mit dem Ziel, bestimmte Kontrollflüsse innerhalb des Quellcodes zu finden.
Slicing bedeutet, den Quellcode des Programms auf einen Teil-Quellcode zu verkleinern. Dieser Teil-Quellcode, welcher auch Slice genannt wird, muss aber die gleiche Verhaltensweise wie der ursprüngliche Quellcode aufweisen und somit eigenständig sein. Die Idee und die erste Technik für das Slicen stammen von [Weiser, 1984]. Neuere Techniken benutzen die Abhängigkeitsgraphen, welche in den nächsten Abschnitten genauer fokussiert werden. Für das Slicen werden die Kontrollflüsse des Programms ermittelt und anhand eines oder mehrerer "Slicing criterions" (Anweisungen im Programm) werden nur bestimmte Kontrollflüsse aus dem Quellcode genommen [Weiser, 1984]. Im Folgenden wird das "Slicing

criterion" nur noch als "Seed-Statement" bezeichnet. Dies liegt, darin begründet, dass die Vorarbeiten diesen so benannt hatten. Mit einem fertigen Slice könnte dann z.B. ein Experte diesen auf eine sicherheitsrelevante Perspektive untersuchen.
Mit der in der Abbildung 2.1 gezeigten Darstellung wird das Slicing an einem abstrakten Beispiel veranschaulicht.

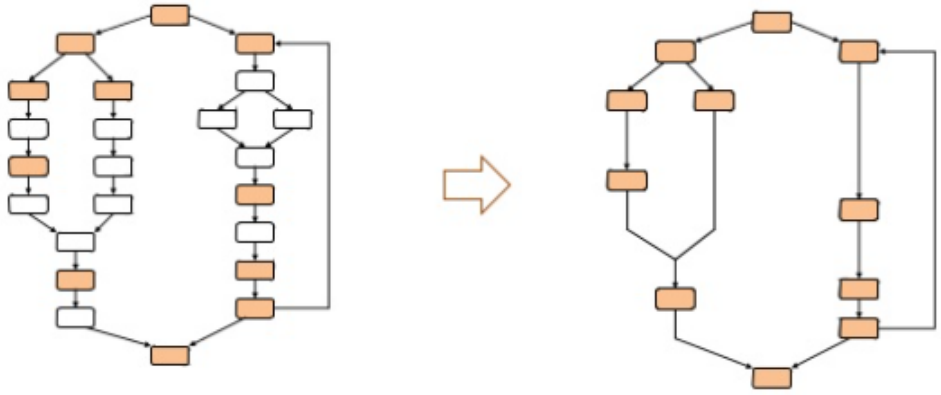


Abbildung 2.1: Abstraktes Beispiel des Slicings [Jain, 2014, Seite 4]

In der Abbildung ist das Seed-Statement nicht markiert. Dieses könnte in diesem Beispiel entweder der Block/Anweisung ganz oben oder ganz unten sein, und zwar in Abhängigkeit, in welche Richtung das Slicing ausgeführt worden ist. Dabei wird zwischen *Backward*- und *Forward*-Slicing unterschieden [Koschke, 2009]. Bei beiden Arten wird versucht, unterschiedliche Informationen aus dem Quellcode zu generieren. Beim *Backward*-Slicing wird geschaut, wie man auf das Seed-Statement kommt. Daher stellt sich die Frage: Welche Anweisungen benötigt werden, bis das Seed-Statement erreicht wird. Im Gegensatz zum *Forward*-Slicing wird die Frage gestellt, welche Anweisungen vom Seed-Statement aus beeinflusst werden. An dem Beispielprogramm von Koschke [Koschke, 2009], welches in Abbildung 2.2 zu sehen ist, sind zwei Seed-Statements ausgewählt:
Zu einem *sum :0 =;* welches für das *Forward*-Slicing verwendet wird und zum anderen *write(product);*, welches für das *Backward*-Slicing benutzt wird. Daher würden folgende Fragen gestellt werden:

- Was beeinflusst die Anweisung *sum :0 =;*?
- Wie kommt es zu den Wert *write(product);*?

In Abbildung 2.3 ist das Ergebnis nach dem *Backward*-Slicing und in Abbildung 2.4 das Ergebnis nach dem *Forward*-Slicing zu erkennen. Alle durchgestrichenen Anweisungen sind im Slice nicht enthalten, daher spielen diese keine Rolle bei der Beantwortung der Fragen.

```
1 read (n);
2 i:= 1;
3 sum := 0;          -- Was beeinflusst diese Anweisung?
4 product := 1;
5 while i <= n loop
6             sum := sum + i;
7             product := product * i;
8             i := i + 1;
9 end loop;
0 write (sum);
1 write (product); -- Wie kommt es zu diesem Wert?
```

Abbildung 2.2: Beispielprogramm für das Slicen [Koschke, 2009, Seite 6]

Am Beispiel des *Backward*-Slicings ist zu erkennen, dass die Variable *sum* überhaupt keinen Einfluss auf das *product* ausübt, weswegen es im Slice auch nicht enthalten ist. Im *Forward*-Slicing von *sum* sind nur die Zeilen enthalten, in denen die Variable *sum* genutzt wird.

Backward Slice (write (product)):

```
1 read (n);
2 i:= 1;
3 sum := 0;
4 product := 1;
5 while i <= n loop
6             sum := sum + i;
7             product := product * i;
8             i := i + 1;
9 end loop;
10 write (sum);
11 write (product); -- Wie kommt es zu diesem Wert?
```

Abbildung 2.3: Beispielprogramm für das Slicen - Backward [Koschke, 2009, Seite 8]

Forward Slice (sum := 0):

```
1 read (n);
2 i:= 1;
3 sum := 0;          -- Was beeinflusst diese Anweisung?
4 product := 1;
5 while i <= n loop
6             sum := sum + i;
7             product := product * i;
8             i := i + 1;
9 end loop;
0 write (sum);
1 write (product);
```

Abbildung 2.4: Beispielprogramm für das Slicen - Forward [Koschke, 2009, Seite 9]

Das WALA-Framework, welches für diese Arbeit genutzt wird, bietet beide Möglichkeiten des Slicings an. Dies wird durch die Methoden *computeBackwardsSlice* und *computeForwardSlice* ermöglicht. Zurzeit wird vom Slicer nur das *Backward*-Slicing genutzt.

2.1.1 Intraprozedural vs. Interprozedural

Beim Slicing muss noch zwischen intra- und interprozedural differenziert werden. Beim intraprozeduralen Slicing werden nur Anweisungen innerhalb einer Methode "gesliced". Dazu kann der bereits erwähnte Kontrollflussgraph benutzt werden. Außerdem ist es auch möglich, einen Kontrollabhängigkeitsgraphen, Datenabhängigkeitsgraphen oder Programmabhängigkeitsgraphen zu benutzen. Bei einem Programmabhängigkeitsgraphen handelt es sich dabei um eine Kombination aus dem Kontroll- und dem Datenabhängigkeitsgraphen. Auf allen diesen Graphen ist ein intraprozedurales Slicing möglich.
Beim interprozeduralen Slicing werden die Abhängigkeiten zwischen den Methoden untersucht. Als Graph für das interprozeduralen Slicing wird der Systemabhängigkeitsgraph (SDG) benutzt, welcher das ganze System bzw. Programm abdeckt. Zusätzlich gibt es noch den Aufrufgraphen, im welchen die Aufrufe zwischen den Methoden dargestellt werden.
Die einzelnen genannten Graphen werden in den nächsten Abschnitten genauer präzisiert.

2.1.2 Kontrollflussgraph

Der Kontrollflussgraph, im Englischen "Control Flow Graph" (CFG), ist ein gerichteter Graph. In diesem Graphen werden die Reihenfolgen der Anweisungen einer Methode eines Programms aufgelistet. Ein Knoten im Graphen entspricht einer Anweisung und eine Kante einem Kontrollfluss. Ein Kontrollfluss gibt an, nach welcher Anweisung welche andere Anweisung ausgeführt wird. Programmiersprachabhängige Schlüsselwörter, wie z.B. *for*, *while*, *throw* und *if*, lassen eine sequentielle Abfolge von Anweisungen nicht zu. Diese werden im Graphen in besonderer Weise ausgedrückt. In Abbildung 2.5 sind einige dieser Schlüsselwörter und die dazugehörigen Knoten und Kanten abgebildet. Zusätzlich gibt es einen *START/ENTRY*- und einen *EXIT*-Knoten, welche keine Anweisungen darstellen, sondern nur den Anfang und das Ende der Methode markieren [Koschke, 2010] [Weiser, 1984].
Der Kontrollflussgraph für den gezeigten Quellcode aus Abbildung 2.2 ist in Abbildung 2.6 zu sehen.

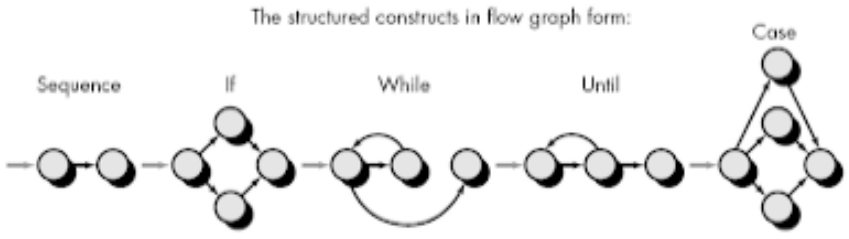


Abbildung 2.5: Schlüsselwörter und deren Kontrollflüsse [1000sourcecodes, 2014]

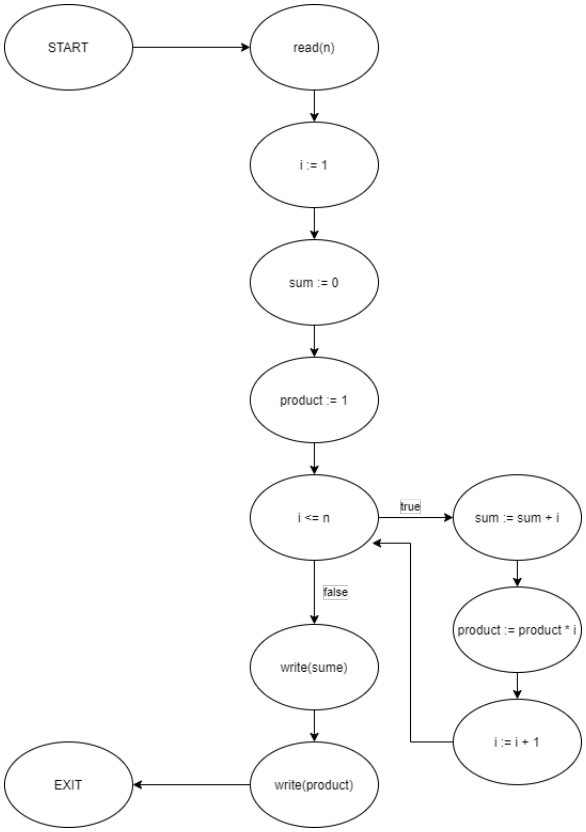


Abbildung 2.6: CFG für das Beispielprogramm (vgl. [Robschink, 2005, Seite 12])

2.1.3 Kontrollabhängigkeit

Eine Kontrollabhängigkeit zwischen zwei Anweisungen X und Y besteht dann, wenn die Anweisung X entscheidet, ob die Anweisung Y ausgeführt wird. Dies passiert in der Regel bei den bereits genannten Schlüsselwörtern. Koschke formulierte dies so:

"Intuitiv gesprochen, hat ein Knoten B Kontrolle über die Ausführung eines anderen Knotens X im KFG, wenn erstens X von B aus erreicht werden kann und zweitens ein Weg im KFG von B zum Exit-Knoten existiert, der X auslässt." (Zitat: [Koschke, 2018] [Kerimov, 2019])

Genauer wird die Kontrollabhängigkeit mithilfe der *Dominanz* bzw. der *Post-Dominanz* bestimmt, siehe [Koschke, 2010, Seite 25 - 30] sowie [Robschink, 2005, Seite 12-13].

Diese Abhängigkeiten können als Kanten in den CFG eingetragen werden. In Abbildung 2.7 ist das Beispiel aus Abbildung 2.5 durch rote Kanten erweitert worden. Diese roten Kanten zeigen, dass drei Knoten kontrollabhängig von $i <= n$ sind.

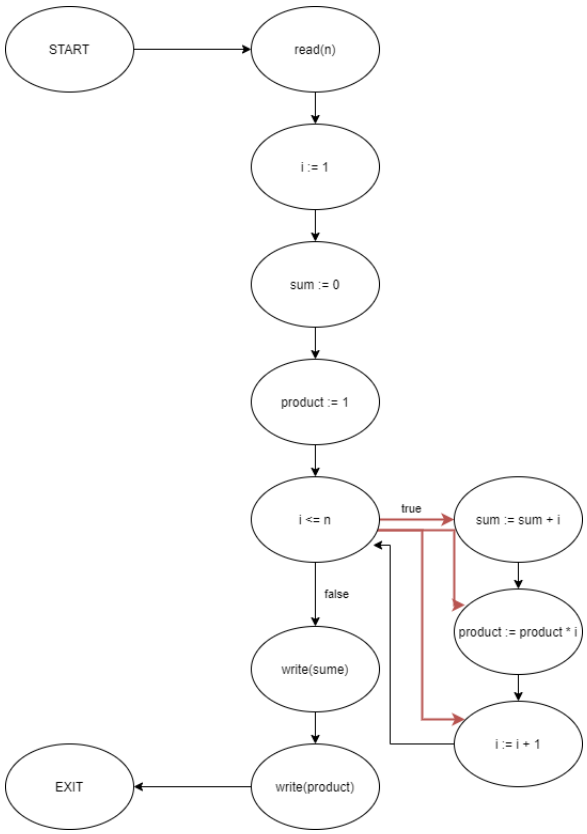


Abbildung 2.7: CFG mit Kontrollabhängigkeiten (vgl. [Robschink, 2005, Seite 14])

Zusätzlich ist es möglich, nur aus dem CFG und den Kontrollabhängigkeiten einen Kontrollabhängigkeitsgraphen, im Englischen "Control Dependence Graph" (CDG), zu ermitteln. Die Knoten im Graphen bleiben die gleichen wie in dem CFG und die Kanten fungieren nur als Kontrollabhängigkeiten. Der Startknoten kontrolliert dabei alle untergeordneten Knoten, welche selbst nicht von anderen Knoten dominiert werden. Daher sind diese kontrollabhängig von Startknoten [Cyl, 2019]. Das heißt, alle Kanten, außer den roten Kanten, fallen weg und zusätzlich gibt es Kanten vom Startknoten zu den beschriebenen untergeordneten Knoten. In Abbildung 2.8 ist ein CDG zu dem genannten Beispiel zu sehen.

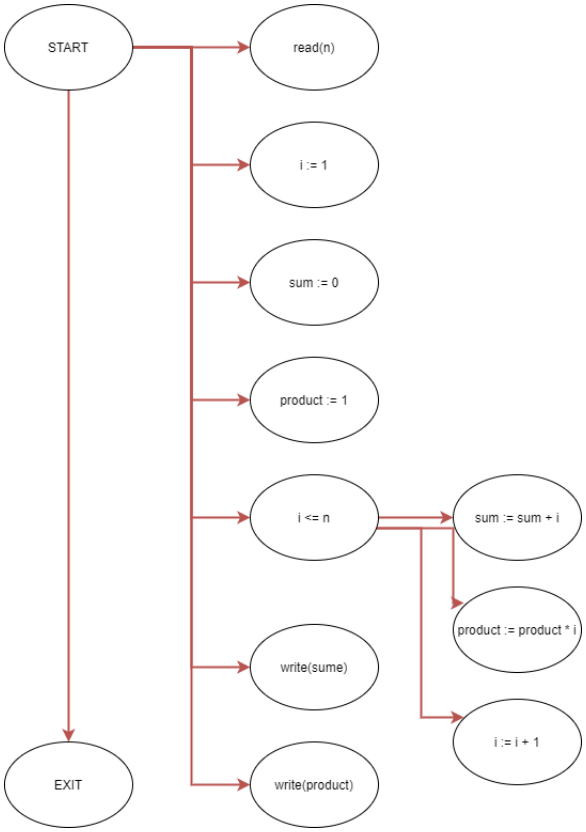


Abbildung 2.8: CDG für das Beispielprogramm (vgl. [Cyl, 2019, Seite 12])

2.1.4 Datenabhängigkeit

Für eine Datenabhängigkeit, welche die Abhängigkeit zweier Anweisungen durch Variablen ausdrückt, müssen erst die *Use*- und *Set*-Mengen einer Anweisung erklärt werden. *Set* gibt an, welche Variablen durch die Anweisung erzeugt bzw. definiert werden. Manchmal wird *Set* auch *Def* genannt. Im Gegensatz dazu gibt *Use* an, welche Variablen durch eine Anweisung benutzt werden. Formal ausgedrückt, ist eine Datenabhängigkeit Folgendes:

"Ein Knoten m *ist dann* datenabhängig *von einem Knoten* n *genau dann, wenn*

1. *es einen Pfad von* n *nach* m *im CFG gibt*
2. *es eine Variable* v *gibt mit* $v \in set(n)$ *und* $v \in use(m)$ *und*
3. *für alle Knoten* $k \neq n$ *auf diesem Pfad* p *gilt, dass* $v \notin set(k)$*"*

(Zitat: [Nguyen, 2018, Seite 31-32])

Einfacher ausgerückt, liegt eine Datenabhängigkeit zwischen zwei Anweisungen m und n dann vor, wenn in der Anweisung n die Variable v definiert wird und in der Anweisung

m die Variable genutzt wird, ohne dass zwischen den beiden Anweisungen die Variable überschrieben wird.
Eine Datenabhängigkeitsbeziehung zwischen zwei Anweisungen kann in einer von drei Beziehungen vorliegen: [Koschke, 2010]

1. Set-Use Beziehung (Anweisung n setzt die Variable v, die von der Anweisung m benutzt wird.)
2. Use-Set Beziehung (Anweisung n benutzt die Variable v, welche von der Anweisung m überschrieben wird.)
3. Set-Use Beziehung (Anweisung n setzt die Variable v, welche von Anweisung m überschrieben wird.)

Ein Beispiel für die Datenabhängigkeiten ist in Abbildung 2.9 zu erkennen. Dort wurde der Graph aus der Abbildung 2.5 durch Kanten, welche die Datenabhängigkeiten repräsentieren, erweitert. Durch die unterschiedlichen Farben soll erkenntlich gemacht werden, dass eine Anweisung mehrere Datenabhängigkeiten zu anderen Anweisungen haben kann.

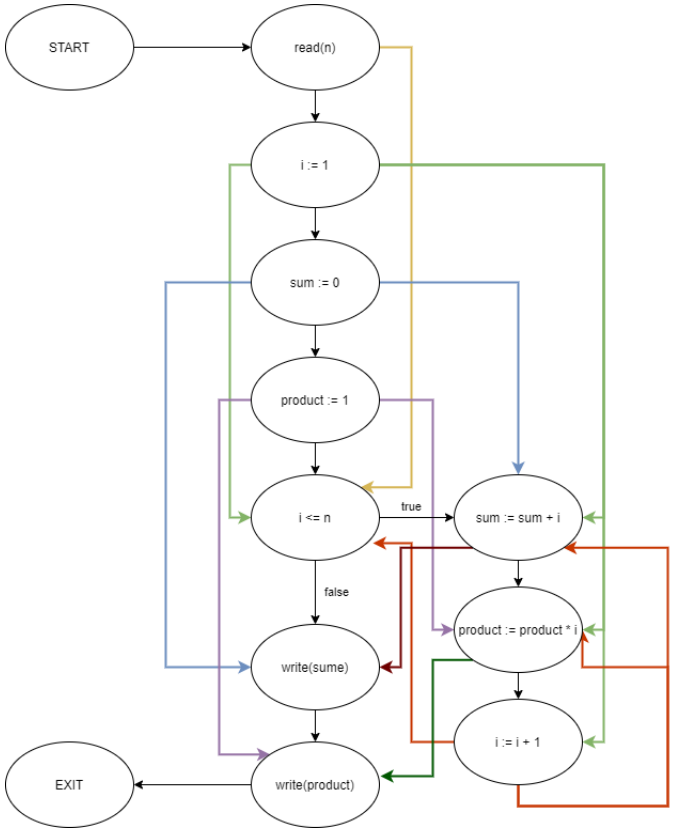


Abbildung 2.9: CFG mit Datenabhängigkeiten (vgl. [Robschink, 2005, Seite 15])

Analog zu den Kontrollabhängigkeiten kann aus den Datenabhängigkeiten ein eigenständiger Datenabhängigkeitsgraph, im Englischen "Data Dependence Graph" (CDG), generiert werden.

2.1.5 Programmabhängigkeitsgraph

Der erste Programmabhängigkeitsgraph, im Englischen "Programm Dependence Graph" (PDG), wurde von [Ottenstein and Ottenstein, 1984] vorgestellt und es handelt sich dabei, anders als der Name es andeutet, nicht um einen Graphen, welcher das ganze Programm, sondern um einen, der nur eine einzelne Methode des Programms widerspiegelt.
Ein ganzes Programm bzw. System wird mithilfe des Systemabhängigkeitsgraphen abgebildet. Der PDG ist eine Kombination aus dem CDG und dem DDG.

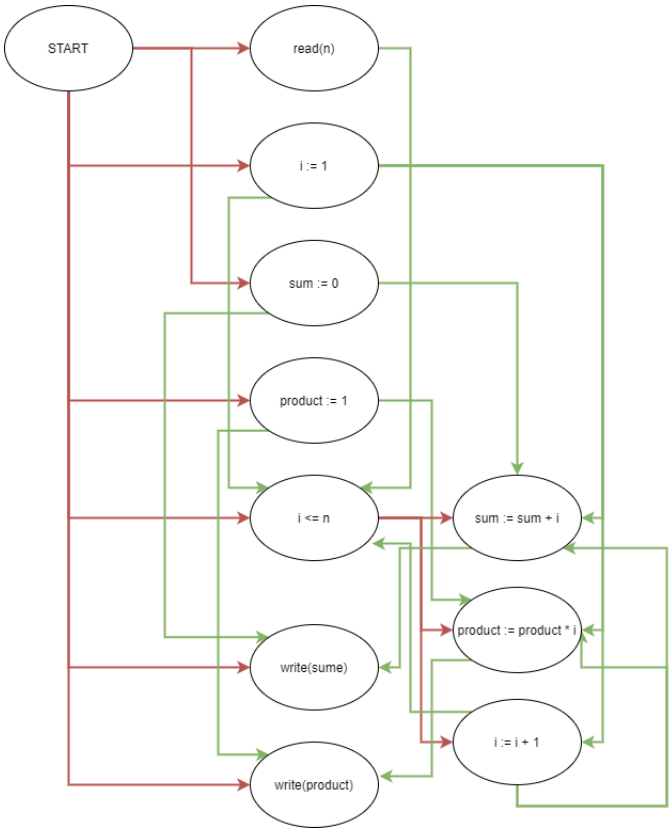


Abbildung 2.10: PDG für das Beispielprogramm (vgl. [Robschink, 2005, Seite 16])

Die Knoten sind damit weiterhin die einzelnen Anweisungen und die Kanten sind zum einen die Kontroll- sowie zum anderen die Datenabhängigkeiten der beiden einzelnen Graphen. Nur der *EXIT*-Knoten fällt weg. Ein Beispiel für einen PDG ist in Abbildung 2.10 visualisiert, indem der CDG aus der Abbildung 2.8 um die Datenabhängigkeiten erweitert worden ist. Die Kanten für die Datenabhängigkeiten sind dies Mal alle einheitlich grün markiert, weil es sonst zu unübersichtlich wäre.
Durch diesen PDG ist es nun möglich, zu verdeutlichen, warum die durchgestrichenen Zeilen in den Beispielen für das *Backward*- und *Forward*-Slicing in den Abbildungen 2.3 und 2.4 weggefallen sind. Wird durch den PDG von den Seed-Statements aus entweder rückwärts oder vorwärts travisiert, werden nicht alle Knoten im PDG besucht.

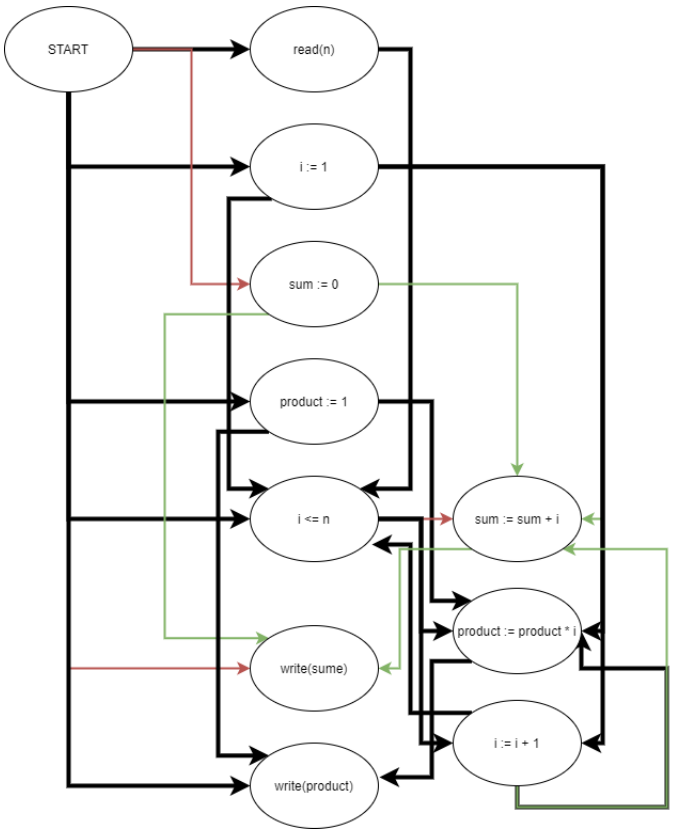


Abbildung 2.11: Backward-Slicing am PDG für das Beispielprogramm (Eigene Darstellung)

Diese nicht besuchten Knoten sind die Zeilen, welche nicht im Slice enthalten sind. In der Abbildung 2.11 sind die Rückwärts- und in Abbildung 2.12 die Vorwärts-Travisierung dargestellt. Die schwarzen dicken Kanten sind die Kanten, welche beim Travisieren besucht worden sind.

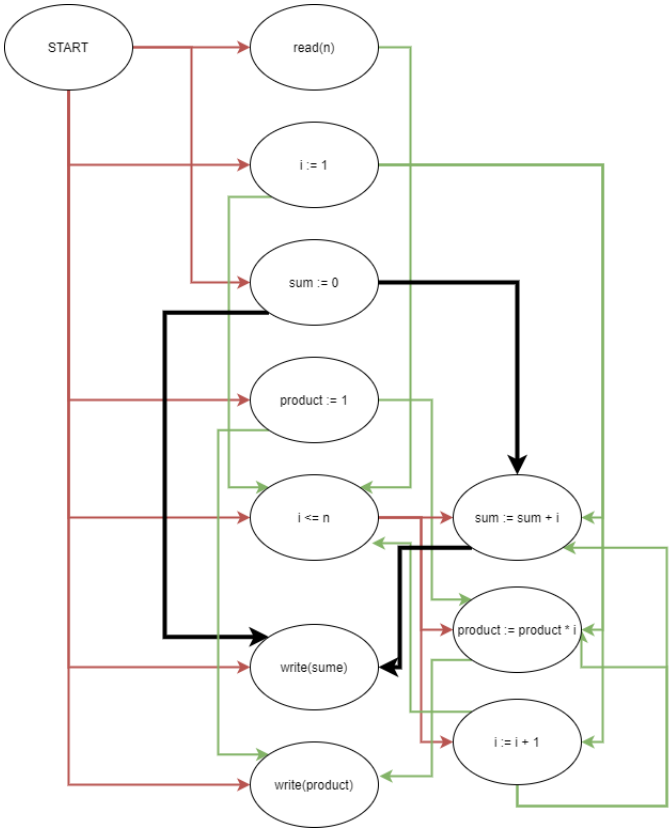


Abbildung 2.12: Forward-Slicing am PDG für das Beispielprogramm (Eigene Darstellung)

2.1.6 Zwischendarstellung SSA

Allgemein ist eine Zwischendarstellung, im englischen "Intermediate Representation" (*IR*), im Kontext des Quellcodes, erst einmal eine Abstrahierung vom Quellcode. Diese wird meistens bei der statischen Analyse vorgenommen, weil nicht direkt mit dem Quellcode gearbeitet wird. Eine Variante der Zwischendarstellung ist die "Static Single Assignment-Form" kurz *SSA*. Diese hat den Vorteil, dass die SSA-Form sehr effizient ist und dies bei der Analyse weiterhilft [Cytron et al., 1991]. Wie der Name bereits andeutet, gibt es in der SSA-Form keine mehrfachen Zuweisungen mehr. Genauer formuliert: Jede Variable darf nur einmal zugewiesen, dafür aber beliebig oft benutzt werden. Durch die Umwandlung in die SSA-Form werden die einzelnen Variablen umbenannt und erhalten dabei einen eindeutigen Namen. In Abbildung 2.13 ist ein Programmcode zu erkennen, welcher in Abbildung 2.14 in die SSA-Form gewandelt worden ist. Die Variable y wurde in *y1* und *y2* umbenannt. Damit sind diese eindeutig und jede Variable hat nur eine Zuweisung. Außerdem ist in diesem Beispiel zu erkennen, dass die Zeile *y :=1* eine unnötige Zuweisung ist. Dieses Faktum ist daran zu erkennen, dass *y1* in der SSA-Form nie benutzt wird.

```
y := 1
y := 2
x := y
```

Abbildung 2.13: Beispielprogramm ohne SSA-Form [Wikipedia, 2021]

```
y1 := 1
y2 := 2
x1 := y2
```

Abbildung 2.14: Beispielprogramm in SSA-Form [Wikipedia, 2021]

Diese Umbenennung reicht allein nicht aus, denn an den Stellen, an denen der Kontrollfluss über mehrere Abzweigungen verfügt, wie z.B. beim *if*, ist nicht mehr klar, welchen Wert eine Variable hat. Denn diese könnte entweder im *if*-Block oder im *else*-Zweig verändert worden sein. Das Problem, das durch diese Mehrfachzuweisung auftritt, wird durch eine künstliche Zuweisung durch einen ϕ-Knoten gelöst. Am ϕ-Knoten wird entschieden, welcher Wert verwendet wird. Dieser Knoten wird an der Stelle angegeben, an dem die Kontrollflüsse wieder zusammenkommen. Ein vollständiger CFG in SSA-Form ist in Abbildung 2.15 zu sehen. Dafür wurde ein anderes Beispielprogramm verwendet. Der Graph links in der Abbildung ist der CFG ohne SSA und der Graph rechts ist der CFG mit SSA.

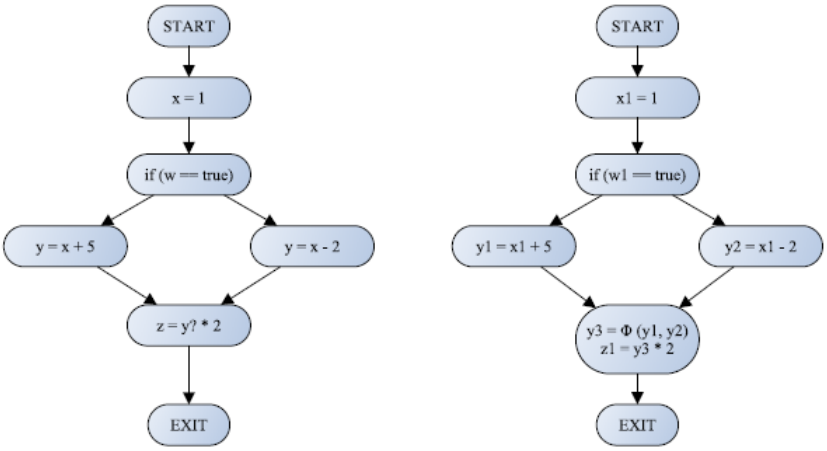
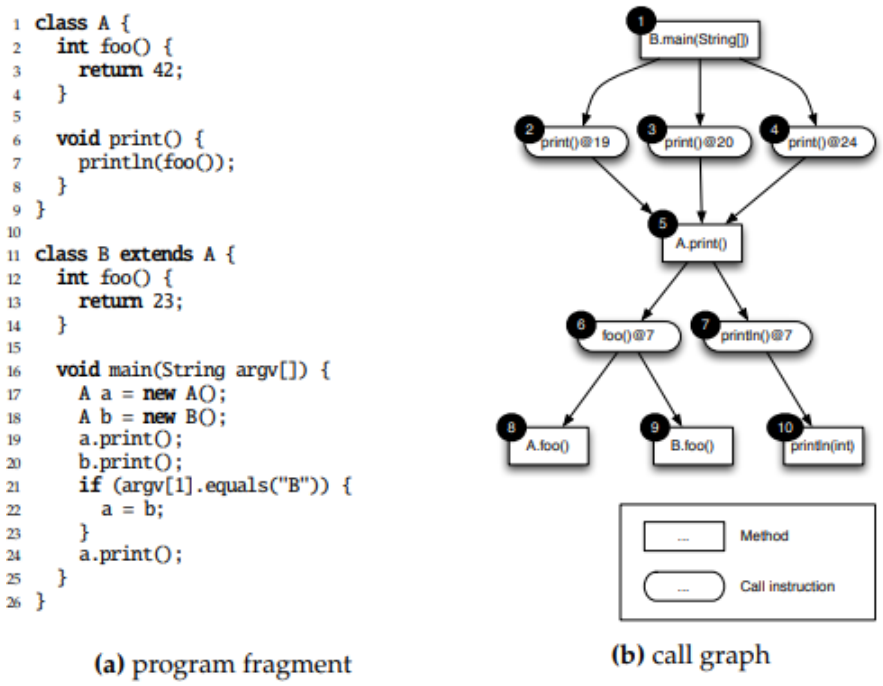


Abbildung 2.15: Ein CFG und der umgewandelte CFG in SSA-Form [Nguyen, 2018, Seite 26]

Das WALA-Framework verwendet die SSA-Form. Das Framework wandelt dafür den Java-Bytecode-Befehl in eine *SSAInstruction*-Klasse um. Für jeden Methodenaufruf sowie alle Art von Anweisungen steht eine passende Unterklasse der *SSAInstruction* zur Verfügung. z.B. wird jeder Methodenaufruf in eine *SSAAbstractInvokeInstruction*-Klasse umgewandelt. Nur für Zuweisungen gibt es in WALA keine *SSAInstruction*. [WALA, 2021b].

2.1.7 Aufrufgraph

Der Aufrufgraph, im Englischen Call Graph, ist ein gerichteter, interprozeduraler Kontrollflussgraph. Dieser soll als eine Annäherung alle Methoden auflisten, welche während der Programmausführung aufgerufen werden würden. Seine Knoten sind die Methoden, welche aufgerufen werden und die Kanten im Graphen verbinden die Caller (Aufrufender) mit dem Callee (Aufgerufenen). Diese Beziehung zwischen Aufrufenden und Aufgerufenen wird als sich "Caller-Callee Beziehung"bezeichnet. In Abhängigkeit von der Programmiersprache kann der Graph Zyklen oder nur einzelne ausgehende Kanten besitzen. Interessant wird es bei Programmiersprachen, wie z.B. Java, welche dynamisches Binden erlauben oder, anders formuliert, Vererbung. Durch Vererbung entsteht ein Problem dahin gehend, dass im Graphen nicht mehr genau unterschieden werden kann, welche Methode welche andere Methode aufgerufen hat. Am besten zeigt sich dies an dem Beispiel von [Graf, 2016] in Abbildung 2.16.



(a) program fragment **(b)** call graph

Abbildung 2.16: "Ein Programmfragment (a) und sein Aufrufgraph (b). Die rechteckigen Knoten entsprechen den Methoden, die runden Formen entsprechen den Aufrufanweisungen."[Graf, 2016, Seite 61]

Es gibt die Klasse *A* und die Klasse *B*, welche von *A* erbt und die Methode *foo* überschreibt. Die *main*-Methode der *B*-Klasse bildet den Einstiegspunkt. Im Graphen ist zu erkennen, dass die *main*-Methode drei Mal die *print*-Methode aufruft, und zwar in den Zeilen 19, 20 und 24. Diese "*Call instructions*" weisen auf die *print*-Methode der *A*-Klasse hin. Diese ruft wiederum *foo()* und *println()* auf. Interessant ist an dieser Stelle, dass die

Call instruction foo() auf die Methoden *foo* in den Klassen *A* und *B* zeigt. Dies passiert, weil *foo* in *B* überschrieben wird. Aber genau hier liegt das Problem begründet. Der *print* Aufruf in Zeile 19 wird auf der Klasse *A* ausgeführt, aber im Aufrufgraphen ist an dieser Stelle nicht eindeutig, ob der Aufruf auf *A* oder *B* vorgenommen worden ist. Dies gilt analog auch für die Zeile 20. Nur für die Zeile 24 ist dies richtig, denn über den *if*-Zweig ist nicht mehr eindeutig zu erkennen, ob *print* auf *A* oder *B* aufgerufen worden ist.
Um dieses Problem zu lösen, werden meistens die Methoden-Knoten mit den Subgraphen dupliziert. Dabei werden diese Subgraphen so angepasst, dass diese der Ausführung des Programms entsprechen. In Abbildung 2.17 ist der Aufrufgraph aus der Abbildung 2.16 zu sehen. An diesem wurde der *A.print()*-Knoten dupliziert und dessen Subgraphen angepasst.

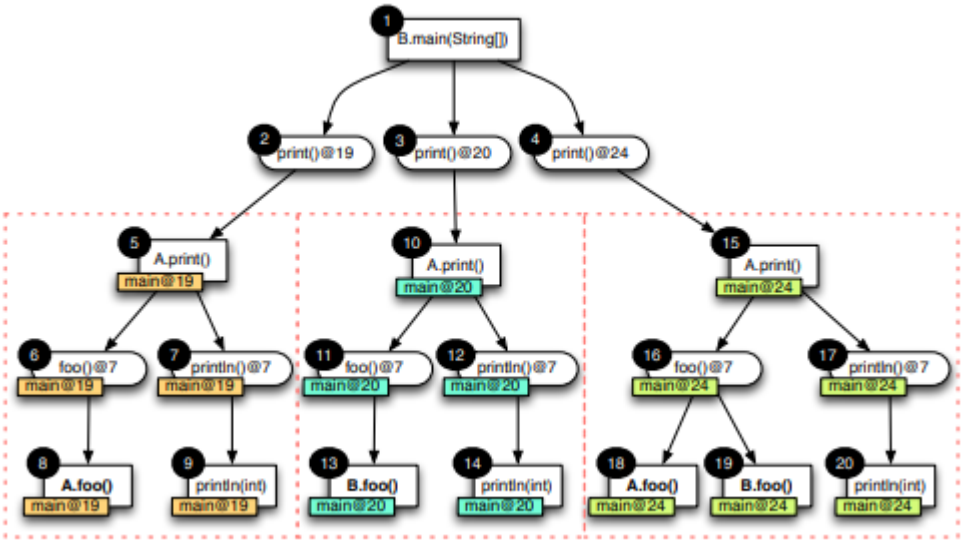


Abbildung 2.17: print-Methoden, geklont im Aufrufgraphen [Graf, 2016] Seite 62

Es ist zu erkennen, dass die *print*-Methode in Zeile 19 nur noch die Methode *foo* in der Klasse *A* aufruft. Damit ist das Problem der Ungenauigkeit gelöst, aber es gibt jetzt Methoden-Knoten, welche doppelt und dreifach vorkommen, wie z.B. *A.foo()* oder *println(int)*. Je größer ein Graph ist, desto größer wird auch der sein, in dem die Knoten dupliziert worden sind. Dies führt zu einem anderen Problem, nämlich dem, dass der Graph zu schnell zu groß werden könnte, sodass dieser nicht mehr in sinnvoller Zeit berechnet werden kann. Daher können diese Knoten zusammengelegt werden, ohne dass die Genauigkeit darunter leidet, bzw. darauf muss beim Duplizieren anderer Knoten geachtet werden [Graf, 2016].
An dieser Stelle muss zur Vollständigkeit erwähnt werden, dass die Informationen für einen Aufrufgraphen über eine "*Point-To-Analyse*" erzeugt wird. Da eine *Point-To-Analyse* sehr komplex ist und im weiteren Verlauf dieser Arbeit keine weitere Rolle spielt, wird auf die Arbeit von Cyl [Cyl, 2019] verwiesen, welcher die *Point-To-Analyse* sehr gut erklärt hat.

2.1.8 Systemabhängigkeitsgraph

Der Systemabhängigkeitsgraph, im Englischen "System Dependency Graph" (SDG), besteht aus vielen PDGs. Diese werden erst einmal durch Kanten namens "Call Sites" miteinander verbunden. Die Verbindung wird vom Knoten der aufgerufenen Methode (Caller) zum Entry-Knoten der passende Methode (Callee) gezogen. Zusätzlich werden neue künstliche Knoten erzeugt, welche das Verhalten der Eingabe- sowie der Rückgabeparameter abdecken. Diese Knoten und einige neue Kantenarten machen den SDG sehr komplex. Die künstlichen Knoten werden in die zwei Arten "*Formal*" und "*Actual*" differenziert. *Actual*-Knoten gehören zum Caller und die *Formal*-Knoten zum Callee. Des Weiteren werden diese Knoten wiederum in "*in*" sowie "*out*"-Knoten unterteilt. Zusammengefasst existieren vier neue Knotenarten:

- *Actual-In* beschreibt die Eingabeparameter am Caller
- *Actual-Out* beschreibt die Ausgabeparameter am Caller
- *Formal-In* beschreibt die Eingabeparameter am Callee
- *Formal-Out* beschreibt die Ausgabeparameter am Callee

Verbunden werden die *Actual-In* mit dem *Formal-In* über eine "*Parameter-In*"-Kante sowie die *Formal-Out* mit dem *Actual-Out* über eine "*Parameter-Out*"-Kante. Zusätzlich werden noch Kontroll- sowie Datenabhängigkeitskanten, wie in den Abschnitten 2.1.3 und 2.1.4 beschrieben, zwischen den alten und neuen Knoten gezogen. Dafür dürfen diese nur zwischen dem PDG des Callers und den *Actual*-Knoten sowie zwischen dem PDG des Callees und den *Formal*-Knoten gezogen werden.
Zuallerletzt werden sogenannte "*Summary-Edges*" zwischen den *Actual-In*- sowie *Actual-Out*-Knoten gezogen. Diese sind besonders wichtig für das spätere Slicing, denn dadurch wird verhindert, dass unnötige Abstiege bei den Methoden vorgenommen werden. Daher werden Ressourcen, wie Zeit und Platz, eingespart [Koschke, 2009, Graf, 2016].
Ein Beispiel für einen SDG ist in Abbildung 2.18 zu erkennen. Links in der Abbildung sind das benutzte Programmbeispiel und rechts der Systemabhängigkeitsgraph dazu zu sehen.

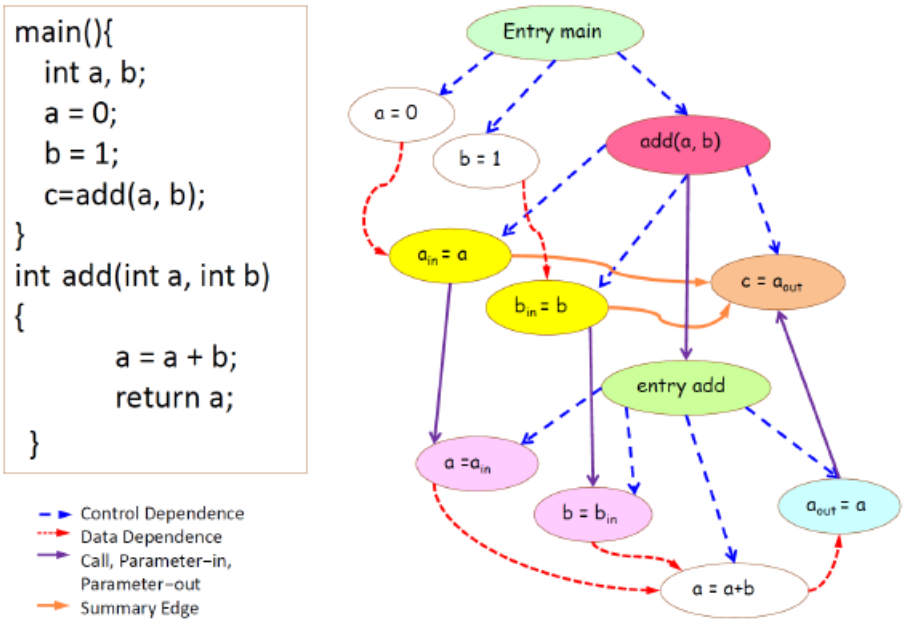


Abbildung 2.18: SDG mit einen Beispielprogramm [Jain, 2014, Seite 24]

2.2 WALA

Im Folgenden wird zunächst kurz darauf eingegangen, was Wala ist und woher es stammt. Danach wird grob erklärt, wie WALA funktioniert und wie dies im Aktuellen Slicer umgesetzt worden ist. Zum Schluss werden die einzelnen Datenabhänigkeits- sowie Kontrollabhängigkeitsoptionen aufgeführt.

2.2.1 Das Framework

Die "T.J.Watson Libraries for Analysis", kurz WALA, wurde im Jahre 2006 im DOMO Forschungsprojekt des IBM T.J. Watson Research Center entwickelt. Später wurde WALA unter der "Eclipse Public License" als Open Source Projekt veröffentlicht. Seitdem wird WALA durch die Community betreut [WALA, 2016]. Das letzte Update des WALA-Frameworks wurde am 01.04.2020 auf die Version 1.5.5 [WALA, 2020] vorgenommen.
Teile der Hauptfunktionen von WALA sind folgende: [WALA, 2016]

- Unterstützung der Sprachen Java und JavaScript
- Analyse von Java-Typ-Systemen und Klassenhierarchien
- Konstruktion der Pointeranalyse und des Aufrufgraphen
- SSA-basierte Register-Transfer-Sprache IR
- Eine Bytecode-Instrumentierungsbibliothek (Shrike) und eine dynamische Laufzeit-Instrumentierungsbibliothek für Java (Dila)

2.2.2 Funktionsweise

WALA erstellt aus dem bereitgestellten Java-Bytecode einen Slice. Dazu werden zuerst von WALA eine Klassenhierarchie sowie interprozedurale Datenflüsse erstellt. Auf dieser Basis werden die Einstiegspunkte, im Englischen Entrypoints, ermittelt. Mithilfe der Entrypoints berechnet WALA den Aufrufgraphen. Weil vorher Entrypoints ermittelt worden sind, deckt der Aufrufgraph nur einen bestimmten Teil des Programms ab. Im nächsten Schritt wird der SDG berechnet, dies passiert mithilfe der Point-To-Analyse. Als letzten Schritt wird ein Slice anhand des Seed-Statements erstellt. Entweder wird dabei das *Forward*- oder das *Backward*-Slicing verwendet [Nguyen, 2018].

2.2.3 Ablauf im Slicer

Der Ablauf und die Verwendung von WALA im Slicer haben sich durch die unterschiedlichen Vorarbeiten immer wieder verändert. Der erste Ablauf wurde von Nguyen [Nguyen, 2018] erstellt. Dem Folgenden ist der Ablauf nach der Implementierung (Kapitel 6) zu entnehmen:

1. Aufbau des Analysescopes
2. Erzeugen der Klassenhierarchie
3. Ermittlung der Entrypoints
4. Ermittlung der Pointeranalyse
5. Berechnung des Aufrufgraphen
6. Ermittlung der Methoden für die Seed-Statements
7. Ermittlung der Seed-Statements
8. Berechnung des Systemabhängigkeitsgraphen (SDG)
9. Ausführung des Objekt- und Parameter-Trackings
10. Ausführung des "Tracking to Slice Criterion"
11. Erzeugen des Slices mithilfe des SDG durch die gefundenen Seed-Statements
12. Finden von lokalen Variablen Namen und Attribut-Namen
13. Rekonstruktion des Java-Quelltextes.

Für vertiefende Informationen zur Ermittlung des Analysescope, zur Klassenhierarchie, zur Pointeranalyse und zum Aufrufgraphen wird auf die Arbeit von Nguyen [Nguyen, 2018, Seite 60-64] verwiesen.
Um Redundanzen zu vermeiden, werden die restlichen einzelnen Punkte des Ablaufs an

dieser Stelle nicht präzise erklärt. Dieses passiert im Laufe dieser Arbeit entweder im Kapitel 3 "Verwandte Arbeiten" oder im Kapitel 6 "Implementierung." Als Beispiel wird der ursprüngliche Ablauf der Ermittlung der Entrypoints im Kapitel 3 "Verwandte Arbeiten" auf der Seite 29 dargelegt. Dieser beruht auf den Arbeiten von Gulmann [Gulmann, 2014], Detmers [Detmers, 2016] und Nguyen [Nguyen, 2018]. Dieser ursprüngliche Slicer wurde durch Cyl [Cyl, 2019] zu einem Android-Slicer umgebaut, welcher durch Möhlmann [Möhlmann, 2020] wieder erweitert worden ist. Dieser konnte dann auch Java-Programme slicen. Dadurch ging die Ermittlung der Entrypoints verloren und es wurde nur noch die WALA-eigene Methode verwendet.

2.2.4 WALA-Einstellungsmöglichkeiten durch Parameter

Beim Erzeugen des SDGs werden eine Datenabhängigkeits- sowie eine Kontrollabhängigkeitsoptionen übergeben. Diese geben an, wie detailreich die jeweiligen Abhängigkeiten im SDSs ausgeprägt sein sollen. An dieser und an sehr vielen weiteren Stellen hat sich gezeigt, dass die Dokumentation von WALA nicht wirklich gut ist. Als Beispiel wurden die einzelnen Optionen nur sehr grob dokumentiert und wurden nicht vollständig aufgelistet [WALA, 2021c].
Detmers führte erste Evaluationen durch, um die Auswirkungen der Optionen besser zu verstehen [Detmers, 2016]. Diese wurden auch weitergeführt in den Arbeiten von Nguyen [Nguyen, 2018] und Cyl [Cyl, 2019], selbst wenn nicht das Hauptaugenmerk der Arbeiten auf diese Optionen gerichtet war.
Zusätzlich gibt es noch die Reflexion-Optionen, welche beim Erstellen des AnalyseScopes angegeben werden können, diese kümmern sich um die Java-Reflexionen. Zum Schluss gibt es noch die CFA-Optionen, welche für die Berechnung der Point-To-Analyse genutzt werden. Weil diesen beiden Optionen in dieser Arbeit keine weitere Beachtung geschenkt wird, wird für die einzelnen Optionen auf die Arbeit von Cyl verwiesen [Cyl, 2019, Seite 69-71].
Es wird als Datenabhängigkeitsoption *FULL* oder *NO_HEAP* sowie für die Kontrollabhängigkeitsoption *NO_EXCEPTIONAL_EDGES* empfohlen. Auf den Grund dafür sowie auf alle einzelnen Abhängigkeitsoptionen wird im Folgenden eingegangen.

2.2.4.1 Datenabhängigkeitsoptionen

FULL
Anhand dieser Option werden alle Datenabhängigkeitskanten verfolgt.
Vorteil: Das Ergebnis der Analyse ist am genausten.
Nachteil: Es werden dafür viel Rechenkapazitäten benötigt.

NO_BASE_PTRS
Datenabhängigkeitskanten, die indirekten Speicherzugriff durch *base pointer* definieren,

werden ignoriert. Unter anderem wird z.B. dadurch die Initialisierung von Arrays und deren Rückgaben ignoriert.

NO_HEAP
Datenabhängigkeitskanten, die auf dem Heap oder den dynamischen Speicher zeigen, werden ignoriert. Dies betrifft unter anderem Zugriffe auf Objekt-Attribute, Instanzvariablen und Konstruktoren. Diese Option wird von den Vorarbeiten dann als die beste Option qualifiziert, wenn es zu Speicherproblemen bei der Analyse kommt. Die Heap-Abhängigkeiten benötigen viel Speicher, daher besitzt diese Option den Vorteil, dass sie weniger Speicher verbraucht, aber dafür können die Ergebnisse ungenauer bzw. unvollständig sein.

NO_EXCEPTIONS
Diese Option ignoriert alle Datenabhängigkeitskanten zu und von *throw*- bzw. *catch*-Anweisungen.

REFLECTION
Bei dieser Option werden *NO_BASE_PTRS* und *NO_HEAP* miteinander kombiniert, aber es werden zusätzlich Datenabhängigkeitskanten verfolgt, welche aus *checkcast*-Anweisungen entstehen. Diese Option funktioniert analog wie die Reflexion-Option, siehe [Cyl, 2019, Seite 70].

NONE
Anhand dieser Option werden keine Datenabhängigkeitskanten verfolgt.
Vorteil: Es werden dafür fast keine Rechenkapazität gebraucht.
Nachteil: Das Ergebnis der Analyse ist am ungenausten.

Durch die Kombination von *NO_BASE*, *NO_HEAP* und *NO_EXCEPTIONS* können mehrere Optionen gleichzeitig gewählt werden, wie z.B. *NO_BASE_NO_HEAP*.

2.2.4.2 Kontrollabhängigkeitsoptionen

FULL
Anhand dieser Option werden alle Kontrollabhängigkeitskanten verfolgt. Im Gegensatz zu der Datenabhängigkeitsoption *FULL* erzeugt diese Option den rechenintensiven SDG. Es gibt noch einen weiteren Punkt, warum von dieser Option abgeraten wird, dabei handelt sich um *False Positives*, welche aber in Wirklichkeit keine sind. Dieses Problem wird genauer bei der nächsten Option detailliert beschrieben.

NO_EXCEPTIONAL_EDGES
Diese Option wird in den offiziellen Dokumenten von WALA nicht aufgeführt [WALA, 2021b, WALA, 2021c] und wurde durch die Vorarbeiten im *JavaDoc* der *Slicer*-Klasse gefunden. Durch die Benutzung dieser Option werden alle Kontrollabhängigkeitskanten ignoriert, welche durch "*Exceptions*" resultieren. Es hat sich gezeigt, dass ein Benutzer beim Blick

auf das Slice-Ergebnis nicht wusste, ob eine Anweisung wegen der Abhängigkeit des Seed-Statements oder durch eine *Exception*-Kontrollabhängigkeit entstanden ist.
Gerken und Detmers haben dies fälschlicherweise als "False Positives" verstanden [Gerken, 2015] [Detmers, 2016].
Nguyen zeigte aber, dass bei Ausschluss dieser *Exception*-Kontrollabhängigkeit daher die Aktivierung von NO_EXCEPTIONAL_EDGES, sehr sinnvoll ist [Nguyen, 2018]. Die Rechenzeit wird nicht nur deshalb stark reduziert, weil eigentlich jede Methode, z.b. eine *NullPointerException* auslösen könnte, sondern weil auch nur noch Anweisungen im Slice-Ergebnis enthalten sind, welche Abhängigkeiten zum Seed-Statement haben. Sollten aber diese *Exception*-Kontrollabhängigkeiten auch untersucht werden, sollte eine andere Option gewählt werden, um Rechenkapazität einzusparen.

NO_INTERPROC_EDGES
Alle interprozeduralen Kanten werden ignoriert. Dadurch wird die Analyse auf eine Methode beschränkt. Zum Zeitpunkt der Arbeit von Nguyen konnte diese Option deshalb nicht benutzt werden, weil WALA diese nicht implementiert hatte. Trotzdem wurde diese zur Vollständigkeit hier mit aufgelistet. Ob diese Option jetzt funktioniert, wurde nicht untersucht.

NONE
Anhand dieser Option werden keine Kontrollabhängigkeitskanten verfolgt.
Vorteil: Es werden dafür fast keine Rechenkapazitäten gebraucht.
Nachteil: Das Ergebnis der Analyse ist am ungenausten.

2.3 Zusammenfassung des Kapitels

In diesem Kapitel wurden zuerst die Programmanalyse sowie das Slicing definiert. Insbesondere wurden auf die einzelnen Graphenarten für das intraprozedurale- sowie das intraprozedurale-Slicing näher eingegangen. Im zweiten Teil des Kapitels wurde das WALA-Framwork vorgestellt. Dazu wurden die unterschiedlichen Daten- und Kontrollabhängigkeitsoptionen konkretisiert.

Kapitel 3

Verwandte Arbeiten

Im Folgenden werde die verschiedenen Vorarbeiten und offene Punkte hierzu beschrieben.

3.1 Gulmann, Gerken, Detmers

Erste Slicing-Implementierungen mithilfe der WALA-API stammten von Gulmann und Gerken. Diese waren eher prototypisch und dienten als Machbarkeitsbeweise. Gulmann konzentrierte sich auf die Android-Systemservices [Gulmann, 2014] und Gerken auf die Java-Enterprise Anwendungen [Gerken, 2015]. Detmers hat den WALA-Slicer bezüglich der Anwendbarkeit auf sicherheitskritische Java-Programme evaluiert. Unter anderem hat die Evaluierung eine optimale Parameterwahl für WALA geliefert [Detmers, 2016].
Auf die unterschiedlichen Parameter wurde im Kapitel 2.2 "WALA" eingegangen.

Wichtige Features:

- Einstiegspunkte (Entrypoints)
  Ein wichtiger Teil für diese Arbeit ist die Implementierung der Einstiegspunkte von Gerken und Detmers, welche nicht nur unterschiedlich sind, sondern auch später im vorhandenen Slicer nicht mehr existent waren [Gerken, 2015][Detmers, 2016].

- JavaParser
  Detmers benutzte die JavaParser Bibliothek, um aus den gefundenen Slice-Zeilen die entsprechenden Java-Zeilen mithilfe des Quellcodes zu rekonstruieren. Der Parser funktioniert, grob erklärt, so:
  Wenn die Slice-Zeile in einer Methode ist, werden die erste und die letzte Zeile der Methode sowie die passende Java-Zeile für die Slice-Zeile verwendet und diese in einer externen Java-Datei abgespeichert. Das Gleiche gilt für die erste und die letzte Zeile der Java-Klasse [Detmers, 2016].

Folgende aufgetretene Problematiken sind zu verzeichnen:

- Exklusionsdatei
  Damit der Slicer für Sicherheitsanalysen auf Java-Applikationen geeignet ist, testeten Gerken und Detmers ihren Slicer an einer Enterprise Java Bean (EJB) Anwendung mit 250 Zeilen Code. Diese Anwendung stellt ein fiktives Klinik-Informationssystem dar. Mit 250 Zeilen Code war die Anwendung zwar sehr klein, benutzte aber viele Fremdbibliotheken. Die Berechnungszeiten waren aber trotzdem sehr groß. Dies lag zum einem an der Auswahl der Datenabhängigkeitsoption, aber zum größten Teil lag es an der Exlusionsdatei, welche zu diesem Zeitpunkt noch zu klein war [Gerken, 2015][Detmers, 2016]. Gerken optimierte dies, indem er Pakete und Klassen der Java-Standardbibliothek hinzufügte [Gerken, 2015].
- Falsche Zeilen?
  Zusätzlich ist noch zu erwähnen, dass arithmetische Operationen im Slice erschienen, bei denen keinerlei Abhängigkeit zu dem Seed-Statement bestand. Dies konnte sich Gerken nicht erklären [Gerken, 2015].

3.2 Philip Nguyen

Die Ergebnisse von Gulmann, Gerken und Detmers sind in die Arbeit von Nguyen eingeflossen. Er baute auf den vorhandenen Arbeiten seinen "Auditor"auf, welcher das Slicen von Java-Programme ermöglichte. Der Auditor ist eine Java-Anwendung ohne GUI, welche mithilfe eines Kommandozeilen-Programms mit unterschiedlichen Parametern aufgerufen werden kann. Dieses Vorgehen ermöglicht es unterschiedliche Ergebnisse für das gleiche Java-Programm zu generieren. Gespeichert wurden zum einem die rekonstruierten Java-Dateien und zum anderen die Ergebnisse des Slicing Prozesses, wie zum Beispiel den Call Graphen [Nguyen, 2018].

Folgende neue Features beinhaltet diese Unterarbeit:

- Analyseergebnis
  Das Analyseergebnis sowie der Slicing-Prozess werden in einem separaten Ausgabeordner korrekt abgespeichert. Dazu gehören unter anderem auch die reduzierten rekonstruierten Java-Dateien [Nguyen, 2018].
  - Optional kann die Pointeranalyse, der Callgraph oder der SDG als Text oder als PDF abgespeichert werden.
  - Zwischenergebnisse werden in der Kommandozeile angezeigt.
- Konfigurationsdatei
  Es wurde eine Konfigurationsdatei hinzugefügt. Diese und die Exlusionsdatei sind vom Benutzer anpassbar. Die Optionen mit Erklärung der Konfigurationsdatei von Nguyen sind in der Abbildung 3.1 zu finden. Für die Erklärung der einzelnen Einstellungsmöglichkeiten wird auf die Arbeit von Nguyen verwiesen [Nguyen, 2018].

Die einzelnen Einstellungen werden aber im Laufe dieser Arbeit dann erklärt, wenn diese vonnöten sind. Vorneweg werden noch drei Informationen angegeben, welche direkt aus der Tabelle nicht zu entnehmen sind:

- Beim *src_callee_class* kann entweder nur die Klasse oder der vollständige Paketpfad angegeben werden.
- Es existiert eine vorgefertigte Liste für die Datenabhängigkeits- und Kontrollabhängigkeitsoptionen.
- Die Option *java_source_folder* ist in der Abbildung 3.1 nicht vorhanden. Durch diese Option gibt der Benutzer den Pfad zum originalen Quelltext an. Nur mit dem Quelltext können die Java-Klassen erfolgreich rekonstruiert werden. Zusätzlich soll der Ordner rekursiv durchlaufen werden, um alle Java-Dateien zu finden.

Option	Beispiel	Beschreibung
jar	test.jar	Das zu analysierende Java-Programm
mainclass	Lde/pkg/Classname	Die zu untersuchende Klasse zur Ermittlung der Entrypoints
only_public_entry	true	Einschränkung der Entrypoint-Algorithmen (true=Gulmann, false=Detmers)
src_caller	doRSAEncryption	Die aufrufende Methode (Caller)
src_callee	doFinal	Die aufzurufende Methode (Callee) bzw. das Slice-Kriterium
multiple_caller	true	Mehrere Caller erlauben
src_callee_class	Ljavax/crypto/Cipher	Die Klasse des Objektes, auf die der Callee-Methode aufgerufen wird
datadep_opts	NO_HEAP	WALA Datenabhängigkeitsoption
controldep_opts	FULL	WALA Kontrollabhängigkeitsoption
pdfview_exe	$PATH/acrobat.exe	Der Pfad zum PDF-Viewer
dot_exe	$PATH/dot.exe	Der Pfad zur DOT Anwendung von GraphViz
create_pdfs	false	Generelle PDF-Erzeugung aktivieren
sdg_pdf	false	SDG Grapherzeugung aktivieren
output_dir	slice_output	Der Ausgabepfad für alle Ergebnisse
split_java_output	true	Slice in einzelne Javadateien trennen
advanced_mode	true	Erweitertes Slicing mit Objektverfolgung
object_inst_methods	getInstance	Liste von Instanziierungsmethoden

Abbildung 3.1: Konfigurationsdatei [Nguyen, 2018, Seite 91]

- Einstiegspunkte (Entrypoints)
  Durch die Angabe der *mainclass* wird das System unterschiedliche Algorithmen benutzen, um Einstiegspunkte zu finden.
  Wie in der Abbildung 3.2 zur Methode *findEntryPoints()* zu erkennen ist, benutzt die Methode im schlimmsten Fall fünf unterschiedliche Algorithmen, um Einstiegspunkte zu finden. Unter anderem werden dort die Methoden von Gulmann/Gerken und

Detmers verwendet. Erst dann, wenn dann keine Einstiegspunkte gefunden worden sind, wird *null* zurückgegeben.

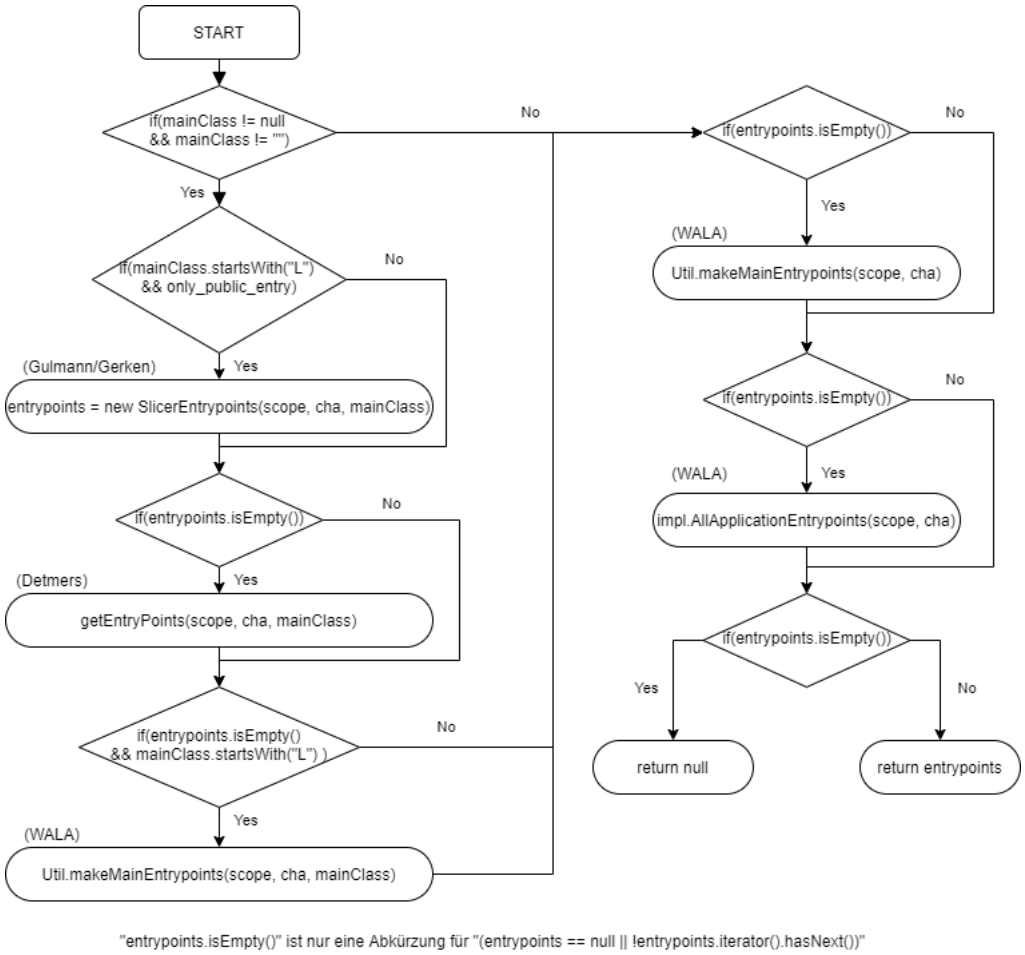


Abbildung 3.2: Programmablauf *findEntryPoints()* (Quelle: Eigene Darstellung)

- Objektverfolgung
  Ist der *advanced_mode* eingeschaltet, wird das Slicing mit Objektverfolgung ausgeführt. Damit ist das Objekt gemeint, welches auf dem die Methode *sr_callee* beziehungsweise das Seed-Statement ausgeführt wird. Durch die Objektverfolgung ist es möglich, dass alle Zeilen, in denen das Objekt vorkommt, auch als Seed-Statement verwendet werden und somit später auch im Slice-Ergebnis mit enthalten sind. Dies ist hilfreich in den Fällen, in denen bestimmte Zeilen nicht in das Slice-Ergebnis mit übernommen werden. Diese werden deshalb nicht übernommen, weil bei ihnen keine Datenabhängigkeiten oder Kontrollabhängigkeiten zu dem Seed-Statement bestehen. Somit könnten wichtige Zeilen für das Verständnis fehlen. Durch die Objektverfolgung wird dieses Problem behoben. Zusätzlich bietet die Option *object_inst_methode* die Möglichkeit an, die Instanziierungsmethoden der Objekte anzugeben. Nicht alle Objekte werden direkt mit dem *new*-Operator erzeugt. Ein *Signature* Objekt wird

zum Beispiel durch die *getInstance*-Methode erzeugt. Zu sehen ist dies in der Abbildung 3.3. Die Option unterstützt die Objektverfolgung und ist dann sehr wichtig, wenn das Objekt durch eine Methode instanziiert worden ist.

Folgende Optimierungen sind zu verzeichnen:

- JavaParser
  Der von Detmers übernommene JavaParser wurde von Nguyen komplett überarbeitet. Folgende Aspekte fehlten im JavaParser und wurden implementiert [Nguyen, 2018].
  - Die Zeilen können umgebrochen sein. Daher reicht es nicht, die erste und die letzte Zeile zu verwenden, insbesondere dann nicht, wenn z.B. die erste Zeile nicht der Kopf der Methode, sondern eine Annotation wie *@Override* ist. Das heißt, die Klassen- und Methodenköpfe sind vollständig im Slice erhalten. Daher soll der Slice, so gut wie es geht, syntaktisch richtig sein.
  - Die Konstruktoren werden nicht mehr ignoriert.
  - Die Verschachtelungen bzw. das in die Tiefe Gehen einer Methode wird berücksichtigt.

Folgende aufgetretene Problematiken sind zu verzeichnen:
Um den Auditor zu evaluieren, hat Nguyen Testfälle konstruiert, welche seine Anforderungen verifizieren sollen. Zum einen sind die kleinen Testfälle TF01 bis TF09, welche die Grundlagen des Slicers und die Richtigkeit des Parsers überprüfen, vorhanden und zum anderen gab es die Testfälle TF10 & TF11 mit öffentlichen Bibliotheken [Nguyen, 2018].

- JavaParser
  Beim Testen ist Nguyen aufgefallen, dass beim Rekonstruieren der Java-Klassen diese immer noch nicht syntaktisch korrekt sind. Umbrüche in Zeilen, in denen z.B. eine Anweisung ausgeführt oder eine Methode aufgerufen wird, sind fehlerhaft.

  "... doch fielen bei der Evaluation der beiden Anwendungen noch einige Parser-Fehler auf, die bis zur Veröffentlichung der Arbeit nicht mehr korrigiert werden konnten." (Zitat: [Nguyen, 2018])

  Obwohl er von einigen Fehler spricht, ist leider an dieser Stelle nicht zu erkennen, ob es immer der gleiche (der eben genannte Fehler) ist.

- Objektverfolgung

  - Testfall TF06 / TF07
    Im Testfall TF07 testet er die Objektverfolgung *(advanced_mode)* mit der Einstellungsoption *object_inst_methods*, welche eine Liste von Instanziierungsmethoden beinhaltet.

Unerklärlich für Nguyen war die Tatsache, dass *Signature.getInstance()* im Slice enthalten ist, und zwar selbst dann. wenn der *advanced_mode* nicht eingeschaltet war. Wenn der *advanced_mode* nicht angeschaltet ist, wird auch *object_inst_methods* ignoriert. Die Abbildung 3.3 veranschaulicht den Testfall ohne Objektverfolgung und die Abbildung 3.4 zeigt diesen mit Objektverfolgung. Die roten Zeilen stellen die Zeilen dar, welche nicht rekonstruiert worden sind. Das Merkwürdige daran ist, dass im Testfall TF06 *Chipher.getInstance()* beim ausgeschalteten *advanced_mode* nicht erhalten war. [Nguyen, 2018]
Dieses Verhalten konnte selbst nicht reproduziert werden. Dies wurde an den aktuellen Slicer sowie an seinem Auditor getestet.

```
112  15    @Override
113  16    public byte[] signData(byte[] data, String keyAlias) throws NoSuchAlgorithmException,
114  -         char[] spass = getSignPass();
115  -         char[] kpass = getKeyPass();
116  -         String ksName = getKsName();
117  -
118  17        mSignature = Signature.getInstance("SHA256withDSA", "SUN") ;
119  -         KeyStore ks = KeyStore.getInstance("Hospital");
120  -         FileInputStream ksfis = new FileInputStream(ksName);
121  -         BufferedInputStream ksbufin = new BufferedInputStream(ksfis);
122  -         ks.load(ksbufin, spass) ;
123  -         PrivateKey priv = (PrivateKey) ks.getKey(keyAlias,kpass) ;
124  -         System.out.println("output");
125  -         mSignature.initSign(priv);
126  -         mSignature.update(data);
127  18        return mSignature.sign();
128  19    }
```

Abbildung 3.3: TF07 ohne Objektverfolgung (Quelle: Eigene Darstellung)

```
112  15    @Override
113  16    public byte[] signData(byte[] data, String keyAlias) throws NoSuchAlgorithmException,
114  -         char[] spass = getSignPass();
115  17        char[] kpass = getKeyPass();
116  -         String ksName = getKsName();
117  -
118  18        mSignature = Signature.getInstance("SHA256withDSA", "SUN") ;
119  19        KeyStore ks = KeyStore.getInstance("Hospital");
120  -         FileInputStream ksfis = new FileInputStream(ksName);
121  -         BufferedInputStream ksbufin = new BufferedInputStream(ksfis);
122  -         ks.load(ksbufin, spass) ;
123  20        PrivateKey priv = (PrivateKey) ks.getKey(keyAlias,kpass) ;
124  -         System.out.println("output");
125  21        mSignature.initSign(priv);
126  22        mSignature.update(data);
127  23        return mSignature.sign();
128  24    }
```

Abbildung 3.4: TF07 mit Objektverfolgung (Quelle: Eigene Darstellung)

– Testfall 05
Auch hier wurde ein Test mit dem *advanced_mode* durchgeführt. Anders als in TF06/TF07 lag die Unklarheit nicht auf dem aufgerufenen Objekt *obj*. Denn die Methode *do_stuff()* war beim eingeschalteten *advanced_mode* vorhanden,

sondern der Parameter *value* in der main-Methode war im Slice nicht enthalten, obwohl auf diese, arithmetische Operationen ausgeführt worden sind [Nguyen, 2018]. Dies ist in der Abbildung 3.5 zu erkennen.

```
1   1  public class SimpleTest {
2      -    Integer value;
3      -    String word;
4      -
5   2       public static String entry_method(Integer anothervalue) {
6      -        String result;
7   3           SimpleTestObject obj = new SimpleTestObject();
8   4           obj.do_stuff();
9   5           Integer addvalue = SimpleTestObject.calc_param();
10  6           result = obj.important_method(addvalue, anothervalue);
11     -        return result;
12  7       }
13  8
14  9       public static void main(String[] argsv){
15     -        Integer value = 4;
16     -        value++;
17  10          String result = entry_method(value);
18     -
19     -        System.out.println("test done");
20     -        System.out.println(result);
21  11      }
22  12 }
23  13
```

Abbildung 3.5: TF05 mit Objektverfolgung (Quelle: Eigene Darstellung)

- Einstiegspunkte (Entrypoints)
  Die Auswahl der *mainclass* und die damit aufzurufende *findEntryPoints()*-Methode sind auch ausschlaggebend für die Komplexität und Ausführungszeit des Slicers. Bei größeren Java-Programmen ist es sinnvoll, weniger Einstiegspunkte, wenn nicht sogar nur einen Einstiegspunkt zu haben. Selbst mit der Angabe der *mainclass* kann es vorkommen, dass mehr als nur ein Einstiegspunkt gefunden wird. Daher muss das Finden der Einstiegspunkte optimiert werden [Nguyen, 2018].

- Dokumentation
  Der Testfall TF11 hat außerdem gezeigt, dass die vorhandenen Methoden ungenau und zu schlecht dokumentiert sind. Diese wurden insbesondere nicht genau getestet. Wenn dies verbessert werden könnte, könnten sich auch die Komplexität und Ausführungszeit verbessern. Die Fehlerbehandlung und die Dokumentation (JavaDoc) des Auditors müssen verbessert werden [Nguyen, 2018].

- Testen
  Zusätzlich wäre es sinnvoll, den Slicer anhand von Java-Programmen mit über 200.000 LoC zu evaluieren [Nguyen, 2018].

- Datenflüsse
  Die Optimierung der rechenintensiven Datenflussanalyse ist unerlässlich, damit diese Datenflüsse außerhalb der eigentlichen Anwendung verhindert werden. Bibliotheken werden bei der Analyse immer mit einbezogen, sind aber für das Rekonstruieren nicht wichtig. Dazu können Stubs für Klassen und Methoden erzeugt werden. Diese ersetzen die echten Klassen und Methoden in den Bibliotheken. Stubs sind voller nicht implementierter Programmcodes. Ob dies den gewünschten Effekt bringt, muss evaluiert werden [Nguyen, 2018].

3.3 Alim Kerimov

Kerimovs Arbeit baut auf der von Nguyen auf. Durchdie Evaluierung wurde nachgewiesen, dass der Slicer ausreichend Parametrisierungsmöglichkeiten für statische Sicherheitsanalysen bietet. Die Evaluierung wurde mit Testfällen mithilfe der Java Cryptography Architecture (JCA) durchgeführt. JCA beinhaltet unter anderem Cipher und Signature. Außerdem evaluierte er den Slicer anhand der Bibliotheken Apache Shiro und Google Tink [Kerimov, 2019].

Folgende Optimierungen sind zu verzeichnen:

- Paketebene
  Anders als die üblichen Optimierungen des Slicers, welche dem Zweck dienten, diesen hinsichtlich Schnelligkeit oder Genauigkeit zu verbessern, hat Kerimov erst einmal den Sourcecode des Slicers umstrukturiert. Es wurden neue Pakete geschaffen, alte weggeworfen und der nicht gebrauchte Sourcecode entfernt. In Abbildung 3.6 ist die alte Paketstruktur zu erkennen. In dem *test*-Package waren die meisten Klassen vorhanden. Im Sinne von Test-Klassen waren aber keine vertreten. In der Klasse *CustomSlicer* wurde das Slicing mithilfe der Methode *doSlicing* vorgenommen. Alle wichtigen Aufrufe auf die Wala-API sowie andere wichtige Methoden waren in dieser Klasse vertreten. Durch die Umstrukturierung hat Kerimov den Sourcecode und die Pakete übersichtlicher gestaltet. Der Abbildung 3.7 ist die neue Paketstruktur zu entnehmen. Die Klasse *CustomSlicer* ist zwar auch noch für die Hauptmethode *doSlicing* zuständig, aber andere wichtige Methoden oder API-Zugriffe wurden in andere Klassen verlegt [Kerimov, 2019].

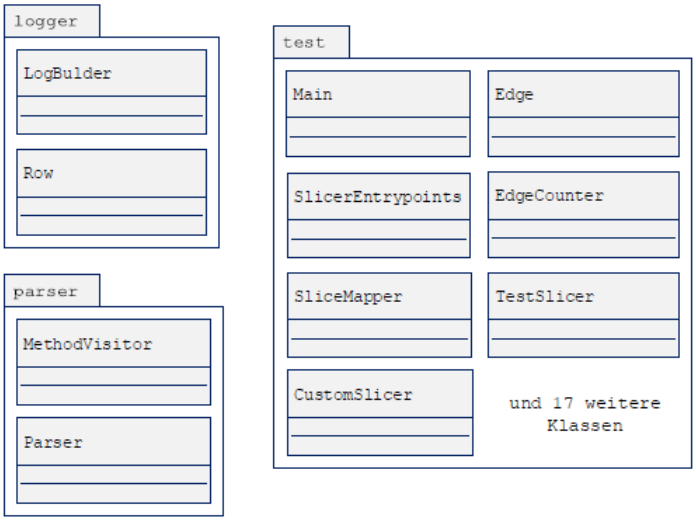


Abbildung 3.6: Alte Paketstruktur des Slicers [Kerimov, 2019, Seite 32]

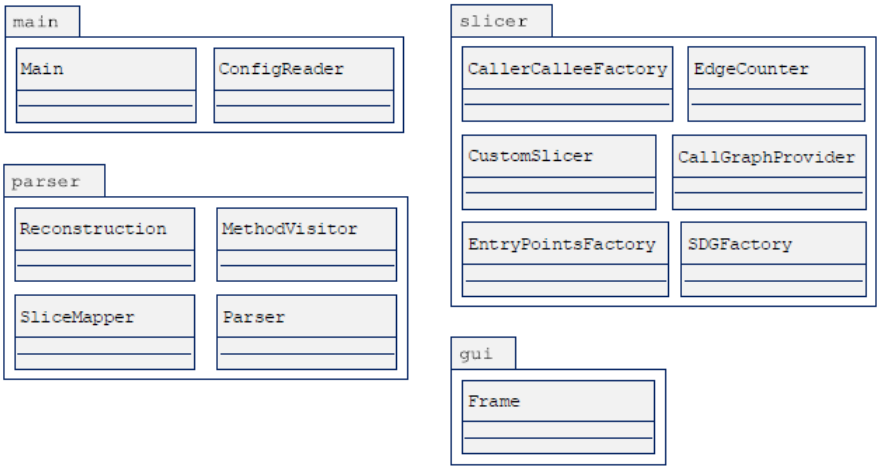


Abbildung 3.7: Neue Paketstruktur des Slicers [Kerimov, 2019, Seite 32]

Wichtig ist auch, dass die Codeoptimierungen durchgeführt worden sind. Auch diese dienten dazu, dass der Sourcecode übersichtlicher wurde.
In diesem Zusammenhang wurden die Methoden, Variablen und Parameter anhand der Java-Codekonvention umbenannt. Unter anderem wurden auch geeignete Zugriffsmodifizierer und Schlüsselwörter, wie z.B. ”final” hinzugefügt [Kerimov, 2019]

- Konfigurationsdatei
  Vorher konnte der Slicer nur Java-Properties einlesen und benutzen. Durch die Einführung der Klasse *ConfigReader* wurde der Reader erweitert. Zusätzlich war es möglich, dass die Konfigurationsdatei nicht nur als Java-Property, sondern auch

als *JavaScript Object Notation* Format (JSON) eingelesen werden konnte. Außerdem konnten die Parameter auch über ein Kommandozeilen-Programm eingegeben werden. Diese wurden zwar für das Slicing benutzt, haben aber die Konfigurationsdatei nicht verändert [Kerimov, 2019].

- GUI
  Über die Konfiguration war es möglich, dass ein kleines Fenster beim Ausführen des Slices mit angezeigt wird. In dem Fenster wurden die Konsolenausgaben angezeigt. Außerdem war es über den Reiter *Options* möglich, die Konfigurationsdatei oder den Dateiordner des Slice-Ergebnis zu öffnen. Als Letztes konnte der Slice erneut gestartet (wiederholt) werden [Kerimov, 2019].

Folgende aufgetretene Problematiken sind zu verzeichnen:

- Rekonstruktion Java-Klasse
  Innere- sowie anonyme-Java-Klassen werden ignoriert. Beim Kompilieren werden die inneren-Klassen in dem Format *ÄußereKlasse$InnereKlasse.class* abgespeichert. Das Rekonstruieren ignoriert dieses Format und gibt immer nur *ÄußereKlasse.class* aus. Deswegen kann der Quellcode nicht gefunden werden.
  Außerdem fehlten am Ende von Catch-Blöcken teilweise geschweifte Klammern [Kerimov, 2019].

- Fehlende Informationen im Slice
  Klassenattribute werden generell nicht betrachtet und werden im Slice-Vorgang ignoriert [Kerimov, 2019].
  Wie später im Kapitel 7 "Evaluation" beschrieben wird, stimmt diese Aussage nur teilweise, denn aus der Arbeit wird auch nicht ersichtlich, welche Attribute genau gemeint sind.

- Konfigurationsdatei
  Nicht nur die Konfigurationsdatei musste Kerimov anpassen, sondern auch die WALA-eigene Konfigurationsdatei *primordial.txt* [Kerimov, 2019].
  Dieses Problem konnte nicht reproduziert werden und wurde auch in keiner weiteren Arbeit der Vorarbeiter benannt.

Ausblick:
Folgende mögliche Ansätze sah Kerimov für die nächsten Arbeiten an den Slicer gerichtet [Kerimov, 2019]:

- Statt des zurzeit benutzen Backward-Slicens könnte auch das Forward-Slicen genutzt werden. Die dafür benötigte Änderung wäre nur minimal. Ob das Forward-Slicen nützlich wäre, müsste dann evaluiert werden.

- Umfassende Evaluation aller Kontroll- und Datenabhängigkeitsoptionen. Zum Bei-

spiel wurden die Optionen *REFLECTION* und *NO_BASE_PTRS* der Datenabhängigkeitsoption noch nicht gründlich evaluiert.

- Bessere Dokumentation durch JavaDoc.

- Falls der Quellcode nicht oder in der falschen Version vorliegt, kann entweder das Rekonstruieren überhaupt nicht funktionieren oder fehlerhaft sein. Eine mögliche Lösung bestünde darin, den Quellcode durch einen Decompiler aus dem Bytecode zu rekonstruieren. Das könnte auch das Problem lösen, dass Innere-Klassen nicht mit rekonstruiert werden.

3.4 Michael Cyl

Als Grundlage für Cyl dienten die Arbeiten von Gulmann und Nguyen. Da Cyl parallel zu Kerimov gearbeitet hat, floss nicht viel aus dessen Arbeit in seine Arbeit ein. Unter anderem lag dies vielleicht auch darin begründet, dass Kerimov sich um Java-Anwendungen gekümmert hat. Durch iteratives Vorgehen und ständige Evaluation entstand aus einem initialen Prototyp der Android-Slicer. [Cyl, 2019]

Folgende neue Features beinhaltet diese Unterarbeit:

- Android
  Statt Java-Anwendungen ist es nun möglich, Android-Ressourcen zu slicen. Damit sind nicht die Applikationen (Apps) gemeint, sondern dass grundlegende System von Android [Cyl, 2019].

- Server-Client-Architektur
  Cyl entwickelte für den Android-Slicer eine Webapplikation, welche es dem Benutzer mit einer leicht verständlichen GUI ermöglichte, Slices in Auftrag zu geben. Somit wird kein Kommandozeilenprogramm mehr benötigt. Serverseitig arbeitet ein javabasierter REST-Server, welcher für das Verwalten und Ausführen der Slices zuständig ist. Mithilfe der MongoDB, werden alle Slices sowie die Einstellungen in einer NoSQL-Datenbank abgespeichert. [Cyl, 2019]
  Statt des normalen relationalen Ansatzes besitzt eine NoSQL-Datenbank keine starren Tabellenschemtaa mehr.
  MongoDB speichert die Einträge als JSON ab [MongoDB, 2020]. Der Client besteht dabei aus einer Angular-Webapplikation und stellt verschiedene Benutzeroberflächen bereit, auf diese wird genauer eingegangen.
  Auf der Startseite der Webapplikation, welche der Abbildung 3.8 zu entnehmen ist, sind zunächst ein kleiner Willkommenstext und wichtige Links zu WALA zu finden [Cyl, 2019].

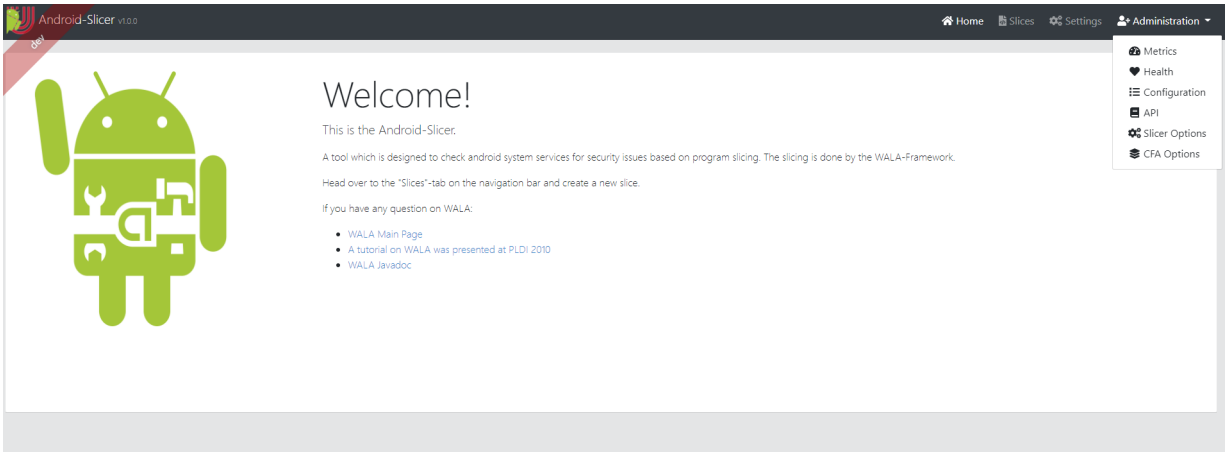


Abbildung 3.8: Startbildschirm Android-Slicer (Quelle: Eigene Darstellung)

Oben rechts sind vier Tabs (Home, Slices, Settings und Administration) zu sehen. Auf die Settings- und Administration-Tabs wird später detailliert eingegangen. Der Home-Tab führt zum Startbildschirm zurück. Unter *"Slices"* werden alle Slices aufgelistet, die in Auftrag gegeben worden sind. Zum Beispiel werden in Abbildung 3.9 zwei Slices aufgeführt. Dabei entspricht jede Zeile einem Slice. Zu einem sieht der Benutzer, mit welchen Optionen (Android Version, Android Class Name, Entry Methods, Seed Statements, CFA Options, Reflection Options, Data Dependence Options und Control Dependence Options) die Slices ausgeführt worden sind. Am Running Symbol (Grün) ist zu erkennen, ob das Slicing noch läuft. Ist das Running Symbol schwarz, ist das Slicing fertig. Zum anderen kann ein Slice an dieser Stelle gelöscht oder angeschaut werden . Zusätzlich ist es dem Benutzer möglich über "Create a new Slice" einen neuen Slice zu erstellen.

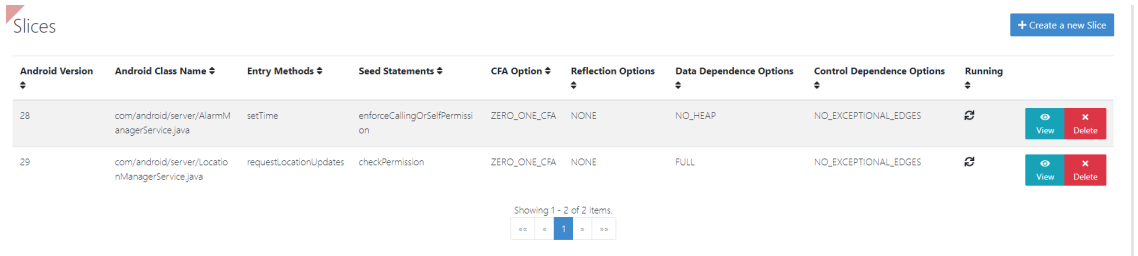


Abbildung 3.9: Slices-Tab Android-Slicer (Quelle: Eigene Darstellung)

Einen Slice erstellen und einen Slice ansehen, gehört zu den beiden Use Cases, die Cyl aufgestellt hat.[Cyl, 2019] In der Abbildung 3.10 ist diese Ansicht zu sehen, in der ein neuer Slice erstellt werden kann.

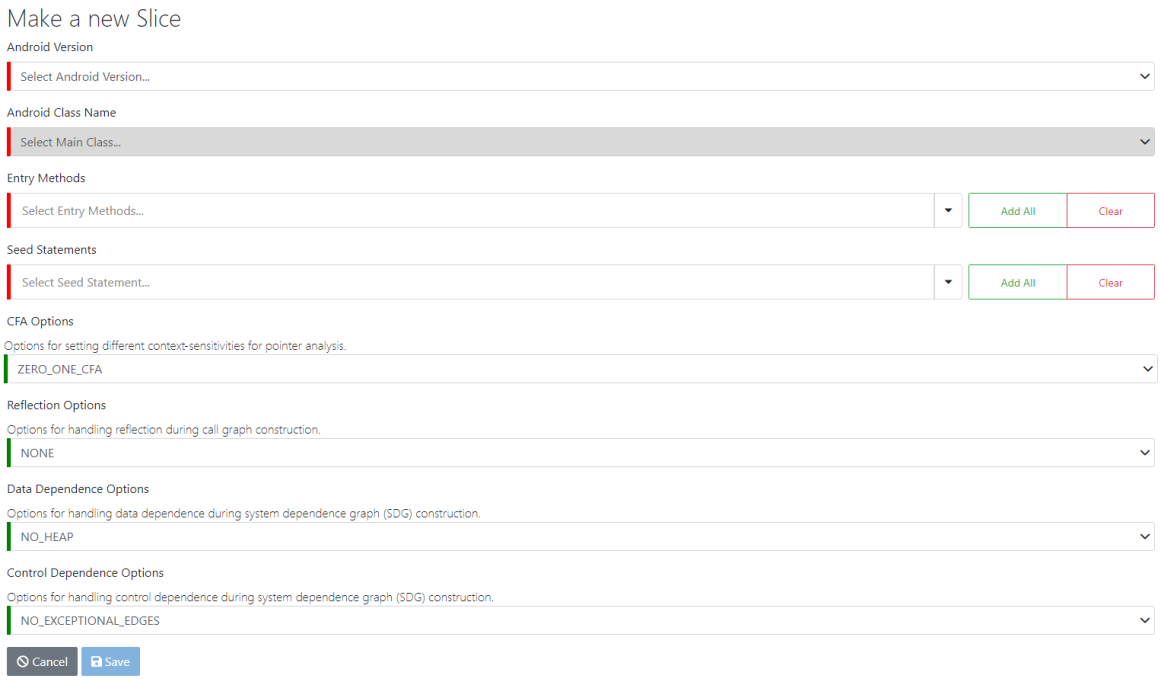


Abbildung 3.10: Ansicht zur Erstellung eines neuen Slices (Quelle: Eigene Darstellung)

An dieser Stelle wird kurz auf die einzelnen Einstellungsmöglichkeiten für einen Slice eingegangen. Jede Auswahlmöglichkeit kann über ein Dropdown-Menü ausgewählt werden. Außerdem wird durch das Schreiben in den Textfeldern automatisch die Auswahlmöglichkeit mithilfe von Autovervollständigung gefiltert.

- Android Version
  Der Benutzer kann sich eine von zwei Versionen aussuchen. Zur Wahl stehen die Android Versionen 28 und 29. Cyl stieß beim Beschaffen und Erstellen der Android Versionen auf Probleme [Cyl, 2019].
  Eingegangen wird auf diesen Punkt später unter "Folgende aufgetretene Problematiken gibt es:".
- Android Class Name
  An dieser Stelle werden nur die Klassennamen angezeigt, welche mit *ManagerService.java* enden. Diese Einstellung wurde vorher als Standardwert festgelegt. Geändert werden kann dies im Settings-Tab unter dem Namen *Android_System_Service_Regex*. Der Grund, warum nur ManagerServices angezeigt werden, besteht darin, dass diese aus sicherheitsrelevanter Perspektive heraus interessant sind. Dieser Punkt wird hier nur sehr grob angerissen. Für eine detaillierte Erläuterung, wie welche Komponenten von Android untereinander arbeiten und kommunizieren, wird auf die Arbeit von Cyl verwiesen [Cyl, 2019]. Damit unter anderem nicht jede App auf alles zugreifen kann, gibt es in den Methoden von den ManagerService-Klassen Methodenaufrufe, die den Zugriff kontrollieren. Dies liegt daran, dass Apps keinen direkten Zugriff auf die Hard-

und Softwarefunktionen von Android haben. Es wird, vereinfacht formuliert überprüft, ob eine App eine bestimmte Berechtigung besitzt und ob der Benutzer diese auch erlaubt hat. In der Abbildung 3.11 ist der *AlarmManagerService* mit der Methode *setTime* zu erkennen. In dieser wird mit der Methode *enforceCallingOrSelfPermission* überprüft, ob die Berechtigung für den aktuellen Prozess (App) vorhanden ist, wenn dies nicht der Fall ist, wird eine *SecurityException* durch den Methodenaufruf geworfen [Cyl, 2019].

```
1   ...
2   package com.android.server;
3   ...
4   class AlarmManagerService extends SystemService {
5       ...
6       private final IBinder mService = new IAlarmManager.Stub() {
7           ...
8           public boolean setTime(long millis){
9               getContext().enforceCallingOrSelfPermission(
10                  "android.permission.SET_TIME",
11                  "setTime");
12              return setTimeImpl(millis);
13          }
14          ...
15      }
16      ...
17  }
```

Abbildung 3.11: Ausschnitt von *AlarmManagerService.java*[Cyl, 2019, Seite 24]

Der Benutzer des Android-Slicers könnte dann *enforceCallingOrSelfPermission* als Seed-Statement angeben und kann das Ergebnis des Slicers dann analysieren. In Abbildung 3.12 ist anhand der Auswahl des Klassennamens *AlarmManagerService* zu erkennen, dass der Quellcode zusätzlich angezeigt wird.

Abbildung 3.12: Quellcodeansicht von *AlarmManagerService* (Quelle: Eigene Darstellung)

- Entry Methods
  Nach der Auswahl des Klassennamens steht dem Benutzer eine Liste von Methoden zur Auswahl, welche in den "Android Interface Definition Language" (AIDL)-Dateien hinterlegt sind. Durch den Android-Build-Prozess, also das Bauen der Android Ressourcen, entstehen die *.aidl-Dateien zu den jeweiligen Service-Klassen. Diese werden vom Slicer ausgelesen und zur Verfügung gestellt [Cyl, 2019].
  Als Beispiel könnte hier die *setTime*-Methode vom *AlarmManagerService* angeführt werden. Durch die Buttons *Add All* und *Clear* ist es möglich, entweder alle Methoden hinzuzufügen oder zu löschen. Zusätzlich wäre es auch möglich,die Methoden anzugeben, welche nicht aufgelistet werden.

- Seed Statements
  Unter den *Seed Statements* kann der Benutzer aus einer vorher festgelegten Liste eine Methode aussuchen. Diese Methoden sind aus der sicherheitsrelevanten Perspektive heraus interessant, wie zum Beispiel die *enforceCallingOrSelfPermission*-Methode [Cyl, 2019].
  Zusätzlich wäre es auch hier möglich, die Methoden anzugeben, welche nicht aufgelistet worden sind. Auch die Buttons *Add All* und *Clear* können dazu benutzt werden. Hier folgt eine kleine Vorschau hinsichtlich der Methoden, welche ausgesucht werden könnten. Alle aufgelisteten Methoden sind in der Abbildung 3.16 zu dem Setting-Tab zu sehen. Diese können dort unter den Namen *Seed_Statements* angepasst werden:

  * *checkCallingOrSelfPermission*
  * *checkCallingOrSelfUriPermission*
  * *checkPermission*
  * *enforceCallingOrSelfPermission*
  * *enforceCallingOrSelfUriPermission*
  * *enforcePermission*
  * *SecurityException*
  * ....

– Options
  Auf die einzelnen Optionen wird hier deshlab nicht eingegangen, da diese ausführlich im Kapitel 2.2 "Wala" erklärt worden sind. Die in der Abbildung 3.10 zu sehenden Einstellungen zu den Optionen sind die Standardwerte. Die Standardwerte können entweder in den *Slicer Options* oder in den *CFA Options* im Administration-Tab angepasst werden [Cyl, 2019].

In den Abbildung 3.13 und 3.14 ist der zweite Use Case abgebildet. Dieser Ansicht kann der Benutzer das Ergebnis oder den Zwischenstand des Slices entnehmen. Der einzige Unterschied zwischen dem Anzeigen des Ergebnis und dem Zwischenstand besteht darin, dass kein Sourcecode angezeigt wird. Der Benutzer sieht also immer, wenn er sich den Slice anzeigen lässt, die ausgewählten Optionen. Im Log wird der aktuelle des Prozesses aufgeführt. An diesem und am Running Symbol (Grün) kann erkannt werden, ob das Slicing noch läuft [Cyl, 2019].

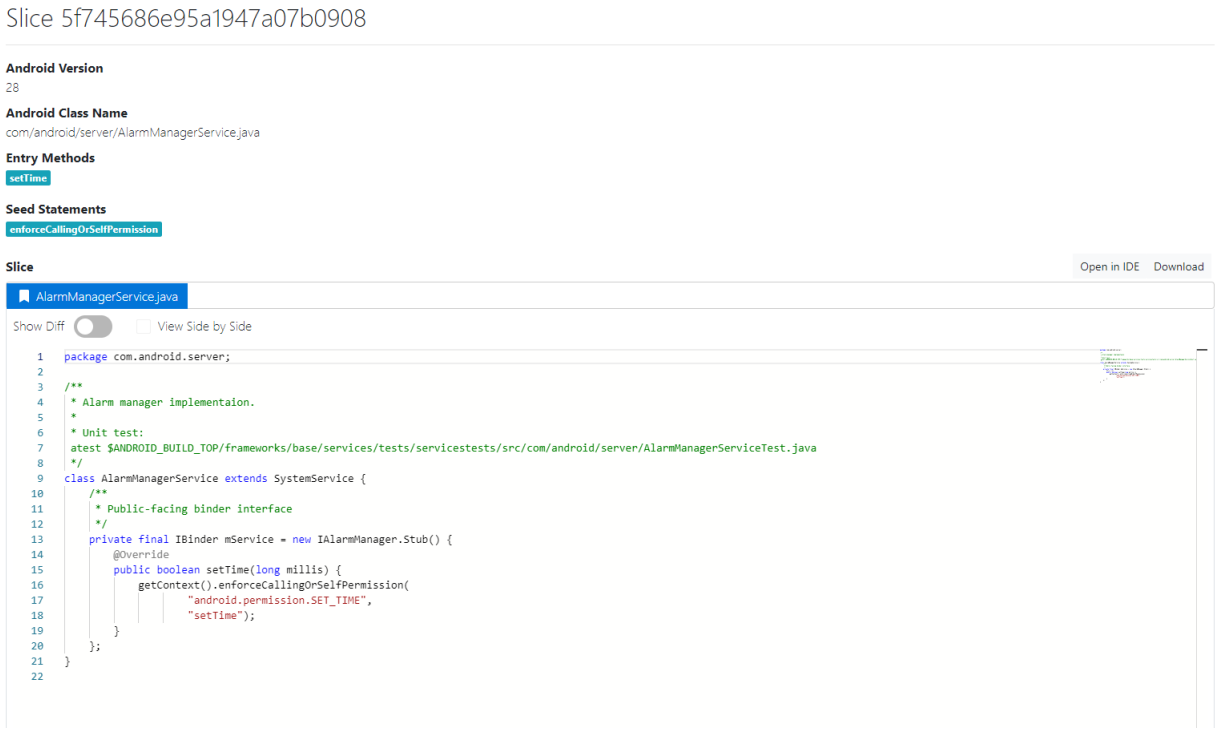


Abbildung 3.13: Ansicht eines fertigen Slices - Teil 1 (Quelle: Eigene Darstellung)

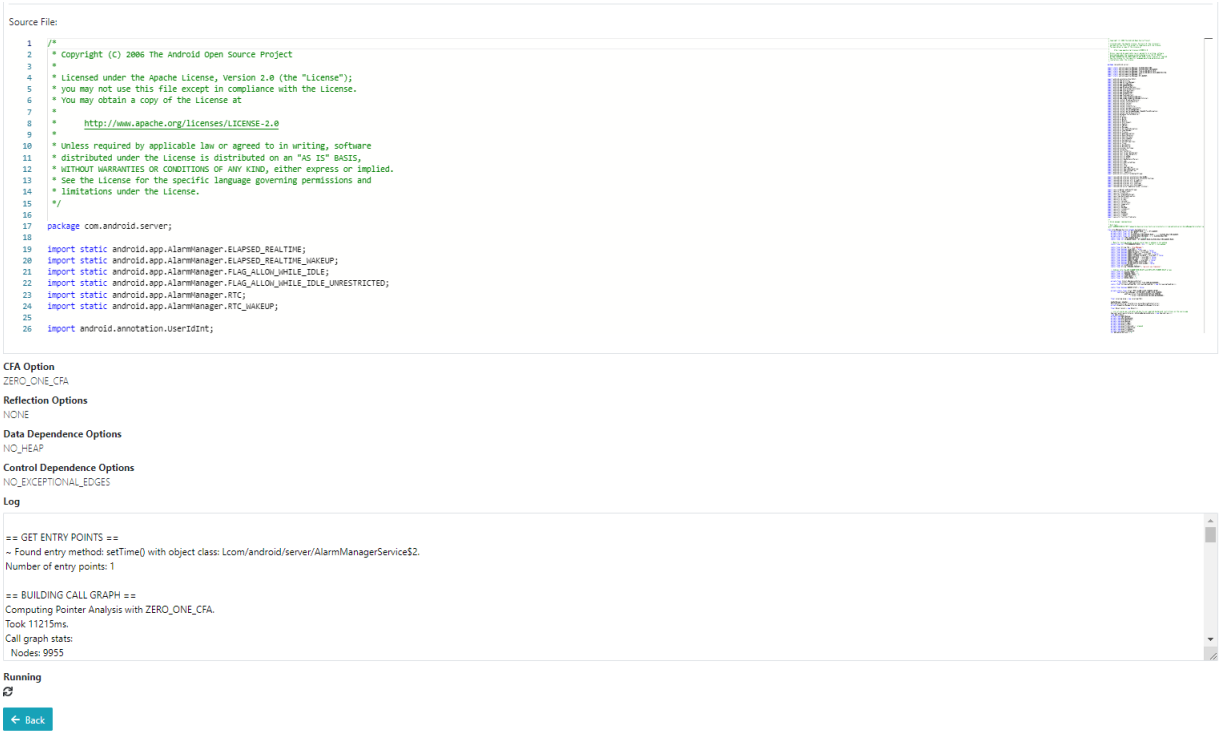


Abbildung 3.14: Ansicht eines fertigen Slices - Teil 2 (Quelle: Eigene Darstellung)

Über den *Download*-Knopf ist es dem Benutzer möglich, die erstellten Java-Dateien herunterzuladen und über den *"Open in IDE"*-Knopf können die Dateien in einer IDE geöffnet werden, solange dies in den Einstellungen eingestellt worden ist. Wenn der *Show-Diff*-Regler benutzt wird, werden das Slice-Ergebnis und der Quellcode gemeinsam angezeigt. Die roten Linien zeigen dabei die Zeilen, welche nicht im Slice-Ergebnis erhalten sind. Wird dann noch zusätzlich *View Side by Side* aktiviert, wird rechts neben dem zusammengefügten Quell- und Slicecode nur der Slicecode angezeigt. In Abbildung 3.15 sind die Codes mit den beiden Einstellungen zu sehen. Der angezeigte Slicecode enthält dieses Mal aber viele schraffierte graue Zeile in denen sonst nichts steht. Damit kann der Benutzer besser die Unterschiede zwischen Quellcode und Slicecode identifizieren.

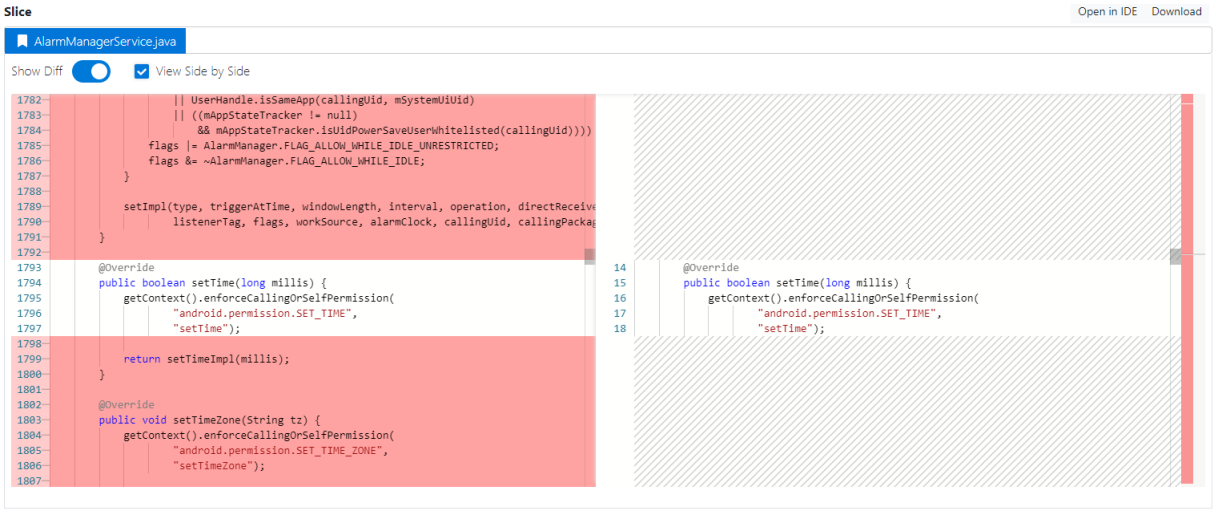


Abbildung 3.15: Ansicht eines fertigen Slices (Show-Diff und View Side by Side) (Quelle: Eigene Darstellung)

Im Settings-Tab, welches in der Abbildung 3.16 zu erkennen ist, können die bereits erwähnten Einstellungen zu *Android_System_Service_Regex* und *Seed_Statements* angepasst werden. An dieser folgt eine kurze Auflistung der wichtigsten, noch nicht benannten Einstellungen.

- Android_Source_Path
  Hier wird der Pfad zum Sourcecode von Android angegeben. Der Standardwert ist hier *android-resources*. In dem Ordner muss es Unterordner geben, die "android-xx"benannt sein müssen. xx ist der API-Level der Android Version.

- Android_Platform_Path
  Der Standardwert und das Prinzip des Unterordners sind die gleichen wie bei *Android_Source_Path*. Der einzige Unterschied liegt darin begründet, dass diese Einstellung der Pfad zur *android.jar* ist.

- Save_To_File
  Standardwert ist *true*. Wenn der Wert *true* ist, werden die Slices auch als Java-Dateien abgespeichert.

- Output_Directory
  Falls die Java-Dateien abgespeichert werden, wird hier der Pfad zum Abspeichern angegeben. Standardwert ist *output*.

- Exclusion_List
  An dieser Stelle können die Exclusions angepasst werden.

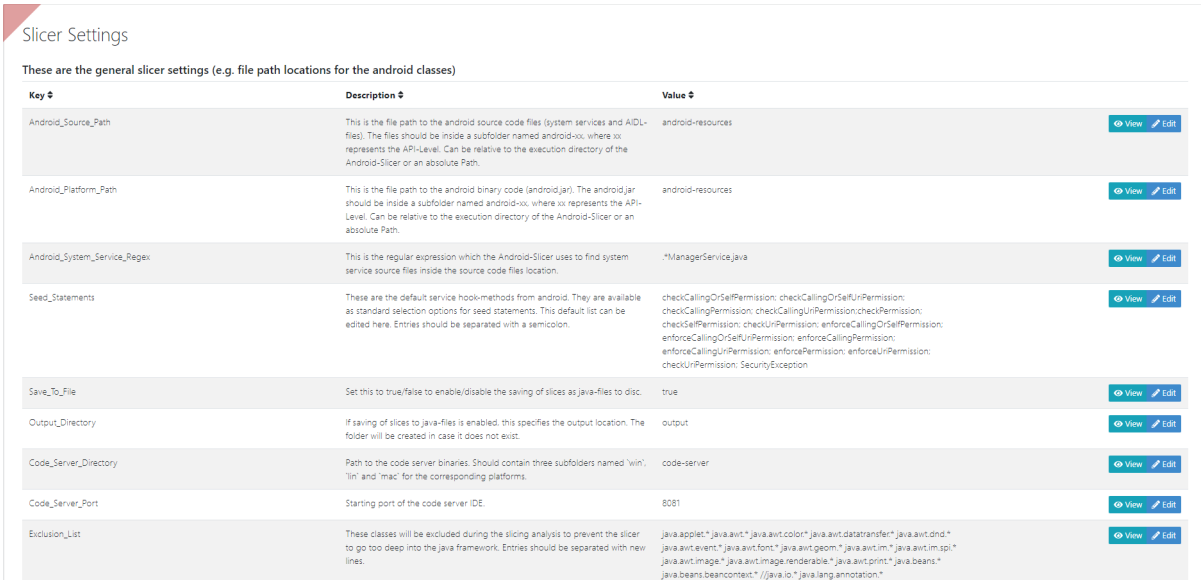


Abbildung 3.16: Settings-Tab Android-Slicer (Quelle: Eigene Darstellung)

Als Letztes gibt es noch den Administration-Tab. Die einzigen wichtigen Menü-Punkte sind hier die Slicer-Options und die CFA-Options. In den Slicer-Options können die Reflection-, Kontrollabhängigkeits- und Datenabhängigkeitsoptionen zu Kenntnis genommen werden. Es kann nur die Beschreibung angepasst werden. Zusätzlich können die einzelnen Optionen als Standard festgelegt werden, siehe Abbildung 3.17. Das Gleiche gilt auch für die CFA-Options, welche in der Abbildung 3.18 zu erkennen sind.

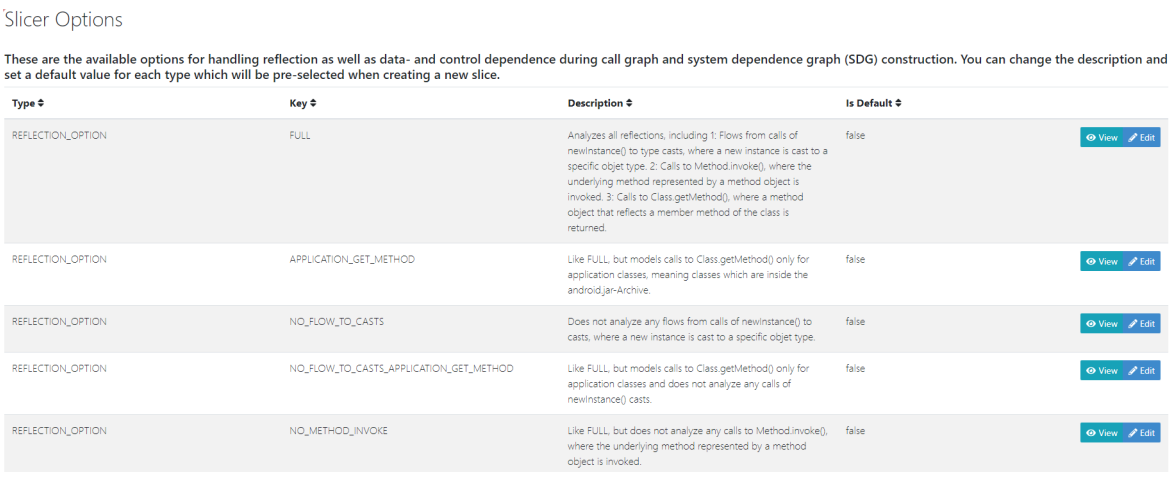


Abbildung 3.17: Slicer-Options Android-Slicer (Quelle: Eigene Darstellung)

Abbildung 3.18: CFA-Options Android-Slicer (Quelle: Eigene Darstellung)

Folgende Änderungen sind zu verzeichnen:

- Einstiegspunkte (Entrypoints)
  Anders als bei Java, bei der es nur die main-Methode als Einstiegspunkt in das System gibt, gibt es bei Android vier verschiedene Arten von Komponenten (Activity, Service, Broadcast-Reveicer und Content-Provider), welche als Einstiegspunkte fungieren. Eine main-Methode gibt es bei Android nicht [Cyl, 2019].
  Zu Erinnerung: Nguyen hat die Einstiegspunkte zum einen anhand der angegebenen *mainClass* gefunden. Diese gefundenen Methoden konnten sowohl *public* als auch *private* sein. Zum anderen wurden die main-Methoden verwendet. Der Programmablauf wurde in der Abbildung 3.2 konkretisiert [Nguyen, 2018].
  Im Gegensatz dazu werden bei Cyl nur diejenigen Methoden als Einstiegspunkte genommen, bei denen der Klassenname dem *Android Class Name* und die Methodennamen den *Entry Methods* entsprechen. Durch eine weitere Änderung ist es nun möglich, dass Methoden von inneren Klassen als Einstiegspunkte gefunden werden [Cyl, 2019].

- Mögliche Methoden der Seed Statements
  Bevor die Seed Statements gefunden werden können, werden die Methoden gesucht, welche die Seed Statements beinhalten könnten. Cyl hat Folgendes bei der Implementierung von Nguyen entdeckt:

  "Zur Identifizierung der Seed Statements werden zunächst die Call Graph-Knoten der Methoden benötigt, welche die entsprechenden Statements enthalten könnten. Der ursprüngliche Algorithmus von Nguyen hatte sich dabei ausschließlich auf das Finden von Entry Point-Knoten konzentriert. Während der ersten Funktionstests des Android-Slicers, [...] konnte allerdings festgestellt werden, dass sich Service

Hooks auch in privaten Methoden befinden können, d. h. nicht direkt in den Entry Points." (Zitat: [Cyl, 2019])

Folgendes Problem ist also aufgetreten:
Als Beispiel wird in der Methode *main* die Methode *print* aufrufen. Daher sind der *srcCaller* die *main*-Methode und *srcCallee* die *print*-Methode und somit auch das Seed-Statement. Der Algorithmus findet erfolgreich die *main*-Methode, ruft aber die *main*-Methode die *blubber*-Methode auf, welche erst dann die *print*-Methode aufruft, dann würde der Algorithmus keine Methode finden, weil *print* nicht in *main* enthalten ist.
Anders als Cyl habe ich durch das Testen der Implementierung festgestellt, dass dies weder an den Entrypoints noch an privaten Methoden liegt. Die Methode *findMethods* von Nguyen benutzt keinen dieser Parameter direkt. Zwar wird der Call Graph übergeben, welcher durch die Entrypoints generiert wird, aber durch die unterschiedlichen Algorithmen, welche in Abbildung 3.2 dargestellt worden sind, können auch private Methoden dann gefunden werden, wenn die richtigen Einstellungen vorgenommen werden. Das Problem liegt in dem übergebenen *srcCaller* begründet, welcher in meinen Beispiel die main-Methode war. Nguyen hat mit seiner Implementierung vorausgesetzt, dass sich das Seed-Statement *(srcCallee)* in der Methode *srcCaller* befindet.
Cyl hat diese Methode komplett hinzu der neuen Methode *findMethodNodes* überarbeitet. Auch hier werden der Call Graph und somit alle Methoden durchgegangen, nur muss der Klassenname der Methode mit dem *Android Class Name* übereinstimmen. Zusätzlich muss der Methodenname mit einen Namen aus der Liste der *Entry Methods* übereinstimmen. Außerdem wird die *findMethodNodes*-Methode für jede gefundene Methode nochmals aufgerufen. Übergeben wird statt der *Entry Methods* die Namen der Methoden, welche in der gefundenen Methode aufgerufen werden. Dieser rekursive Aufruf wiederholt sich so lange, bis alle Knoten der Klasse hinzugefügt worden sind. Somit konnte Cyl dieses Problem lösen [Cyl, 2019].

Folgende Optimierungen sind zu verzeichnen:

- Seed-Statements
  Zusätzlich zu dem Finden von *SSAAbstractInvokeInstruction* (dem Aufruf einer Methode) ist es möglich, dass *SSANewInstruction* (*new*-Anweisungen) als Seed-Statements gefunden werden. Somit ist es zum Beispiel möglich, throw-Anweisungen, in denen eine Exception mit dem new-Operator direkt instanziiert wird, gefunden wird [Cyl, 2019].
- JavaParser
  Folgende Optimierungen hat Cyl vorgenommen [Cyl, 2019].
  - Die Klassen *MethodVisitor* und *Parser*, welche für das Rekonstruieren des Quellcodes zuständig sind, wurden im großen Stiel refaktoriert.

- Redundanter Code wurde in neue Methoden ausgelagert. So sind die Methoden *isLineInNode* und *addAllLinesFromBeginToEnd* in der Klasse *MethodVisitor* entstanden. Die beiden Methoden sind sehr wichtig für das Rekonstruieren des Quellcodes. Die *isLineInNode*-Methode überprüft, ob eine Zeile in einem Knoten liegt. Auf dieser Weise wird sichergestellt, dass der Knoten nicht unnötig durchsucht wird. Die *addAllLinesFromBeginToEnd*-Methode fügt, wie der Name bereits verrät, von einer Zeilennummer bis einschließlich einer anderen Zeilennummer alle Zeilen zum Ergebnis hinzu. Damit ist es leichter, umgebrochene Zeilen zu rekonstruieren.
- Kommentare aus dem Quellcode werden übernommen, welche an den gefundenen Zeilen liegen. Dies schließt JavaDoc mit ein. Die dafür entsprechenden Methoden liegen in der *SliceMapper*-Klasse.
- Es werden alle Zeilen hinzugefügt, welche mit der Zeichenkette "package" beginnen.
- Das Finden von Zeilenumbrüchnen wurde verbessert. Dafür wurden die Anweisungen beziehungsweise die Statements *ForEachStmt*, *SynchronizedStmt*, *Expression-Stmt*, *ThrowStmt*, *SwitchStmt*, *SwitchEntry*, *ReturnStmt*, *ContinueStmt*, *BreakStmt*, *Catch-* und *Finally-Blöcke* erweitert oder hinzugefügt.
- Die Klassendeklaration und damit auch die Kommentare werden hinzugefügt.
- *return-*, *continue-* und *break-*Anweisungen werden unabhängig vom Wala-Ergebnis übernommen, damit der Code verständlicher wird.
- Der JavaParser wurde auf die neueste Version 3.14 aktualisiert. Genutzt wurde vorher die Version 1.0, welche maximal die Java-Version 7 unterstützte.

Folgende aufgetretene Problematiken sind zu verzeichnen:

- Android Versionen
  Das Bereitstellen der Android-Ressourcen war für Cyl komplizierter und umständlicher als angenommen. Bis Mitte des Jahres 2018 war es noch möglich, auf der Hosting-Plattform *Grepcode* auf die vorkompilierten Android-Ressource zugreifen zu können. Diese Möglichkeit nutzte Gulmann in seiner Arbeit [Gulmann, 2014].
  Aber die Plattform stellte den Betrieb ein. Cyl versuchte daraufhin, mithilfe unterschiedlichen Methoden selbst die Android-Ressourcen zu erstellen und schaffte dies auch. Auf diesen Punkt wird aber nicht weiter eingegangen. Es wird auf die Arbeit von Cyl verwiesen, in der er sein Vorgehen sehr gut dokumentiert hat [Cyl, 2019].

Ausblick:

- Usability
  Damit der Android-Slicer im Hinblick auf die Usability verbessert werden kann,

müsste eine größere Studie durchgeführt werden, welche mit standardisierten Tests arbeitet [Cyl, 2019].

- Ergebnis der Evaluation
  Durch die Evaluation konnte nachgewiesen werden, dass das Tool den Benutzer bei der Arbeit in der Software Security gut unterstützen kann. Folgende Punkte, welche noch nicht umgesetzt wurden sind, wurden als wünschenswert geschildert.

  – *import*-Anweisungen der Klassen sollten für das Verständnis von Variablen mit aufgenommen werden [Cyl, 2019].

  – Übersetzung des Tools in mehrere Sprachen. Dies soll laut Cyl mit der Internationalisierungsfunktion des JHipster-Frameworks, welches für die Server-Client-Architektur genommen wurde, möglich sein [Cyl, 2019].

  – Ein Slice auf Basis eines vorhandenen Slices mit veränderten Einstellungen sollte wiederholt werden können [Cyl, 2019].

3.5 Jan Möhlmann

Möhlmann erweiterte den Android-Slicer dahin gehend, dass dieser auch Java-Anwendungen slicen kann. Die Evaluation der Erweiterung und der hinzugefügten Funktionalitäten ist wegen der herrschenden Covid-19-Pandemie knapp ausgefallen [Möhlmann, 2020].

Folgende neue Features beinhaltet diese Unterarbeit:

- Oberfläche
  In der Abbildung 3.19 sind zwei neue Buttons links neben den "Create a new Slice" Button zu erkennen. Durch einen Klick auf einen der beiden Buttons wechselt der Benutzer zwischen der Liste für die Slices von Android oder von Java hin und her.

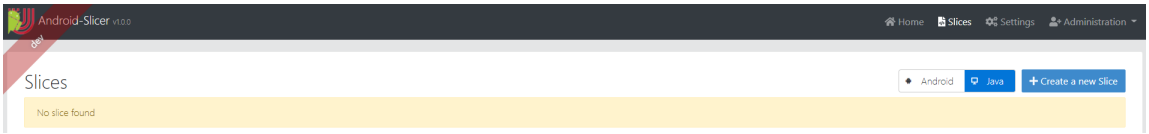


Abbildung 3.19: NEW Slices-Tab Android-Slicer (Quelle: Eigene Darstellung)

Eine größere Veränderung hat die Ansicht zur Erstellung eines neuen Slices vollzogen. Diese ist in den Abbildungen 3.20 und 3.21 zu sehen.

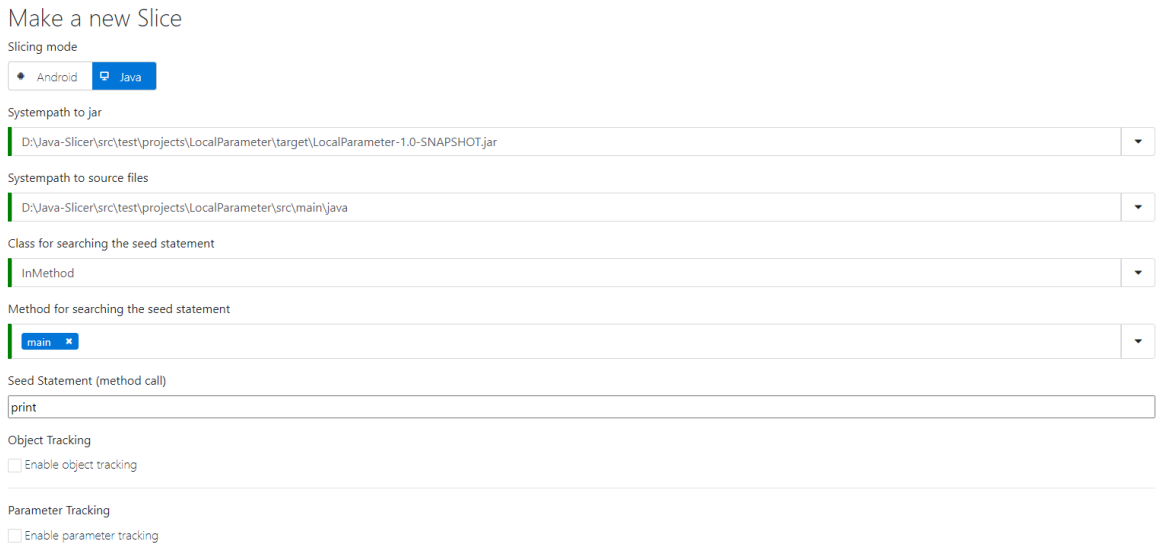


Abbildung 3.20: NEW Ansicht zur Erstellung eines neuen Slices (Quelle: Eigene Darstellung)

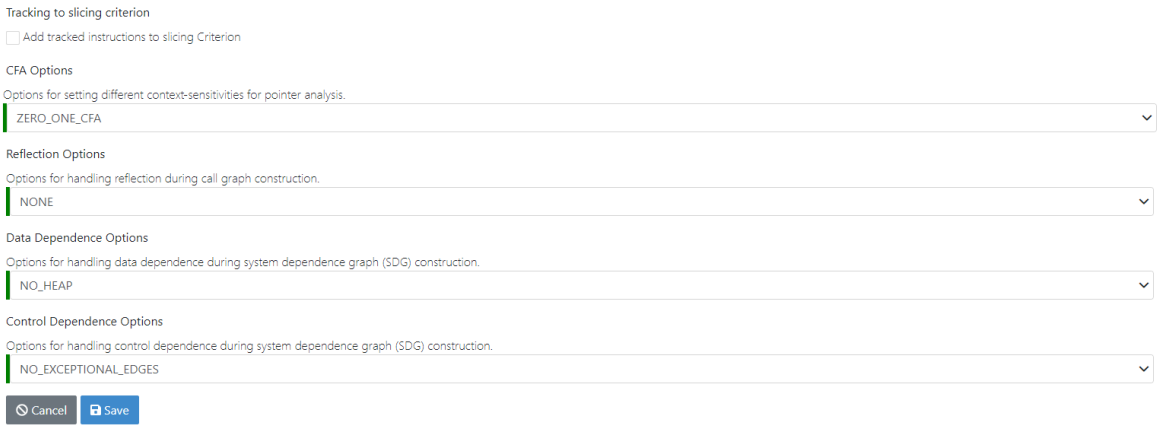


Abbildung 3.21: NEW Ansicht zur Erstellung eines neuen Slices (Quelle: Eigene Darstellung)

An dieser Stelle wird kurz auf die veränderten Einstellungsmöglichkeiten eingegangen.

- Oben links kann der Benutzer zwischen den *Slicing mode* Java und Android wechseln. Dieses Vorgehen ist das gleiche wie bei der Slices-Ansicht.

- Systempath to...
  Die Android-spezifische Option *Android Version* ist weggefallen. Stattdessen gibt es die beiden neuen Optionen *Systempath to jar* und *Systempath to source files*. Wie die Namen der Optionen bereits aussagen, geben diese den Pfad zur Jar-Datei und zu den Source-Dateien an.

  - Class for searching the seed statement
    Im Slicing-Mode Android heißt diese Option *Android Class Name*. Es werden alle Klassennamen angezeigt, welche WALA durch das Erzeugen der ClassHierarchy findet. Bei der Auswahl einer Klasse wird kein Quellcode angezeigt.
  - Method for searching the seed statement
    Im Slicing-Mode Android heißt die Option *Entry Methods*. Nach der Auswahl des Klassennamens stand dem Benutzer eine Liste von Methoden zur Auswahl, welche WALA durch die Methode *getAllMethods* auf der Klasse findet. Die Buttons *Add All* und *Clear* wurden entfernt.
  - Seed Statement (method call)
    Es existiert keine vorher festgelegte Liste. Daher kann der Benutzer keine Methode aussuchen, sondern muss genau wissen, welches Seed-Statement er benutzen möchte. An dieser Stelle kann auch nur noch ein einziges Seed-Statement angegeben werden.

    Die folgenden drei Optionen sind neu:
  - Object Tracking
    Ist die Option *Object Tracking* aktiviert, so wird das Objekt, auf dem das Seed-Statement aufgerufen wird, verfolgt. Daher wird die Objektverfolgung eingeschaltet. Alle Verwendungen des Objekts sollten im besten Fall im Slice enthalten sein.
  - Parameter Tracking
    Ist die Option *Parameter Tracking* aktiviert, werden die Parameter, welche dem Seed-Statement übergeben werden, verfolgt. Daher wird die Parameterverfolgung eingeschaltet. Alle Verwendungen der Parameter sollten im besten Fall im Slice enthalten sein.
  - Tracking to slicing criterion
    Ist diese Option und zusätzlich *Object Tracking* oder *Parameter Tracking* aktiviert, werden die gefundenem Verwendungen des Objekts oder der Parameter als Seed-Statements benutzt.
  - Options
    Alle anderen vorhandenen Optionen haben sich nicht verändert.

- Jar-Dateien und Java-Quellcode
  Durch die Angabe der Dateipfade ist es nur noch möglich, dass der Server auf dem PC des Clients läuft. Möhlmann will damit verhindern, dass zu große Jar- und Quellcode-Dateien auf den Server hochgeladen werden. Ein potenzieller Angreifer könnte aber durch das Ausführen des Programms die Ordnerstruktur des PC ermitteln [Möhlmann, 2020].

- Einstiegspunkte (Entrypoints)
  Möhlmann benutzt für das Finden der Einstiegspunkte für die Java-Anwendungen nur die *Util.makeMainEntrypoints* Methode, welche WALA mit sich bringt.

- Mögliche Methoden der Seed Statements
  Anders als Cyl benutzt Möhlmann den Ansatz von Nguyen für das Java-Slicing. Somit setzt Möhlmann wieder voraus, dass sich der Seed-Statement in der angegebenen Methode *"Method for searching the seed statement"* befindet.

- Tracking
  Für das Tracking des Objekts und/oder der Parameter stehen die zwei neue Methoden (*addParameterTracking* & *addObjectTracking*) zur Verfügung und für das Hinzufügen zum Ergebnis des Slices bzw. das Hinzufügen als Seed-Statement ist die Methode *addAliasesToSlice* zuständig. Diese drei Methoden werden etwas genauer in dem Kapitel 6 "Implementierung" ausgeführt. Für eine detaillierte Beschreibung der Implementierung wird auf die Arbeit von Möhlmann verwiesen [Möhlmann, 2020].
  Möhlmann hat das Tracking deshalb komplett neu implementiert, weil die Implementierung von Nguyen drei Schwachstellen aufwies:
  - Die Objektverfolgung ist nur intraprozedural zuverlässig.
  - Es gib dann False-Positives, wenn mehrere Objekte die gleiche Klassen haben. Es werden daher alle Objekte der Klasse gefunden.
  - Die Instanzierungsmethoden müssen bekannt sein und werden teils nicht in den Slice integriert.

Folgende Optimierungen sind zu verzeichnen:

- Exklusionsdatei
  Im Settings-Tab, welcher in Abbildung 3.16 gezeigt worden ist, kann die Exklusiondatei angepasst werden. Das Problem, welches Möhlmann entdeckte, besteht darin, dass immer nur der erste Eintrag in der Liste verwendet worden ist, wenn die Liste verändert wurde. Als Lösung erkennt nun das Programm Leerzeichen als Trenner der Einträge an.

- JHipster
  JHipster wurde von der Version 6.2.0 auf die 6.8.0 aktualisiert.

Folgende aufgetretene Problematiken sind zu verzeichnen:

- Tracking
  Es werden auch Methodenaufrufe vom Callee in das Slice-Ergebnis aufgenommen, welche erst nach dem Seed-Statement aufgerufen werden würden. Die gefundenen Methodenaufrufe sind False-Positiv. Beim Tracking wird nicht zwischen Backward- und Forwardslicing differenziert. In einem Test war die Zeile *consoleInput = new BufferedReader(new InputStreamReader(System.in));* enthalten, obwohl diese nichts

mit dem Objekt *cipher* vom Seed-Statement *doFinal* zu tun hatte. Dieser False-Positiv konnte sich nicht erklärt werden [Möhlmann, 2020].

Ausblick:
Folgende mögliche Ansätze sah Möhlmann für die nächsten Arbeiten an dem Slicer: [Möhlmann, 2020]

- Durch WALA können nur Java-Programme bis zur Version 8 analysiert werden. Eine Erweiterung mit einer zweiten Analysebibliothek könnte das Problem lösen.
- Statt des zurzeit benutzten Backward-Slicen könnte auch das Fordward-Slicen verwendet werden.
- Die Analyse von Java SE Anwendungen ist durch seine Arbeit ermöglicht worden, ob Java EE Anwendungen analysiert werden können, muss evaluiert werden.
- Wie Nguyen hat Möhlmann vorgeschlagen, Mockups sollten die richtigen Klassen in den Bibliotheken ersetzen, um die Performance der Analyse zu verbessern.
- Die Quelltextrekonstruktion erzeugt immer noch keine syntaktisch korrekten Java-Dateien.
- Die Jar- und Quellcode-Dateien müssen zurzeit deshalb auf dem PC des Benutzers liegen, weil dort der Server sein muss. Das Problem könnte mit einem Buildserver behoben werden.
- Über einen "Klonen"-Button sollte ein Slice auf Basis eines vorhandenen Slices mit veränderten Einstellungen erzeugt werden.
- Unterschiedliche Slices sollten miteinander verglichen werden.
- Slices sollten in verschiedene Ordner einsortiert werden können.
- Das Festlegen des Seed-Statements ist nicht immer eindeutig. Das Problem könnte gelöst werden, indem das Seed-Statement direkt im Quellcode ausgesucht wird.

3.6 Aufgetretene Problematiken und Ausblicke der Vorarbeiterin

An dieser Stelle werden die Punkte der Vorarbeiten zu deren Ausblicken und aufgetretenen Problematiken kurz zusammengefasst. Im Kapitel 5 "Anforderungen" werden unter anderen durch diese Punkte, die verschwundenen Funktionalitäten und die ersetzten Funktionalitäten die Anforderungen für den Slicer erstellt.

- JavaParser
  - Die Umbrüche in den Zeilen, in denen z.B. eine Anweisung ausgeführt oder eine Methode aufgerufen wird, sind fehlerhaft [Nguyen, 2018].

  - Innere- sowie anonyme-Java-Klassen werden ignoriert [Kerimov, 2019].
  - Am Ende von Catch-Blöcken fehlten teilweise geschweifte Klammern [Kerimov, 2019].
  - Den Quellcode durch einen Decompiler aus dem Bytecode rekonstruieren lassen. Dadurch könnte insbesondere das Problem gelöst werden, wenn kein Quellcode vorhanden ist [Kerimov, 2019].
  - import-Anweisungen der Klassen sollten mit aufgenommen werden [Cyl, 2019].
  - Die rekonstruierten Java-Dateien sind noch nicht syntaktisch korrekt [Möhlmann, 2020].

- Objektverfolgung
  - Der Slice enthielt bestimmte Zeilen, selbst wenn der *advanced_mode* ausgeschaltet war, und andersherum fehlte z.B. der Parameter *value*, obwohl *advanced_mode* eingeschaltet war [Nguyen, 2018].
  - Methodenaufrufe vom Callee werden in das Slice-Ergebnis aufgenommen, welche erst nach dem Seed-Statement aufgerufen werden würden. [Möhlmann, 2020].
  - Es war mindestens eine Zeile im Slice-Ergebnis enthalten, welche nichts mit dem Objekt zu tun hatte [Möhlmann, 2020].

- Einstiegspunkte (Entrypoints)
  - Das Finden der Einstiegspunkte muss verbessert und evaluiert werden. Je größer ein Programm ist, desto sinnvoller ist es, dass so wenige Einstiegspunkte wie möglich gefunden werden [Nguyen, 2018].

- Seed-Statement
  - Das Festlegen des Seed-Statements ist nicht eindeutig. Dieser könnte direkt im Quellcode ausgesucht werden [Möhlmann, 2020].

- Attribute
  - Die Klassenattribute werden generell nicht betrachtet [Kerimov, 2019].

- Dokumentation
  - Die Fehlerbehandlung und Dokumentation müssen verbessert werden ([Nguyen, 2018], [Kerimov, 2019]).

- Testen
  - Evaluation des Slicers an großen Java-Programmen mit über 200.000 LoC [Nguyen, 2018].
  - Ob Java EE Anwendungen analysiert werden können, müsste evaluiert werden [Möhlmann, 2020].

- Datenflüsse

  - Gibt es eine Optimierung durch das Ersetzen der Klassen und Methoden in Bibliotheken durch Stubs und verändert sich das Ergebnis dadurch ([Nguyen, 2018], [Möhlmann, 2020])?

- Forward-Slicen
  - Statt nur Backward-Slicen könnte auch Fordward-Slicen genutzt werden. Ob dies nützlich wäre, müsste dann evaluiert werden ([Kerimov, 2019], [Möhlmann, 2020]).

- Android Versionen
  - Das Generieren der unterschiedlichen Android Versionen ist sehr zeitaufwendig und rechenintensiv. Ein einfacher Download ist nicht möglich [Cyl, 2019].

- Usability
  - Eine große Studie müsste durchgeführt werden, um den Slicer im Hinblick auf Usability zu verbessern [Cyl, 2019].
  - Übersetzung des Tools in mehrere Sprachen [Cyl, 2019].
  - Einen Slice, auf Basis eines vorhandenen Slices, mit veränderten Einstellungen wiederholen ([Cyl, 2019], [Möhlmann, 2020]).
  - Unterschiedliche Slices miteinander vergleichen [Möhlmann, 2020].
  - Slices in verschiedene Ordner einsortieren [Möhlmann, 2020].

- Bibliotheken
  - Eine weitere Analysebibliothek hinzufügen, um Java-Programme über Version 8 analysieren zu können [Möhlmann, 2020].

- Server
  - Durch einen Buildserver müssten die Jar- und Quellcode-Dateien nicht mehr auf dem PC des Benutzers liegen [Möhlmann, 2020].

3.7 Verschwundene Funktionalitäten

In diesem Unterkapitel werden die verschwundenen Funktionalitäten aufgelistet, welche nicht durch andere Vorarbeiten wieder ersetzt worden sind. Sowohl in diesem als auch im nächsten Unterkapitel werden keine Funktionalitäten aufgelistet, welche mit Android direkt zu tun haben. Dies hat den Grund, dass keine Funktionalität, welche Cyl erarbeitet hat, verschwunden oder ersetzt worden ist.

- Die Option *src_calle_class* aus der Arbeit von Nguyen.
  Über diese Option in der Konfigurationsdatei war es möglich, das Seed-Statement auf eine bestimmte Klasse zu beschränken. Wenn z.B. das Seed-Statement *toString*

und die Klasse *Auto* sind, dann würde der Slicer auch nur die Methoden *toString* von der Klasse *Auto* nehmen und sonst von keiner weiteren Klasse.

- Die Option *only_public_entry* aus der Arbeit von Nguyen.
  Damit wurde entweder der Entrypoint Algorithmus von Gulmann oder von Detmers genutzt.
- Das Erstellen von unterschiedlichen PDFs und die dazu gehörigen Optionen aus der Arbeit von Nguyen.
- Beim Erstellen eines neues Slices sind folgende Funktionalitäten nicht mehr vorhanden:
  - Es wird kein Quellcode mehr angezeigt, wenn die Java-Klasse ausgesucht wird.
  - Der Benutzer hat nicht mehr die Möglichkeit, alle Callees hinzuzufügen oder zu löschen.
  - Es kann nur noch ein Seed-Statement gewählt werden.

3.8 Ersetzte Funktionalitäten

In diesem Unterkapitel werden alle Funktionalitäten aufgelistet, welche durch die Vorarbeiten ersetzt oder immer wieder erweitert worden sind.

- Der Slicer
  Der ursprüngliche Java-Slicer von Nguyen wurde durch Cyl als Android-Slicer neu implementiert. Dieser wurde dann von Möhlmann um Java erweitert. Somit kann der Slicer Android- und Java-Anwendungen slicen.
- GUI
  Aus einem Slicer, welcher durch ein Kommandozeilenprogramm ausgeführt werden musste, wurde eine Client-Server-Webanwendung.
- Konfigurationsdatei / Output-Dateien
  Alle Einstellungsmöglichkeiten und Ergebnisse werden in einer Datenbank abgespeichert. Vorher wurde dies durch Extradateien geregelt.
- Objektracking
  Die erste Version stammt von Nguyen und wurde später von Möhlmann neu implementiert. Die Optionen *advancded_mode* und *object_inst_methods* wurden durch die Optionen *Object Tracking*, *Parameter Tracking* und *Tracking to slicing criterion* ersetzt.
- Einstiegspunkte (Entrypoints)
  Der Algorithmus zum Finden der Einstiegspunkte wurde sehr oft "ersetzt" beziehungsweise zusammengesetzt. In Abbildung 3.2 ist der Algorithmus zu sehen, welchen Ngyuen in seiner Arbeit verwendet hat. Dieser beinhaltete die Algorithmen von

Gulmann, Detmers und von WALA. Im Android-Slicer fiel dieser deshalb komplett weg, weil Android andere Einsteigpunkte besitzt. Beim aktuellen Stand des Slicers wird nur die Methode *Util.makeMainEntrypoints* im *Slicing-Mode* Java von WALA verwendet.

- Mögliche Methoden der Seed Satetemtens
  Im Slicer von Nguyen und im aktuellen Slicer (*Slicing-Mode: Java*) musste sich das Seed-Statement im Callee befinden. Im *Slicing-Mode* Android muss der Seed-Statement nur in der Klasse sein, welche beim *Android Class Name* angegeben worden ist.

- JavaParser
  Das Rekonstruieren der Java-Dateien wurden stetig verbessert und die dazugehörigen Klassen wurde durch Cyl im großen Stil refaktoriert. Trotzdem scheint dies noch fehlerhaft zu sein.

3.9 Zusammenfassung des Kapitels

In diesem Kapitel wurde die stetige Weiterentwicklung des vorhandenen Java/Android-Slicer veranschaulicht. Angefangen von den prototypischen Machbarkeitsbeweisen von Gulmann und Gerken, über ein Kommandozeilenprogramm von Nguyen, bis hin zum aktuellen Slicer von Möhlmann. Jeder von ihnen legte andere Schwerpunkte und lieferte damit neue offene Problematiken sowie Ausblicke. Durch genau deren Zusammenfassung sowie durch die verschwundenen und ersetzten Funktionalitäten wurde die Grundlage für die nächsten Schritte dieser Arbeit geschaffen.

Kapitel 4

Methodik

In diesen Kapitel werden die methodischen Ansätze dieser Arbeit konkretisiert. Als Erstes wird die allgemeine Vorgehensweise bei der Implementierung und der Evaluation erklärt. Im zweiten Teil wird auf die Herangehensweise bei den funktionalen und nicht-funktionalen Anforderungen eingegangen.

4.1 Vorgehensweise

Für diese Arbeit wird das inkrementelle iterative Verfahren von Basili und Turner aus dem Jahre 1975 minimal abgeändert und benutzt [Basil and Turner, 1975]. Das Verfahren besteht grundlegend aus vier Phasen:

- Initiale Implementierung eines skelettartiges Subproblems des Projekts
- Gestaltung der Implementierung (Design Phase)
- Coding und Debugging (Implementation Phase)
- Durchführung einer Analyse der bestehenden Teilumsetzung (Analyse-Phase).

Zusätzlich wird eine Liste geführt, in welcher die durchzuführenden Aufgaben aufgelistet sind. Nachdem die initiale Implementierung abgeschlossen worden ist, werden eine Aufgabe aus der Liste gewählt und die Phasen 2-4 ausgeführt. Zwischen und nach den Phasen können weitere Aufgaben zur Liste hinzukommen. Dieses Vorgehen wird solange praktiziert, bis alle Aufgaben abgearbeitet sind und keine weiteren mehr hinzugekommen sind [Basil and Turner, 1975].
Für diese Arbeit wird deshalb es keine initiale Implementierung geben, weil auf den bestehenden Systemen (Slicer), welche die Vorarbeiter implementiert haben, aufgebaut wird. Diese wurden bereits ausführlich im Kapitel 3 "Verwandte Arbeiten" beschrieben. Die genannten "Aufgaben" werden in diesem Fall die Anforderungen sein, die an den Slicer gestellt werden.
Am Anfang werden nur wenige neue Anforderungen bekannt sein. Diese werden aus dem

gefundenen Problematiken, den Ausblicken der Vorarbeiter sowie den zwischenzeitlich verschwundenen und ersetzten Funktionalitäten abgeleitet.
In der Analyse-Phase wird die Evaluation durchgeführt. Diese wird zum einen durch automatische Unit-, Integrations- und UI-Tests und zum anderen durch manuelle Tests vorgenommen. Für die automatischen Tests werden unter anderem Benchmarks für Cipher und Signature angelegt und beim manuellen Testen wird der Slicer anhand einer Open-Source Bibliothek evaluiert. Es können nur einzelne Aspekte bzw. Fälle des Slicers evaluiert werden, denn den ganzen Slicer mit allen möglichen Einstellungen zu testen, ist unmöglich. Der Fokus der Evaluation liegt deswegen auf den neuen Anforderungen, welche im Kapitel 5 mit aufgelistet werden. Daher fällt die Evaluation unter die empirische Methodik der Fallstudie. Dieses beschrieben Tichy und Padberg folgendermaßen:

„Eine Fallstudie liefert eine eingehende Analyse und Beschreibung eines einzelnen oder einiger weniger Individuen, Organisationen oder Ereignisse. [...] Als Quellen einer Fallstudie werden Dokumente, Testergebnisse, [...] ausgewertet. [...] In der Softwareforschung werden Fallstudien oft dazu benutzt, die prinzipielle Funktionsfähigkeit eines Werkzeugs oder einer Methodik zu zeigen.[...] “ (Zitat: [Tichy and Padberg, 2007, S.215-216])

Die einzelnen Testfälle und die Evaluation werden im Kapitel 7 präzisiert.
Aufgrund der Tatsache, dass am Anfang noch nicht alle Anforderungen zu erkennen sind, bietet dieses iterative Verfahren den Vorteil, dass zu jedem Zeitpunkt neue Anforderungen hinzugefügt werden können. Beim bekannten Wasserfallmodell, welches sehr strikt ist, ist dies nicht möglich. Denn beim Wasserfallmodell von [Royce, 1970] müssen alle Anforderungen am Anfang bekannt sein und jede Phase aus diesem Modell wird nur einmal ausgeführt. Zum Schluss muss das fertige System stehen. Es gibt unter anderem auch andere iterative Verfahren, wie zum Beispiel das Spiralmodell von Boehm [Boehm, 1988]. Dieses ist aber zu detailreich und komplex. In jeder Iteration müsste unter anderem eine Risikoanalyse durchgeführt werden. Diese Analyse wird aber für so ein kleines Projekt nicht benötigt.

4.2 Anforderungen

Wie im Abschnitt 1.2 "Ziele der Arbeit" beschrieben, sind die Ziele dieser Arbeit darauf ausgerichtet, die verschwundenen oder ersetzten Funktionalitäten der Vorarbeiten zu identifizieren, gegebenenfalls zu implementieren und zu testen. Identifiziert wurden diese in den Abschnitten 3.7 und 3.8. Noch nicht umgesetzte Funktionalitäten (Ausblick) und aufgetretene Problematiken der Vorarbeiten sollen durch mögliche Ansätze gelöst werden. Identifiziert wurden diese im Abschnitt 3.6. Diese Erweiterungen und Optimierungen des

Slicers werden mithilfe von Anforderungen aufgeschrieben, implementiert und evaluiert. Die Anforderungen werden grob durch zwei unterschiedliche Ansätze generiert:

- Aufarbeitung der Vorarbeiten
- Das manuelle und automatische Testen des Slicers (Evaluation).

Damit das Ziel, die Arbeiten der Vorgängern zusammenführen, erreicht wird, müssen auch die Anforderungen von den Vorarbeiten zusammengeführt werden. Eine vollständige Auflistung aller neuen und alten Anforderungen von Nguyen, Cyl und Möhlmann ist im Kapitel 5 zu finden. Dazu wurden die alten entweder direkt übernommen oder leicht angepasst.

Diese funktionale Anforderungen werden an die *Product functions* der "Anforderungsspezifikation" des Standards *ISO/IEC/IEEE 29148:2018* angelehnt [iee, 2018].
Es wird daher nicht die ganze Anforderungsspezifikation für dieses Projekt umgesetzt. Dieses Projekt besteht nur aus einer Person, daher würde ein so umfangreiches Dokument nur unnötig viel Zeit kosten. Es gibt außerdem Punkte in dem Dokument, die für dieses Projekt unwichtig sind, die daher für diese und für spätere Arbeiten keinen Mehrwert generieren, wie z.B. *Stakeholders*. Zu dem Zeitpunkt dieser Arbeit und bei den Vorarbeiten gab es keinen bekannten *Stakeholder*.

Die Produktfunktionen geben eine Zusammenfassung der wichtigsten Funktionen des Systems wieder. Es gibt aber keine klare Definition, wie die Produktfunktionen auszusehen haben:

> *„Die Produktfunktionen sollten so organisiert sein, dass die Liste der Funktionen für den Erwerber oder jeden anderen, der das Dokument zum ersten Mal liest, verständlich ist. [...] Zur Darstellung der verschiedenen Funktionen und ihrer Beziehungen können textliche oder grafische Methoden verwendet werden.[...]“ (Übersetztes Zitat: [iee, 2018] S.78)*

Die funktionalen Anforderungen werden in dieser Arbeit textlich dargestellt und werden an die Strukturen von Nguyen [Nguyen, 2018] und Cyl [Cyl, 2019] angelehnt. Jede Anforderung wird mit einer eindeutigen ID mit der Kennung "AF", einer fortlaufender Zahl und einem kurzen, präzisen Namen versehen. "AF" steht für Anforderung. Darauf folgt die genaue Beschreibung der Anforderung. Als zweiter Punkt wird der Ursprung der Anforderung angegeben. Entweder wird an dieser Stelle *Initial* oder *Evaluation* benutzt. Alle initialen Anforderungen wurden direkt am Anfang der Arbeit aufgeschrieben. Alle anderen Anforderungen resultieren aus der iterativen Evaluation.
Der Punkt "Priorität" beschreibt die Wichtigkeit der Anforderung. Dabei wird zwischen *Muss*, *Soll* und *Wunsch* unterschieden. Der vierte Punkt betrifft die "Rolle". Die Anforderung kann entweder zum *System* und/oder den *Benutzer* zugeordnet werden. Die meisten Anforderungen werden beide Rollen haben, weil sowohl auf der Benutzer- und als auf der

Systemseite die Anforderungen umgesetzt werden müssen.
Die alten Anforderungen wurden aufgrund der eben genannten Punkte angepasst, daher haben diese unter anderem eine neue ID erhalten. So wurde der Ursprung z.B. von allen fertigen Umsetzungen der Anforderungen auf *Initial* gesetzt. Textuelle Veränderungen werden durch kursive und durchgestrichene Wörter markiert. Dabei wurden durchgestrichene Wörter verworfen und kursive Wörter kamen neu hinzu.

Zusätzlich werden noch die nicht-funktionalen Anforderungen und die Einschränkungen aufgelistet. Auch diese sind an den Standard *ISO/IEC/IEEE 29148:2018* angelehnt. Die nicht-funktionalen Anforderungen definieren die Rahmenbedingungen und Qualitäten und überschneiden sich mit den funktionalen Anforderungen. So betrifft z.B. die nicht-funktionale Anforderung *Korrektheit* alle Anforderungen, die mit dem Slice-Ergebnis zu tun haben.
An dieser Stelle würde es keinen Sinn machen, die nicht-funktionalen Anforderungen und die Einschränkungen neu zu definieren. Denn das Ziel dieser Arbeit ist es, die Arbeiten der Vorgänger zusammenzuführen, zu erweitern und zu evaluieren. Deswegen werden die nicht-funktionalen Anforderungen und die Einschränkungen von Nguyen und Cyl übernommen, welche im Kapitel 5 mit aufgelistet werden.

Kapitel 5

Anforderungen

In diesen Kapitel werden zuerst alle alten und neuen funktionalen Anforderungen aufgelistet. Neben den Anforderungen ist vermerkt, von welchem Vorarbeiter diese übernommen worden sind. Steht ein *NEW* neben einer Anforderung, ist diese neu, aber meistens wurde der Text aus einer alten Anforderungen übernommen und entsprechend angepasst. Im zweiten Teil werden die nicht-funktionalen Anforderungen von Cyl und Nguyen konkretisiert. Zum Schluss werden die Einschränkungen, welche für den Slicer gemacht werden müssen, definiert.

5.1 Funktionale Anforderungen

AF01: Ansicht zum Slice erstellen [Cyl, 2019]

Beschreibung: Das Tool muss dem Benutzer eine Eingabemöglichkeit für die Erstellung eines neuen Slices bereitstellen. Die Eingabe soll dabei über eine grafische Benutzeroberfläche (im Englischen „Graphical User Interface“, kurz GUI) erfolgen.
Ursprung: Initial
Priorität: Muss
Rolle: Benutzer/System

AF02: Auswahl Android-Version [Cyl, 2019]

Beschreibung: Das Tool muss die zur Verfügung stehenden Android Versionen anzeigen. Weiterhin muss der Benutzer die zu analysierende Version auswählen können. *Nur im Slicemode Android möglich.*
Ursprung: Initial
Priorität: Muss
Rolle: Benutzer/System

AF03: Vorauswahl der Android-Version [Cyl, 2019]

Beschreibung: Sofern lediglich eine Android-Version zur Verfügung steht, soll diese vom Tool für den Benutzer vorausgewählt werden.
Ursprung: Initial
Priorität: Kann
Rolle: Benutzer/System

AF04: Auswahl Java-Jar-Name *NEU*

Beschreibung: Das Tool muss die zur Verfügung stehenden Java-Jar-Names anzeigen. Weiterhin muss der Benutzer die zu analysierende Jar-Datei auswählen können. Nur im Slicemode Java möglich.
Ursprung: Evaluation
Priorität: Muss
Rolle: Benutzer/System

AF05: Auswahl der Einstiegspunkt-Klasse [Nguyen, 2018]

Beschreibung: Der Benutzer soll vor der Analyse *über* verschiedene Möglichkeiten ~~haben~~ *verfügen*, mit Angabe einer *Klasse* ~~Hauptklasse~~ die Ermittlung der Programm-Einstiegspunkte zu beeinflussen. Das System soll abhängig davon, ob der Benutzer *im Slicemode Java* eine explizite *Klasse* ~~Hauptklasse~~ angibt, verschiedene Methoden durchprobieren, um Einstiegspunkte für die Analyse zu finden, ansonsten diese Angabe ignorieren und global alle Einstiegspunkte ermitteln. *Im Slicemode "Android" muss diese Klasse angegeben werden. Das System nimmt dann entweder alle Methoden oder nur spezifische als Einstiegspunkt.*
Ursprung: Initial
Priorität: Muss
Rolle: Benutzer/System

AF06: Filterung der Einstiegspunkt-Klasse [Cyl, 2019]

Beschreibung: Das Eingabefeld für Anforderung *AF05* ~~A4~~ sollte dem Benutzer eine Liste mit *den Klassennamen (Java)* und den Systemdiensten *(Android)*, welche in der zuvor ausgewählten Framework-Version (vgl. Anforderung *AF02* ~~A2~~) *bzw. Java-Jar-Name (vgl. Anforderung AF04)* vorhanden sind, zur einfachen Auswahl bereitstellen. Zudem sollte die Benutzereingabe im Rahmen einer Autovervollständigung sinnvoll ergänzt werden. Die Eingabe beliebiger Klassennamen, welche ggf. nicht in der Liste vorhanden sind, muss weiterhin möglich sein.
Ursprung: Initial
Priorität: Muss
Rolle: Benutzer/System

AF07: Auswahl der Einstiegspunkt-Methoden *NEU*

Beschreibung: Der Benutzer soll vor der Analyse verschiedene Möglichkeiten haben, durch die Angabe mehrerer Methodennamen die Ermittlung der Programm-Einstiegspunkte zu beeinflussen. Das System soll abhängig davon, nur Einstiegspunkte für die Analyse finden, welche mit den angegebenen Methodennamen übereinstimmen. Wurden keine Methodennamen angegeben, werden alle Methoden der gefundenen Klasse aus der Anforderung AF04 als Einstiegspunkt entnommen.
Ursprung: Evaluation
Priorität: Muss
Rolle: Benutzer/System

AF08: Filterung der Einstiegspunkt-Methoden *NEU*

Beschreibung: Das Eingabefeld für Anforderung AF07 sollte dem Benutzer eine Liste mit Methodennamen, welche in der zuvor ausgewählten Klasse (vgl. Anforderung AF06) vorhanden sind, zur einfachen Auswahl bereitstellen. Zudem sollte die Benutzereingabe im Rahmen einer Autovervollständigung sinnvoll ergänzt werden. Die Methodennamen sollten dabei für Android aus der AIDL-Datei des Systemdienstes ausgelesen werden. Für Java sollen die Methodennamen anhand der Klassenhierarchie ermittelt werden, welche durch WALA erzeugt werden. Die Eingabe beliebiger Methodennamen, welche ggf. nicht in der Liste vorhanden sind, muss weiterhin möglich sein.
Ursprung: Evaluation
Priorität: Muss
Rolle: Benutzer/System

AF09: Anzeige des Quellcodes der Einstiegspunkt-Klasse [Cyl, 2019]

Beschreibung: Der Quellcode für den in *der* Anforderung *AF05* ~~A4~~ ausgewählte *Klasse oder* Systemdienst sollte dem Benutzer angezeigt werden, sodass dieser in die weitere Konfiguration der Slicing-Parameter einbezogen werden kann.
Ursprung: Initial
Priorität: Muss
Rolle: System

AF10: Auswahl der Methoden-Klasse *NEU*

Beschreibung: Der Benutzer soll vor der Analyse die Möglichkeit haben, mit Angabe einer Klasse die Ermittlung der "Methoden für das Seed-Statement" zu beeinflussen. Das System soll abhängig davon, ob der Benutzer eine explizite Klasse angibt oder nicht, entweder die "Methoden für das Seed-Statement" nur in der Klasse oder aber global ermitteln.
Ursprung: Evaluation
Priorität: Muss
Rolle: Benutzer/System

AF11: Filterung der Methoden-Klasse *NEU*

Beschreibung: Das Eingabefeld für die Anforderung AF10 sollte dem Benutzer eine Liste mit den Klassennamen (Java) und den Systemdiensten (Android), welche in der zuvor ausgewählten Framework-Version (vgl. Anforderung AF02) bzw. Java-Jar-Name (vgl. Anforderung AF04) vorhanden sind, zur einfachen Auswahl bereitstellen. Zudem sollte die Benutzereingabe im Rahmen einer Autovervollständigung sinnvoll ergänzt werden. Die Eingabe beliebiger Klassennamen, welche ggf. nicht in der Liste vorhanden sind, muss weiterhin möglich sein.
Ursprung: Evaluation
Priorität: Muss
Rolle: Benutzer/System

AF12: Auswahl der Methoden-Namen *NEU*

Beschreibung: Der Benutzer soll vor der Analyse verschiedene Möglichkeiten besitzen, mit der Angabe mehrerer Methodennamen die Ermittlung der "Methoden für das Seed-Statement" zu beeinflussen. Das System soll abhängig davon nur "Methoden für das Seed-Statement" für die Analyse finden, welche mit den angegebenen Methodennamen übereinstimmen. Wurden keine Methodennamen angegeben, werden alle Methoden der gefundenen Klasse aus der Anforderung AF10 als "Methoden für das Seed-Statement" verwendet.
Ursprung: Evaluation
Priorität: Muss
Rolle: Benutzer/System

AF13: Filterung der Methoden-Namen *NEU*

Beschreibung: Das Eingabefeld für Anforderung AF12 sollte dem Benutzer eine Liste mit Methodennamen, welche in der zuvor ausgewählten Klasse (vgl. Anforderung AF10) vorhanden sind, zur einfachen Auswahl bereitstellen. Zudem sollte die Benutzereingabe im Rahmen einer Autovervollständigung sinnvoll ergänzt werden. Die Methodennamen sollten dabei für Android aus der AIDL-Datei des Systemdienstes ausgelesen werden. Für Java sollen die Methodennamen anhand der Klassenhierarchie ermittelt werden, welche durch WALA erzeugt wird. Die Eingabe beliebiger Methodennamen, welche ggf. nicht in der Liste befinden, muss weiterhin möglich sein.
Ursprung: Evaluation
Priorität: Muss
Rolle: Benutzer/System

AF14: Anzeige des Quellcodes der Methoden-Klasse *NEU*

Beschreibung: Der Quellcode für die in Anforderung AF10 ausgewählte Klasse oder Systemdienst sollte dem Benutzer angezeigt werden, sodass dieser in die weitere Konfiguration der Slicing-Parameter einbezogen werden kann. Der Quellcode wird aber nur dann angezeigt, wenn die ausgewählte Klasse unterschiedlich zur ausgewählten Klasse aus der Anforderung AF05 ist.
Ursprung: Evaluation
Priorität: Kann
Rolle: System

AF15: Konfiguration des Seed-Statements [Cyl, 2019]

Beschreibung: Das Tool muss dem Benutzer eine Eingabemöglichkeit zur Definition des *Seed-Statements* ~~Slicing-Kriteriums~~ bereitstellen ~~(siehe Abschnitt 2.1)~~. ~~Das Slicing-Kriterium wird durch die WALA-Entwickler und daher auch in der GUI des Android-Slicers als "Seed-Statement" bezeichnet.~~ Es sollte die Möglichkeit *existieren* ~~(~~bestehen), mehrere *Seed-Statements* ~~Slicing-Kriteriums~~ in das Eingabefeld ~~für Anforderung A10~~ einzugeben. *Zusätzlich sollen die Seed-Statements auch in Form eines regulären Ausdrucks konfigurierbar sein.*
Ursprung: Initial
Priorität: Muss
Rolle: Benutzer/System

AF16: Filterung des Seed-Statements [Cyl, 2019]

Beschreibung: Das Eingabefeld für Anforderung *AF15* ~~A7~~ sollte dem Benutzer *für Android* eine Liste mit den ~~in Abschnitt 3.2.3 beschriebenen~~ Service Hooks*, wie z.B. "checkPermission" und "enforcePermission",* als sinnvolle Vorauswahl *zur Verfügung stellen* ~~bereitstellen~~*. Für Java wird eine Liste mit den Methodennamen der ausgewählten Klasse aus der Anforderung AF10 bereitgestellt, falls in dem Eingabefeld eine Klasse ausgewählt worden ist, falls nicht, werden die Methodennamen von der Klasse aus der Anforderung AF05 ausgewählt.* Die Benutzereingabe sollte dabei im Rahmen einer Autovervollständigung unter ~~dem~~ Bezug auf die Liste vereinfacht werden. Die Eingabe beliebiger Seed-Statements, welche nicht in der Liste *vorzufinden* ~~vorhanden~~ sind, muss trotzdem weiterhin möglich sein.
Ursprung: Initial
Priorität: Muss
Rolle: Benutzer/System

AF17: Auswahl der Seed-Statement-Klassen *NEU*

Beschreibung: Der Benutzer soll vor der Analyse über die Möglichkeit verfügen, mit Angabe von keiner, einer oder mehreren Klassen die Ermittlung der Seed-Statements zu beeinflussen. Das System soll abhängig davon, ob der Benutzer eine oder mehrere Klassen angegeben hat, nur diejenigen Seed-Statements ermitteln, bei denen die Klasse mit den angegebenen Klassen übereinstimmt. Wenn keine Klasse angegeben worden ist, werden die gefundenen Seed-Statements nicht weiter eingeschränkt. Die Klassennamen können entweder als reine Namen oder zusammen mit dem Package angegeben werden.
Ursprung: Evaluation
Priorität: Muss
Rolle: Benutzer/System

AF18: Objekt-Tracking *NEU*

Beschreibung: Es kann vorkommen, dass im Slice-Ergebnis nicht alle objektabhängigen Anweisungen der Seed-Statements enthalten sind, obwohl die Datenabhängigkeits- und Kontrollabhängigkeitsoptionen mit unterschiedlichen Einstellungen benutzt werden. Das System soll dann intraprozedural sowie interprozedural alle Objektabhängigkeiten ermitteln und diese dem Slice-Ergebnis hinzufügen. Auf der Benutzeroberfläche bietet das Tool die Möglichkeit für den Benutzer, diese Option zu aktivieren.
Ursprung: Initial
Priorität: Muss
Rolle: Benutzer/System

AF19: Parameter-Tracking *NEU*

Beschreibung: Analog zu *AF18* gibt es das gleiche Problem zu den parameterabhängigen Anweisungen des Seed-Statements. Das System soll dann auch intraprozedural sowie interprozedural alle Parameterabhängigkeiten ermitteln und diese dem Slice-Ergebnis hinzufügen. Auf der Benutzeroberfläche bietet das Tool die Möglichkeit für den Benutzer, diese Option zu aktivieren.
Ursprung: Initial
Priorität: Muss
Rolle: Benutzer/System

AF20: Tracking to slicing criterion *NEU*

Beschreibung: Das System nimmt die gefundenen Anweisungen aus den Anforderungen AF18 und AF19 als Seed-Statements an, dabei wird Anforderung AF17 nur für diese neuen Seed-Statements ignoriert. Es werden nur Anweisungen als Seed-Statements hinzugefügt, welche in den gleichen Methoden, wie die Seed-Statements aus Anforderung AF15, liegen. Auf der Benutzeroberfläche bietet das Tool die Möglichkeit für den Benutzer, diese Option zu aktivieren.
Ursprung: Initial
Priorität: Kann
Rolle: Benutzer/System

AF21: Lokale Variablen & Attribute Tracking *NEU*

Beschreibung: Beim Slicing der Java-Anwendungen oder der Android-Ressourcen werden die lokalen Variablen der Methoden und Attribute häufiger "ignoriert" und sind dann nicht im Slice-Ergebnis vorhanden. Daher soll der Slicer das Slicing durch WALA unterstützen und die verwendeten lokalen Variablen und Attribute selbst identifizieren. Auf der Benutzeroberfläche bietet das Tool die Möglichkeit für den Benutzer, diese Option zu aktivieren.
Ursprung: Evaluation
Priorität: Muss
Rolle: Benutzer/System

AF22: Hochladen & Löschen von Java- und Jar-Dateien *NEU*

Beschreibung: Der Benutzer kann über ein Pop-up Fenster, welches sich über einen Button öffnen lässt, entweder Zip-Dateien hochladen oder bereits hochgeladene wieder löschen. Die Zip-Datei enthält dabei die zu analysierende Jar-Datei sowie die Java-Dateien, welche zum Rekonstruieren benutzt werden. Die Zip-Datei und die Jar-Datei müssen für das Hochladen den gleichen Namen tragen. Das System muss darauf achten, dass die Namen gleich sind und dass bei Problemen beim Hochladen oder Löschen, diese auch angezeigt werden.
Ursprung: Evaluation
Priorität: Muss
Rolle: Benutzer/System

AF23: Auswahl des k-l-Levels [Cyl, 2019]

Beschreibung: Das Tool muss dem Benutzer eine Eingabemöglichkeit zur Auswahl des Levels der Points-To-Analysen ~~(siehe Abschnitt 2.3.2)~~ bieten.
Ursprung: Initial
Priorität: Muss
Rolle: Benutzer/System

AF24: Konfiguration des Umgangs mit Reflections [Cyl, 2019]

Beschreibung: Im Rahmen der Konstruktion des Call Graphen ~~(siehe Abschnitt 2.3.1)~~ implementiert das WALA-Framework verschiedene Herangehensweisen bei der Analyse von Java-Reflections, d. h. von Aufrufen von Klassen und Methoden über String-Parameter, wie z. B. über Method.invoke. Das Tool muss dem Benutzer eine Konfigurationsmöglichkeit der unterschiedlichen Optionen bieten.
Ursprung: Initial
Priorität: Muss
Rolle: Benutzer/System

AF25: Konfiguration der Datenabhängigkeitsoptionen [Cyl, 2019]

Beschreibung: Das WALA-Framework bietet zur Anpassung der Genauigkeit und *zur Reduzierung* des Ressourcenaufwands der Datenabhängigkeitsanalysen ~~(siehe Abschnitt 2.2.3)~~ verschiedene Möglichkeiten zum Ausschluss von bestimmten Datenabhängigkeitskanten. Das Tool muss dem Benutzer eine Konfigurationsmöglichkeit der verschiedenen Optionen *eröffnen* ~~bieten~~.
Ursprung: Initial
Priorität: Muss
Rolle: Benutzer/System

AF26: Konfiguration der Kontrollabhängigkeitsoptionen [Cyl, 2019]

Beschreibung: Das WALA-Framework bietet zur Anpassung der Genauigkeit und *zur Reduzierung* des Ressourcenaufwands der Kontrollabhängigkeitsanalysen ~~(siehe Abschnitt 2.2.2)~~ verschiedene Möglichkeiten zum Ausschluss von bestimmten Kontrollabhängigkeitskanten. Das Tool muss dem Benutzer eine Konfigurationsmöglichkeit der verschiedenen Optionen *eröffnen* ~~bieten~~.
Ursprung: Initial
Priorität: Muss
Rolle: Benutzer/System

AF27: Erklärung und Vorauswahl der Slicing-Optionen [Cyl, 2019]

Beschreibung: Die Eingabefelder für die Anforderungen ~~A14, A15, A16 und A17~~ AF23, AF24, AF25 und AF26 sollten dem Benutzer einerseits eine Erklärung der jeweiligen Parameter anzeigen. Andererseits sollten sinnvolle Parameter im Hinblick auf das Slicing von Android- Systemdiensten vorausgewählt und dementsprechend markiert sein.
Ursprung: Initial
Priorität: Muss
Rolle: Benutzer/System

AF28: Einschränkung des Analyseumfangs [Cyl, 2019]

Beschreibung: Das Tool muss die Möglichkeit *eröffnen* ~~bieten~~, den Umfang der Slicing-Analysen entsprechend zu begrenzen, um die Analyse von grundlegenden Java-Bibliotheken (z. B. java.math.*) und damit einen zu großen Analyseaufwand zu verhindern. Dem WALA-Framework kann zu diesem Zweck eine Liste mit Ausnahmen (im Englischen "Exclusions") übergeben werden. Der Android-Slicer kann dafür die von Detmers entworfene Exclusion-Datei nutzen (vgl. [Detmers, 2016]).
Ursprung: Initial
Priorität: Muss
Rolle: Benutzer/System

AF29: Durchführung der Analysen [Cyl, 2019]

Beschreibung: Das Tool muss *dazu* in der Lage sein, anhand der ausgewählten Optionen und Slicing- Parameter einen Backward-Slice ~~des entsprechenden Android-Systemdienstes~~ zu berechnen.
Ursprung: Initial
Priorität: Muss
Rolle: System

AF30: Threading der Slices [Cyl, 2019]

Beschreibung: Da das Slicing eine längere Zeitspanne in Anspruch nehmen kann, müssen die Analysen durch das WALA-Framework in einen separaten Thread ausgelagert werden. Somit ist die weitere Nutzung des Android-Slicers auch bei laufenden Analysen möglich.
Ursprung: Initial
Priorität: Muss
Rolle: System

AF31: Anzeigen des Fortschritts [Cyl, 2019]

Beschreibung: Die Zwischenergebnisse des WALA-Frameworks, wie z. B. die erfolgreiche Konstruktion des Call Graphen, sollten während des Slicings kontinuierlich aktualisiert und dem Benutzer angezeigt werden.
Ursprung: Initial
Priorität: Muss
Rolle: System

AF32: Rekonstruktion des Quellcodes [Cyl, 2019]

Beschreibung: Nach dem Abschluss des Backward-Slicings stellt das WALA-Framework eine Sammlung der ermittelten Anweisungsknoten im SDG ~~(siehe Abschnitt 2.3.3)~~ zur weiteren Auswertung *zur Verfügung* ~~bereit~~. Über den Bytecode-Index der entsprechenden Anweisungen können zudem deren Zeilennummern im originalen Quellcode festgestellt werden (vgl. [WALA, 2021a]). Das bedeutet allerdings, dass die konkreten Zeilennummern mehrzeiliger Anweisungen sowie der Klassen-, Methoden- und Kontrollanweisungsblöcke, welche die identifizierten Anweisungen umgeben, zusätzlich ermittelt werden müssen, um einen sinnvollen Programmabschnitt zu erhalten. Die Umsetzung dieser Anforderung sollte auf der Implementierung von Nguyen basieren, da diese bereits Optimierungen gegenüber den Algorithmen von Gulmann und Detmers enthält. Dazu zählt u. a. die korrekte Rekonstruktion *von* ~~bei~~ bestimmten syntaktischen Besonderheiten, wie z. B. if-Anweisungen ohne else- Zweige, sowie von mehrfach verschachtelten Anweisungsblöcken (vgl. [Nguyen, 2018]).
Ursprung: Initial
Priorität: Muss
Rolle: System

AF33: Ausgabe von Kommentaren [Cyl, 2019]

Beschreibung: Für die in Anforderung ~~A23~~ AF32 ermittelten Anweisungen, Methoden und Klassen sollten die Kommentare aus dem originalen Quellcode in den Slice übernommen werden, um die Verständlichkeit des Codes zu verbessern.
Ursprung: Initial
Priorität: Muss
Rolle: System

AF34: Anzeige des Slices [Cyl, 2019]

Beschreibung: Das Tool muss dem Benutzer eine GUI-Oberfläche zur Anzeige der Slicing-Ergebnisse bereitstellen. Zudem müssen dabei die ausgewählten Optionen und Parameter *zu erkennen* ~~ersichtlich~~ sein.
Ursprung: Initial
Priorität: Muss
Rolle: Benutzer/System

AF35: Anzeige des originalen Quellcodes [Cyl, 2019]

Beschreibung: Um die Ergebnisse der Analyse, d. h. den entstandenen Slice-Code, effizienter analysieren zu können, sollte der originale Quellcode *der unterschiedlichen Klassen* ~~des untersuchten Systemdienstes~~ zur leichteren Referenzierung ebenfalls in dem GUI-Element der Anforderung *AF34* ~~A25~~ eingeblendet werden.
Ursprung: Initial
Priorität: Muss
Rolle: System

AF36: Speicherung als Java-Dateien [Cyl, 2019]

Beschreibung: Die Slices sollten zusätzlich als Java-Dateien auf der Festplatte gespeichert werden, um diese ggf. für weitere Analysen in einer vollwertigen integrierten Entwicklungsumgebung (im Englischen „Integrated Development Environment“, kurz IDE) zu öffnen.
Ursprung: Initial
Priorität: Muss
Rolle: System

AF37: Internationalisierung [Cyl, 2019]

Beschreibung: Die Hauptsprache des ~~Android-~~Slicers soll *deshalb* Englisch sein, da die Endanwender zu großen Teilen mit der englischen Sprache arbeiten werden. Dies betrifft sowohl *die* Anleitungen bzgl. des WALA-Frameworks als auch den Quellcode der *Klassen* ~~Android-Systemdienste~~ und der entsprechenden Slices. Eine Internationalisierung, d. h. Übersetzung des Tools in andere Sprachen, sollte allerdings ohne großen Aufwand möglich sein.
Ursprung: Initial
Priorität: Muss
Rolle: System

AF38: Slice mit gleichen Einstellungen starten *NEW*

Beschreibung: Das Tool muss die Möglichkeit eröffnen, einen (un)fertigen Slice zu wiederholen. Daher soll der Benutzer zur "Ansicht zum Slice erstellen" weitergeleitet werden, welche in Anforderung AF01 beschrieben worden ist. Die Ansicht wird dazu mit den Einstellungen des (un)fertigen Slice befüllt, welche durch den Benutzer verändert werden kann.
Ursprung: Evaluation
Priorität: Kann
Rolle: Benutzer/System

5.2 Nicht-funktionale Anforderungen

Die nicht-funktionalen Anforderungen wurden von Cyl und Nguyen zum großen Teil eins zu eins übernommen. Dabei wurden die Texte vereinzelt angepasst. Die kursive Wörter sind neu und die durchgestrichenen Wörter werden verworfen.

Wartbarkeit (N1)
„Im Rahmen dieser Arbeit soll ein später öffentlich verfügbares sowie quelloffenes Analysetool entstehen, welches Sicherheitsexperten weltweit bei Code-Audits unterstützen kann. Dies benötigt zukünftig ggf. die Aktualisierung bestimmter Komponenten oder das Hinzufügen neuer Funktionen. Deshalb sollte der Quellcode eine gute Wartbarkeit und Erweiterbarkeit aufweisen, was einerseits einen modularen Aufbau und andererseits eine gute Dokumentation des Programms voraussetzt.“ (Zitat: [Cyl, 2019])
Außerdem soll bei der Implementierung auf eine englische Internationalisierung geachtet werden. [Nguyen, 2018]

Benutzbarkeit (N2)
„Eines der Hauptziele des ~~Andoroid-~~Slicers stellt die Vereinfachung der *Analysen*

~~Systemdienstanalysen~~ dar, weshalb das Tool eine intuitive und leicht zu verstehende grafische Konfigurationsoberfläche aufweisen soll. Um den Benutzern einen einfachen Einstieg zu bieten, soll zudem möglichst viel Expertise, basierend auf der Grundlagenforschung dieser Arbeit, im Slicer gekapselt werden.“ (Zitat: [Cyl, 2019])

Korrektheit (N3)
Die Analyseergebnisse sollten sowohl sinngemäß als auch syntaktisch möglichst korrekt sein [Cyl, 2019] [Nguyen, 2018].

Portierbarkeit (N4)
„Um das Tool einem großen Anwenderkreis zur Verfügung stellen zu können, wäre eine plattformunabhängige Implementierung bzw. eine Lauffähigkeit für verschiedene Betriebssysteme wünschenswert.“ (Zitat: [Cyl, 2019])

Skalierbarkeit (N5)
„Das Anbieten der verschiedenen Konfigurationsmöglichkeiten soll in einem gewissen Maße auch die Skalierbarkeit der Anwendung gewährleisten. Im besten Fall kann bei großen komplexen Programmen die Genauigkeit des Slices mit richtiger Parametrisierung dahingehend reduziert werden, dass sich noch nutzbare Ergebnisse ergeben. Dadurch ist der Trade-off zwischen Genauigkeit und Skalierbarkeit für den Anwender selbständig justierbar.“ (Zitat: [Nguyen, 2018])
„Das Tool sollte in der Lage sein, auch für genauere, d. h. aufwändigere Slicing- Analysen noch Ergebnisse zu liefern. Dafür bietet sich u. a. eine Client-Server- Struktur an, welche die Anzeige auf dem Endgerät des Benutzers und die Berechnungen auf einem leistungsstarken Server-System durchführt. Die Aufteilung des Tools in Client- und Server-Komponenten würde dabei zudem *die bestehenden und neuen Anforderungen* ~~die Anforderung N1~~ unterstützen.“ (Zitat: [Cyl, 2019])

Leistung / Performance (N6)
„Bezüglich der Leistung des *Slicers* ~~Auditors~~, soll dieser nach Möglichkeit ressourcensparend und effektiv implementiert sein. Um das Analyseverhalten nach Komplexität der zu analysierenden Anwendung den verfügbaren Ressourcen anzupassen, wird eine große Auswahl an Konfigurationsmöglichkeiten angeboten.“ (Zitat: [Nguyen, 2018])

5.3 Einschränkungen

Wie bei allen Vorarbeiten muss auch an dieser Stelle auf die Einschränkungen verwiesen und aufgelistet werden.
Der wichtigste Punkt betrifft die *Korrektheit* des Slice-Ergebnisses. Im Laufe der letzten Jahren wurde der Slicer stetig verbessert und auch an dem WALA-Framework wurde selbst immer weitergearbeitet. Aber wie Gulmann, Nguyen und Cyl in ihren Arbeiten betont haben, liegt auch genau dort der Schwachpunkt der Korrektheit. Weil auf dem

WALA-Framwork aufgebaut wird und diesem nur Parameter übergeben werden für das Slicen, besitzt das Slicing-Tool, keine Möglichkeiten, in das Slicing von WALA direkt einzugreifen. Dazu kommt, dass WALA sowohl bekannte als auch mögliche unbekannte Fehler beinhaltet. Somit ist der Slicer von den Ergebnissen von WALA abhängig und noch viel wichtiger ist, dass diese Ergebnisse weder validiert noch korrigiert werden können. Zum Beispiel zeigt sich dies in dem Fall, dass sehr einfache Zuweisungen weg optimiert werden [WALA, 2021b]. Dies kann aber leider nicht geändert werden.

An dieser Stelle muss auch das Halteproblem genannt werden, welches dann zu verzeichnen ist, wenn grundsätzlich nicht entschieden werden kann, ob ein Algorithmus terminiert oder nicht. Durch dieses Problem kann das Slicing in diesem Fall nie zu einem perfekten Ergebnis gelangen und es wird insbesondere *false positives* sowie *false negatives* geben.
Im Kontext des Slicing bedeutet ein *false positives*, dass Anweisungen, also Zeilen, im Slice-Ergebnis enthalten sind, welche nichts mit dem Seed-Statement zu tun haben. Ein *false negative* beschreibt genau das Gegenteil. Daher handelt es sich um eine Anweisung (Zeilen), welche nicht im Slice-Ergebnis enthalten ist, aber zum Seed-Statement gehört. Daher sollten *false negatives* am besten nicht vorkommen. *False positives* hingegen können toleriert werden [Nguyen, 2018] [Cyl, 2019].

Es existiert noch ein weiterer Punkt, welcher die Korrektheit einschränkt, und dies ist Point-To-Analyse. An dieser Stelle wird auf die Arbeit von [Cyl, 2019] verwiesen, welcher die Point-To-Analysen sehr gut und präzise beschreibt.

Kapitel 6

Implementierung

In dem Kapitel wird auf die Implementierung des Slicers eingegangen. Am Anfang werden die Problematiken (Abschnitt 6.1) konkretisiert, welche beim Testen des Slicers vor der Implementierung aufgetreten sind. In dem Abschnitt werden auch die Bug Fixes und die Updates präzisiert, welche durchgeführt worden sind, damit der Slicer benutzbar wurde. Darauffolgend werden in den Abschnitten 6.2 bis 6.5 die Veränderungen der einzelnen Seiten der Benutzeroberfläche sowie die dafür verantwortlichen Algorithmen erklärt. Zum Schluss wird dieses Kapitel im Abschnitt 6.6 zusammengefasst.

6.1 Problematiken vor der Implementierung

Das Aufsetzten und die Inbetriebnahme des Slicers bereiteten keine Probleme. Beim Austesten und Ausprobieren sind aber entweder Fehler aufgetreten oder der Slicer hat ein anderes Ergebnis geliefert, als vermutet wurde. Als Allererstes ist die Tatsache aufgefallen, dass die "*Slicer Options*"-Seite nicht aufrufbar ist. Es wird die Fehlermeldung "*The page does not exist.*" angezeigt. Dieses Problem wird im Abschnitt 6.5 behoben.

Das größere Problem bestand dain, dass beim Starten eines Slices eine *ClassCastException* geworfen worden ist und daher kein Slice-Ergebnis erzeugt worden ist. Nachdem dieses Problem beseitigt worden war, wurde ein weiterer Fehler festgestellt.
Der erste Fehler ist beim Erzeugen des Aufrufgraphen aufgetreten. Im Programmcode 6.1 ist nur die Message der Fehlermeldung zu erkennen, weil der Stracktrace an dieser Stelle nicht interessant ist. Der Fehler konnte mit dem Update der WALA Version von 1.5.3 auf die neueste Version 1.5.5 behoben werden. Die Version 1.5.5 wurde am 01.04.2020 veröffentlicht [WALA, 2020]. Die genauen Veränderungen in der Bibliothek durch die Aktualisierung sind nicht bekannt. Es hat sich aber durch die Evaluation gezeigt, dass die Aktualisierung keinen negativen Effekt hatte.

```
1 java.lang.ClassCastException: class com.ibm.wala.classLoader.JavaLanguage$JavaInstructionFactory$12 cannot be
      cast to class com.ibm.wala.ssa.SSAInvokeDynamicInstruction
```

Programmcode 6.1: Fehlermeldung beim Erzeugen eines Aufrufgraphen (Quelle: Eigene Darstellung)

Der zweite Fehler zeigte sich dann beim Rekonstruieren der Java-Dateien. Wenn eine Java-Datei in keinem *Package* lag, wurde eine *IndexOutOfBoundsException* ausgelöst, denn es wurde angenommen, dass ein Klassenname immer das "/" Symbol beinhaltet. Der Klassenname ist die Klasse, welche über die Felder "*Android Class Name*" und "*Class for searching the seed statement*" auf der "*Make a new Slice*"-Seite vom Benutzer angegeben worden ist. Im Programmcode 6.2 ist der problematische Quellcode aus der Methode *process* aus der Klasse *SliceService* zu sehen. Der überarbeitete Quellcode ist im Programmcode 6.3 zu finden.

```
1  // add the slice code to the slice entity
2  String javaClassFileName = packageAndJavaClass.substring(packageAndJavaClass.lastIndexOf("/") + 1,
       packageAndJavaClass.length());
3  String packagePath = packageAndJavaClass.substring(0, packageAndJavaClass.lastIndexOf("/"));
4  slice.getSlicedClasses().add(new SlicedClass(javaClassFileName, packagePath, builder.toString()));
```

Programmcode 6.2: SliceService.java - *process*-Methode alt (Quelle: Eigene Darstellung)

```
1  // add the slice code to the slice entity
2  String javaClassFileName;
3  String packagePath;
4  if (packageAndJavaClass.contains("/")) {
5    javaClassFileName = packageAndJavaClass.substring(packageAndJavaClass.lastIndexOf("/") + 1);
6    packagePath = packageAndJavaClass.substring(0, packageAndJavaClass.lastIndexOf("/"));
7  } else {
8    javaClassFileName = packageAndJavaClass;
9    packagePath = "";
10 }
11 slice.getSlicedClasses().add(new SlicedClass(javaClassFileName, packagePath, builder.toString(), new TreeSet<>(
       sourceCodeLineNumbers)));
```

Programmcode 6.3: SliceService.java - *process*-Methode neu (Quelle: Eigene Darstellung)

Des Weiteren ist aufgefallen, dass das gelieferte Slice-Ergebnis viele Klassen beinhalten kann, welche überhaupt nichts mit den angegebenen Parametern zu tun haben. Der Abbildung 6.1 ist das Slice-Ergebnis mit folgenden Parametern zu entnehmen:

Parameter	**Wert**
Systempath to jar	D:\Slicer_Jan\java-jars\LocalParameter\LocalParameter.jar
Systempath to source files	D:\Slicer_Jan\java-jars\LocalParameter
Class for searching the seed statement	objectTracking/SimpleWithParameter
Method for searching the seed statement	main
Seed Statement (method call)	fooMain
CFA Options	ZERO_ONE_CFA
Data Dependence Options	NO_HEAP
Control Dependence Options	NO_EXCEPTIONAL_EDGES

Abbildung 6.1: Falsches Slice-Ergebnis (Quelle: Eigene Darstellung)

Es dürfte nur die Klasse *SimpleWithParameter* als Ergebnis geliefert werden, aber es wurden auch andere Klassen in den Slice mit aufgenommen. Dieses Problem resultiert bei der Ermittlung der Einstiegspunkte und wird später in der Arbeit behoben. Zum Schluss das Faktum ist noch aufgefallen, dass die Bezeichnungen der Einstellungsmöglichkeiten in den unterschiedlichen Slicemodes und in der Ansicht der fertiges Slices nicht einheitlich waren.

6.2 Make a new Slice

In diesen Abschnitt werden die Veränderungen der "*Make a new Slice*"-Seite und deren Algorithmen detailliert ausgeführt. Zuerst wird die neue Seite in den Abbildungen 6.2 und 6.3 dargestellt. Im Abschnitt 6.2.1 wird das Problem der unterschiedlichen Benennung der Einstellungsmöglichkeiten fokussiert. Danach werden die Veränderungen jeder einzelnen Einstellungsmöglichkeit der fertigen Benutzeroberfläche Punkt für Punkt beschrieben.

Abbildung 6.2: Ansicht zur Erstellung eines neuen Slices Teil 1 - Marcel (Quelle: Eigene Darstellung)

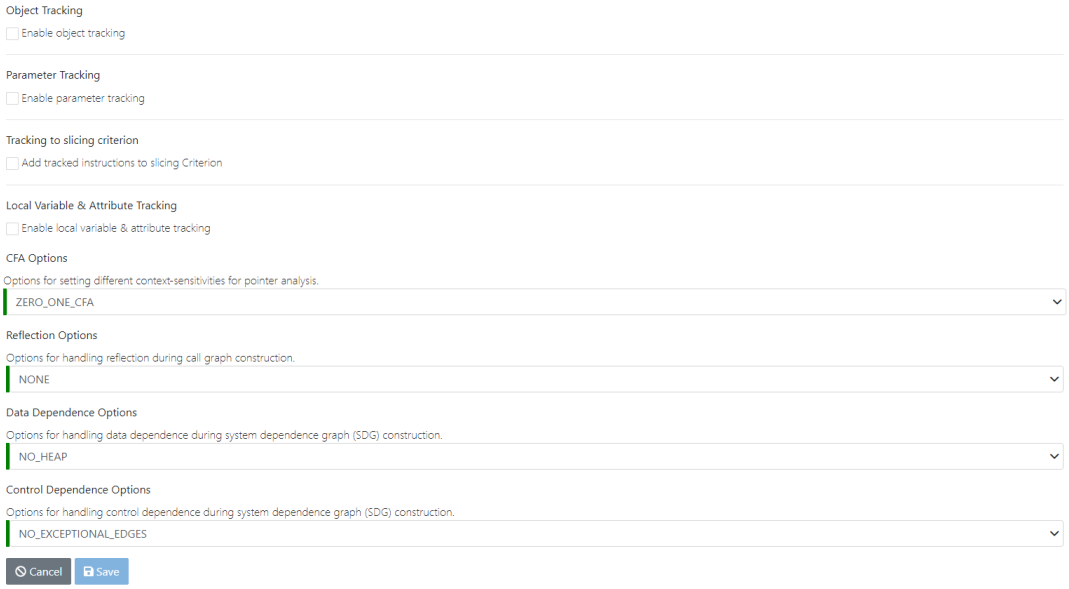


Abbildung 6.3: Ansicht zur Erstellung eines neuen Slices Teil 2 - Marcel (Quelle: Eigene Darstellung)

6.2.1 Problematik der Ungleichheit

Einer der ersten Schritte bei den Veränderungen an der GUI war die Vereinheitlichung der Einstellungsmöglichkeiten für das Erstellen eines neues Slices. Mit diesem Schritt sollen Teile der nicht-funktionalen Anforderungen N1 und N2 realisiert werden. Für Android und für Java wurden die gleichen Einstellungsmöglichkeiten in der GUI und als Attribute in der *Slice*-Klasse an einigen Stellen unterschiedlich benannt, welche dadurch die Wartbarkeit und Nutzbarkeit des Slices beeinträchtigten. Diese sind in Abbildung 6.4 veranschaulicht.

Java	Android	Attribute
Class for searching the seed statement	Android Class Name	className
Method for searching the seed statement	Entry Methods	entryMethods
Seed Statement (method call)	Seed Statements	seedStatements

Abbildung 6.4: Ungleiche Benennung der gleichen Einstellungsmöglichkeiten (Quelle: Eigene Darstellung)

Ziel ist es, den Einstiegspunkt- und den "Methoden für das Seed-Statement"-Algorithmus von Nguyen zu übernehmen und ggf. anzupassen. Dadurch würde auch das Problem gelöst werden, dass falsche Klassen im Slice-Ergebnis zu verzeichnen sind. An dieser Stelle wurde festgestellt, dass Cyl, Möhlmann und Nguyen jeweils bei "Method for searching the seed statement" und "Entry Methods" etwas anderes verstanden haben. Das Problem der unterschiedlichen Sichtweisen wurde auf der Seite 42 unter dem Punkt "Mögliche Methoden der Seed Statemens" beschrieben.

Beide Felder gehören zu dem Attribut *entryMethods* des Slices. *entryMethods* ist ein Set aus Methodennamen.

Wenn man sich nur die Namen der Einstellungsmöglichkeiten anschaut und weiß, wie der Slice-Algorithmus grob funktioniert, dann kommt man zunächst zu diesem folgenden Schluss:
"Entry Methods" (Android) Diese Einstellung wird für das Ermitteln der Einsteigpunkte benutzt, welche für den Aufrufgraphen verwendet werden.
"Method for searching the seed statement" (Java) Diese Einstellung wird für das Ermitteln der Methoden benutzt, in denen sich das Seed Statement befinden soll.

Dies war als eine der größten Herausforderungen zu verstehen, warum für etwas Unterschiedliches das gleiche Attribut *entryMethods* benutzt wurde. Schaut man sich den Slice-Algorithmus detailliert an, gelangt man zu einem anderen Ergebnis.

Vor der Erweiterung des Slicers wurde das Attribut *entryMethods* wie folgt genutzt:
Zum einen wird das Attribut, wie der Name andeutet, für die Ermittlung der Einsteigpunkte verwendet, aber nur dann, wenn Android als SliceMode gewählt worden ist, siehe Programmcode 6.5. An dieser Stelle muss der Methodenname mit einem Namen aus dem Set vom Attribut *entryMethods* gleich sein (Zeile 21). Außerdem muss der Klassenname entweder mit dem Attribut *className*, übereinstimmen oder der Klassenname fängt mit dem Attribut *className* gefolgt von dem $ Symbol (innere Klasse) an (Zeile 13-16). Für Java wird nur die WALA eigene *Util.makeMainEntrypoints* Methode verwendet, siehe Programmcode 6.4 Zeile 12.

```
1 private static Iterable<Entrypoint> getEntrypoints(Slice slice, AnalysisScope scope,
2                 IClassHierarchy classHierarchy,
```

```
3                   String className,
4                   SliceLogger logger) throws WalaException {
5   Iterable<Entrypoint> entrypoints;
6   Set<String> entryMethods = slice.getEntryMethods();
7   if (slice.getSliceMode().equals(SliceMode.ANDROID)) {
8     logger.log("\n== GET ENTRY POINTS FOR ANDROID ==");
9     entrypoints = EntryPointAlgorithms.getAndroidEntrypoints(scope, classHierarchy, className, entryMethods,
          logger);
10  } else { // sliceMode == JAVA
11    logger.log("\n== GET ENTRY POINTS FOR JAVA==");
12    entrypoints = com.ibm.wala.ipa.callgraph.impl.Util.makeMainEntrypoints(scope, classHierarchy);
13  }
14  if (!entrypoints.iterator().hasNext()) {
15    throw new WalaException("Failed to find any entry points from " + entryMethods
16      + " in " + className +"!");
17  } else {
18    logger.log("Number of entry points: " + StreamSupport.stream(entrypoints.spliterator(), false).count());
19  }
20  return entrypoints;
21 }
```

Programmcode 6.4: getEntrypoint-Algorithmus (Quelle: Eigene Darstellung)

```
1  public static Set<Entrypoint> getAndroidEntrypoints(AnalysisScope scope, IClassHierarchy classHierarchy, String
       androidClassName,
2                           Set<String> entryMethods, SliceLogger logger) {
3   Set<Entrypoint> entrypoints = new HashSet<Entrypoint>();
4
5   if (classHierarchy == null) {
6     throw new IllegalArgumentException("ClassHierarchy is null!");
7   }
8
9   for (IClass clazz : classHierarchy) {
10    if (clazz instanceof ShrikeClass & !clazz.isInterface() & isApplicationClass(scope, clazz)) {
11      String typeName = clazz.getName().toString();
12
13      // check if type name of current class equals android class name ...
14      if (typeName.equals(androidClassName)
15        // .. also check inner class names as well (e.g Lcom/android/server/AlarmManagerService$[InnerClassName
              ])
16        | typeName.startsWith(androidClassName + "$")) {
17
18        for (Iterator<? extends IMethod> methodIt = clazz.getDeclaredMethods().iterator(); methodIt.hasNext();)
              {
19          IMethod method = (IMethod) methodIt.next();
20          for (String entryMethod : entryMethods) {
21            if (!method.isAbstract() && method.getName().equals(Atom.findOrCreateUnicodeAtom(entryMethod))) {
22              entrypoints.add(new ArgumentTypeEntrypoint(method, classHierarchy));
23              logger.log("~ Found entry method: " + entryMethod + "() with object class: " + typeName + ".");
24            }
25          }
26        }
27      }
28    }
29  }
30  return entrypoints;
31 }
```

Programmcode 6.5: getAndroidEntrypoint-Algorithmus (Quelle: Eigene Darstellung)

Zum anderen wird das Attribut für die Ermittlung der "Methoden für das Seed-Statement" für Android sowie für Java benutzt. Unter Java muss eine Methode nur den gleichen Namen wie ein Name aus *entryMethods* haben, siehe Programmcode 6.6, Zeile 13. Im Programmcode 6.7 ist der Algorithmus für Android abgebildet. Dort wird außerdem der Klassenname genauso überprüft wie bei den Einstiegspunkten (Zeile 12-15). Dies stimmt aber nicht wirklich, denn durch den rekursiven Aufruf von *findMethodNodes* und den Aufruf der Methode *getInnerMethodNames* (Zeile 27-30) werden alle Methoden der Klasse als "Methoden für das Seed-Statement" ermittelt. Denn anders als der Name *getInnerMethodNames* andeu-

tet, gibt die Methode alle Methoden der Klasse *className* zurück. Somit ist das Attribut *entryMethods* für Android für die Ermittlung der "Methoden für das Seed-Statement" nutzlos.

```
1  Set<String> entryMethods = slice.getEntryMethods();
2  if (slice.getSliceMode() != SliceMode.JAVA) {
3    logger.log("\n== FIND METHOD(s) FOR SEED_STATEMENT(s) ==");
4    WalaSlicer.findMethodNodes(callGraph, entryMethods, methodNodes, className, logger);
5    if (methodNodes.size() == 0) {
6      throw new WalaException("Failed to find any methods from" + entryMethods + "!");
7    }
8  } else {
9    // SliceMode == Java => entryMethodes are the methods to search the seedstatement
10   callGraph.forEach(node -> {
11     Atom methodNodeName = node.getMethod().getName();
12     for (String seedMethod : entryMethods) {
13       if (methodNodeName.equals(Atom.findOrCreateUnicodeAtom(seedMethod)))
14         methodNodes.add(node);
15     }
16   });
17 }
```

Programmcode 6.6: Java Algorithmus (Quelle: Eigene Darstellung)

```
1  private static Set<CGNode> findMethodNodes(final CallGraph callGraph, Set<String> methodNames,
2                      final Set<CGNode> methodNodes, final String className,
3                      SliceLogger logger) throws WalaException {
4    for (Iterator<? extends CGNode> nodeIt = callGraph.iterator(); nodeIt.hasNext();) {
5      CGNode node = nodeIt.next();
6
7      // get the method name
8      Atom methodNodeName = node.getMethod().getName();
9      // get the class name
10     TypeName declaringClassName = node.getMethod().getDeclaringClass().getName();
11
12     // check if class name of current node equals class name ...
13     if (declaringClassName.equals(TypeName.findOrCreate(className))
14       // ... or inner class name
15       | declaringClassName.toString().startsWith(className + "$")) {
16
17       // check all given method names (i.e. entry methods and inner methods)
18       for (String methodName : methodNames){
19         // compare method name
20         if (methodNodeName.equals(Atom.findOrCreateUnicodeAtom(methodName))) {
21           // add node if not already in set
22           if(!methodNodes.contains(node)){
23             methodNodes.add(node);
24             logger.log("~ Found call graph method: " + methodName + "() with object class: "
25             + declaringClassName + " in " + node + ".");
26             // search inner method nodes
27             Set<String> innerMethodNames = getInnerMethodNames(node);
28             if(!innerMethodNames.isEmpty()){
29               findMethodNodes(callGraph, innerMethodNames, methodNodes, className, logger);
30             }
31           }
32         }
33       }
34     }
35   }
36   return methodNodes;
37 }
```

Programmcode 6.7: Android Algorithmus (Quelle: Eigene Darstellung)

Somit war das Attribut *entryMethods* für die Bestimmung zum einen der Einstiegspunkte (Android) und zum anderen der Methoden ("Android", Java) zuständig. Das Attribut *className* war für die Bestimmung der Einstiegspunkte (Android) und der Methoden (Android) zuständig. Weil unter anderem *className* bei der Bestimmung der Einstiegs-

punkte nicht für Java verwendet worden ist, wurden andere Klassen ins Slice-Ergebnis aufgenommen, welche mit dem Ergebnis nicht in Verbindung standen.

src_caller wurde bei Nguyen als das Attribut *entryMethods* bezeichnet. Das Attribut war nur für die Bestimmung der Methoden (Java) zuständig. Das Attribut *mainClass* wurde nur für die Einstiegspunkte verwendet. Der Programmablauf wurde in der Abbildung 3.2 veranschaulicht.

Damit die Algorithmen von Nguyen übernommen und angepasst werden können, müssen die Einstellungsmöglichkeiten für Android und für Java einmal gleich benannt sein, und zwar, sowohl als Attribute als auch in der Benutzeroberfläche. Außerdem müssen die Attribute den gleichen Einfluss auf die Algorithmen ausüben. Jedes Attribut darf somit nur noch in einem Algorithmus benutzt werden. Die Attribute *className* und *entryMethods* wurden verworfen und durch die neuen Attribute *entryPointClass*, *entryPointMethods*, *methodClass* und *methodNames* ersetzt. *entryPointClass* und *entryPointMethods* sind nur für die Einstiegspunkte und *methodClass* und *methodNames* nur für die "Methoden für das Seed-Statement" zuständig.

Diese Aufsplittung bietet die folgenden Vorteile:

- Diese Vereinheitlichung soll spätere Verwirrungen bei Entwicklern und Benutzern überflüssig machen. Daher steigert dies die Wartbarkeit N1 sowie die Benutzbarkeit N2.
- Erstmals kann genau festlegt werden, welche Methode in welcher Klasse als Einstiegspunkt oder als "Methode für das Seed-Statement" verwendet werden kann.
- Die gefundenen Einstellungsmöglichkeiten können komplett unterschiedlich sein und in unterschiedlichen Klassen liegen. Dies war vorher nicht möglich.
- Alle vier Attribute dürfen auch *null* sein. Dadurch kann die Analyse grober ausfallen. Im Slicemode Android muss aber *entryPointClass* angegeben werden.

In Abbildung 6.2 wurde der veränderte Teil der Benutzeroberfläche für das Erstellen eines neues Slices im Slicemode Java gezeigt. In diesen sind die neuen vier Felder *Entrypoint Class*, *Entrypoint Methods*, *Method Class* und *Method Names* sowie die neuen Felder *Java Jar Name* und *Seed Statement Classes* zu erkennen. Auf alle diese Felder und deren Algorithmen wird in den nächsten Abschnitten detailliert eingegangen.

6.2.2 Android Version & Java Jar Name

Der einzige Unterschied zwischen dem Slicemode Android und Java in der veränderten Benutzeroberfläche „*Make a new Slice*“ besteht bei den Feldern *Android Version* und *Java Jar Name*. *Android Version* wurde nicht verändert und wird daher nicht weiter betrachtet.

Durch Möhlmann konnte der Slicer wieder Java-Anwendungen analysieren. Er hatte sich dafür entschieden, dass der Server auf dem Rechner des Benutzers laufen muss, weil er verhindern wollte, dass keine zu großen Datenmengen hochgeladen werden. Diese Umsetzung macht aber die Server-Client-Architektur sinnlos, welche auch die Benutzbarkeit N2 und Portierbarkeit N4 einschränkt. Wenn es nur einen Benutzer gibt, dann wäre die aktuelle Umsetzung gut. Weil aber davon nicht ausgegangen werden kann, würde die aktuelle Umsetzung bei mehreren Benutzern zu Problemen führen. Eine Zusammenarbeit dieser Benutzer ist derzeit nicht möglich, weil z.B. der Zugriff auf die Slice-Ergebnisse fehlt. Das bedeutet, die Benutzer müssten sich diese gegenseitig zuschicken, wobei diese dann nicht in den Slicer eingebunden werden können.

Deswegen wurden im Zuge der Änderungen die beiden Felder "Systempath to jar" und "Systempath to source files" verworfen und durch "Java Jar Name" ersetzt. In Abbildung 6.5 ist dieses Feld abgebildet.

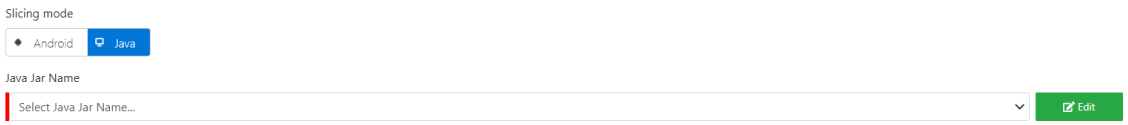


Abbildung 6.5: Javar Jar Name (Quelle: Eigene Darstellung)

Auf der rechten Seite befindet sich der neue *Edit*-Button, mit welchem die Anforderung AF22 umgesetzt wird. Über diesen können neue Zip-Dateien hochgeladen werden oder alte gelöscht werden. Nachdem auf diesen Button geklickt worden ist, wird ein Pop-up-Fenster angezeigt, siehe Abbildung 6.6.

Um eine Zip-Datei hochzuladen, muss entweder auf das graue Feld mit dem Text "Click or drag .zip file here to upload" geklickt werden, dann öffnet sich ein Fenster zum Durchsuchen der Dateien von der Festplatte oder eine Zip-Datei wird auf das graue Feld gezogen. Eine Zip-Datei muss genau eine Jar-Datei beinhalten und die Zip-Datei und Jar-Datei müssen gleich benannt sein. Die Jar-Datei darf sich in keinem weiteren Ordner befinden. Zusätzlich kann die Zip-Datei die Java-Dateien für das Rekonstruieren beinhalten. Auch in diesen Zusammenhang muss darauf geachtet werden, dass die Java-Dateien in dem richtigen Ordner liegen. Diese müssen in genau dem Ordner liegen, welcher das Package vorgibt (dies wird aber durch das System nicht überprüft). Die Ordnerstruktur wird an zwei Beispielen verdeutlicht:

- Liegt die *Hallo.java* Datei im Package *de.uni*, dann gibt es in der Zip-Datei den Ordner *de*, welcher den Ordner *uni* enthält, welcher die Java-Datei beinhaltet.
- Liegt die *Test.java* Datei in keinem Package, dann liegt die Java-Datei auch in keinem Ordner innerhalb der Zip-Datei.

Die hochgeladene Zip-Datei wird auf dem Server entpackt. Die entpackten Dateien werden im neuen Ordner (*Name der Zip-Datei*) im Ordner "java-jars" abgespeichert. Falls vorher

der Ordner (*Name der Zip-Datei*) existiert hatte, wird dieser mit allen Dateien vorher gelöscht. Der Ordner "java-jars" ist ein Standardwert, welcher über den Settings-Tab beim Key *Java_Directory_Path* abgeändert werden kann. Wird dieser abgeändert, kann aber nicht mehr auf die bereits hochgeladenen Dateien zugegriffen werden, außer der Key wird wieder auf den Standardwert zurückgesetzt.

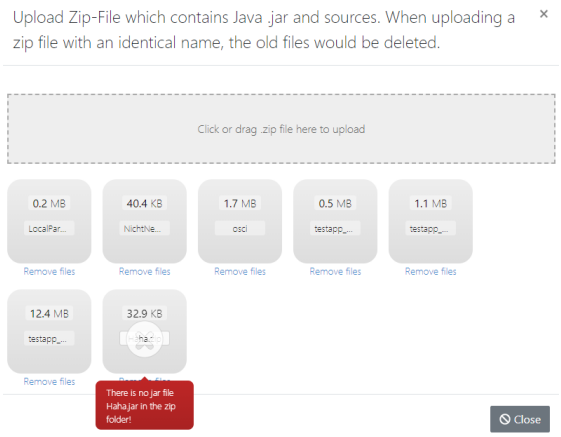


Abbildung 6.6: Zip Datei hochladen (Quelle: Eigene Darstellung)

Unter dem Feld zum Hochladen der Zip-Datei werden alle hochgeladenen Dateien mit Namen und Größe der Datei aufgelistet. Falls das Hochladen der Zip-Datei fehlgeschlagen ist, wird dies mit einem grauen Kreuz markiert. Wenn die Maus auf dieses graue Kreuz bewegt wird, wird die entsprechende Fehlermeldung angezeigt, warum das Hochladen schiefgelaufen ist. An der Abbildung 6.6 ist zu erkennen, dass versucht worden ist, die Zip-Datei namens "Haha.zip" hochzuladen. Dieser Versuch ist aber deshalb fehlgeschlagen, weil die Zip-Datei keine Jar-Datei beinhaltet, welche den Namen "Haha.jar" trägt.

Nachdem das Löschen oder das Hochladen abgeschlossen worden ist, kann das Fenster über den *Close*-Button geschlossen werden. Dann kann über das Feld *Java Jar Name* der passende Name ausgesucht werden, um eine Jar-Datei zu analysieren. Damit wurde die Anforderung AF04 umgesetzt.
Wenn ein Jar-Name ausgewählt worden ist, werden die Felder *Entrypoint Class* und *Method Class* deshalb deaktiviert, weil im Hintergrund die Klassenhierarchie durch WALA erzeugt wird. Die beiden Felder *Entrypoint Class* und *Method Class* werden mit allen gefundenen Klassen gefüttert, welche auch rekonstruiert werden könnten. Unter dem Füttern ist zu verstehen, dass die gefundenen Klassen später als Suchvorschläge angezeigt werden (Anforderung AF06 & AF11). Vorher wurde das alte Feld *Class for searching the seed statement* mit allen Klassen als Suchvorschlag beladen.

Dadurch wurde die Auswahl einer Klasse sehr langsam und die Benutzeroberfläche reagierte bei dem Feld sehr zeitverzögert. Daher werden bei den neuen Feldern nur die Klassen angezeigt, welche auch rekonstruiert werden können. Nachdem das Füttern abgeschlossen worden ist, werden die beiden Felder wieder aktiviert. Zusätzlich wird die Klassenhierarchie abgespeichert, damit bei einer erneuten Anfrage für diese der Jar-Name und die Klassenhierarchie nicht erneut erzeugt werden müssen.

Falls aber eine Zip-Datei mit einem Namen hochgeladen wird, welcher bereits vorhanden ist, wird nicht nur die vorhandene Dateien für diesen Namen gelöscht und ersetzt, sondern auch die gespeicherte Klassenhierarchie kann dann ignoriert werden, wenn die Uhrzeit der letzten Veränderung der Jar-Datei nicht mit der alten Jar-Datei übereinstimmt. Wenn dies der Fall ist, würde eine neue Klassenhierarchie erzeugt werden.

6.2.3 Entrypoint Class & Entrypoint Methods

Wenn in der Benutzeroberfläche eine *Entrypoint Class* im Slicemode Java ausgewählt wird (Anforderung AF05), wird zusätzlich eine Checkbox "*Only Public Methods*" unter *Entrypoint Methods* angezeigt, siehe Abbildung 6.3. Wird diese Checkbox aktiviert, werden bei der Ermittlung der Einstiegspunkte nur *public*-Methoden in Betracht gezogen.
Zusätzlich wird bei der Auswahl der *Entrypoint Class* der Quellcode der Java-Datei angezeigt (Anforderung AF09). Neben diesem Feld gibt es den neuen *Clear*-Button, welcher den Eintrag und den angezeigten Quellcode für *Entrypoint Class* löscht. Im Slicemode Java wird somit gewährleistet, dass *Entrypoint Class* auch *null* sein darf. Im Slicemode Android muss dieses Feld angegeben werden. Wenn nichts angegeben wird, wird dies zum einen an der linken Seite in roter Farbe und zum anderen mit dem Text "*This field is required.*" markiert.
Für *Entrypoint Methods* gibt es wieder einen *Add All*- und einen *Clear*-Button, welche entweder alle Methoden der ausgewählten Klasse hinzufügen oder löschen. Diese Buttons fehlten im Slicemode Java. Der Benutzer kann an dieser Stelle *Entrypoint Methods* hinzufügen (Anforderung AF07), welche durch den Slicer gefiltert worden sind (Anforderung AF08).

Abbildung 6.7: Entrypoint Class und Entrypoint Methods (Quelle: Eigene Darstellung)

Für den Algorithmus für die Anforderungen AF05 und AF07 für beide Slicemodes wird grob Folgendes gelten:

- Wird eine *Entrypoint Class* angegeben, dürfen sich die Einstiegspunkte nur in der angegebenen (Inneren)-Klasse befinden.

- Wird keine *Entrypoint Class* angegeben (nur Slicemode Java), darf der Einstiegspunkt in jeder Klasse sein.

- Werden ein oder mehrere *Entrypoint Methods* angegeben, dürfen die Namen der Einstiegspunkte nur den Namen aus *Entrypoint Methods* entsprechen.

- Wird nicht *Entrypoint Methods* angegeben, werden alle Methoden aus der gefundenen Klasse als Einstiegspunkte verwendet.

Dadurch kann im extremen Fall nur eine Methode einer Klasse als Einstiegspunkt genommen werden. Für den Algorithmus der Ermittlung der Einstiegspunkte für Android verändert sich nur der Teil, dass entweder alle Methoden der Klasse oder nur einzelne Methoden der Klasse als Einstiegspunkte verwendet werden. Im Programmcode 6.8 ist der alte Teilcode vom Algorithmus zu erkennen. Im Programmcode 6.9 ist die veränderte Variante zu sehen.

```
for (String entryMethod : entryMethods) {
  if (!method.isAbstract() && method.getName().equals(Atom.findOrCreateUnicodeAtom(entryMethod))) {
    entrypoints.add(new ArgumentTypeEntrypoint(method, classHierarchy));
    logger.log("~ Found entry method: " + entryMethod + "() with object class: " + typeName + ".");
  }
}
```

Programmcode 6.8: getAndroidEntrypoint Algorithmus Ausschnitt - alt (Quelle: Eigene Darstellung)

```
if (entryPointMethods != null && !entryPointMethods.isEmpty()) {
  for (String methodName : entryPointMethods) {
    if (!method.isAbstract() && method.getName().equals(Atom.findOrCreateUnicodeAtom(methodName))) {
      entrypoints.add(new ArgumentTypeEntrypoint(method, classHierarchy));
      logger.log("~ Found method: " + methodName + "() with object class: " + typeName + ".");
    }
  }
} else {
  if (!method.isAbstract()) {
    entrypoints.add(new ArgumentTypeEntrypoint(method, classHierarchy));
    logger.log("~ Found method: " + method.getName() + "() with object class: " + typeName + ".");
  }
}
```

Programmcode 6.9: getAndroidEntrypoint Algorithmus Ausschnitt - neu (Quelle: Eigene Darstellung)

Für Java wurde der bestehende Algorithmus, welcher nur aus einer Zeile bestand, verworfen. Der Algorithmus von Nguyen wurde übernommen und in die neue Methode *getJavaEntrypoints* ausgelagert. Dieser Algorithmus wurde so angepasst, dass, falls *Entrypoint Methods* angegeben worden sind, nur noch solche Einstiegspunkte genommen werden, bei denen der Methodenname mit einem Namen aus *Entrypoint Methods* übereinstimmt.

Außerdem wurden die beiden Teilalgorithmen von (Gulmann/Gerken) und (Detmers) zusammengelegt, weil beide sich nur durch die Überprüfung von *onlyPublicEntry* unterschieden. Der neue Programmablauf ist der Abbildung 6.8 zu entnehmen. Nguyen hat in seiner Evaluation nachgewiesen, dass sein Algorithmus funktioniert, nur dass kein explizierter Einstiegspunkt ausgesucht werden kann. Dieses Problem wurde mit der Erweiterung *Entrypoint Methods* gelöst.

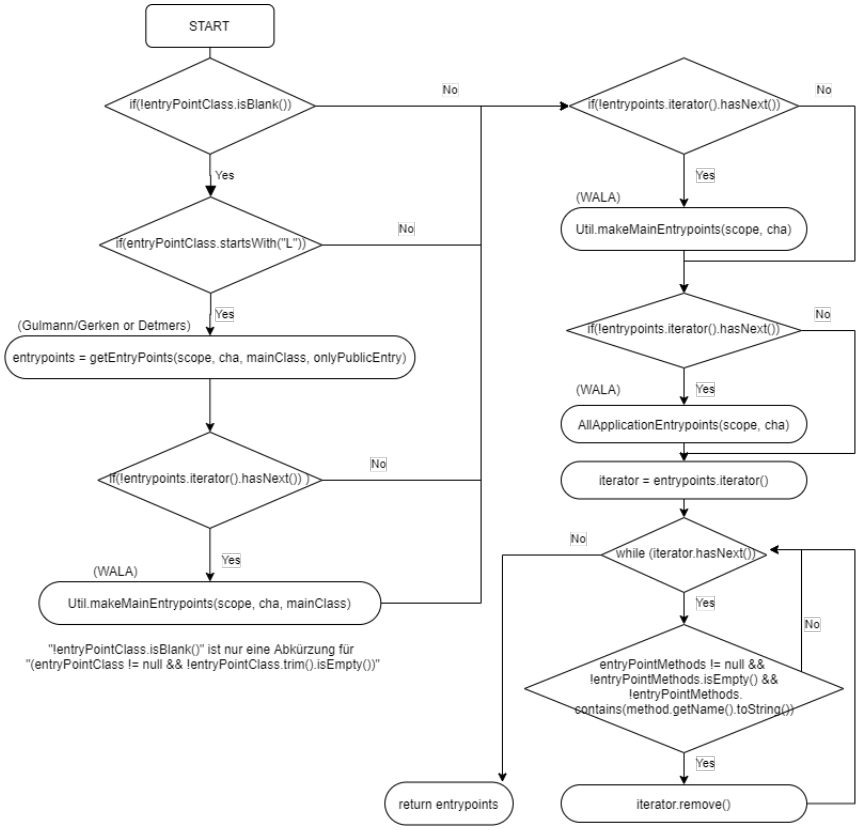


Abbildung 6.8: Programmablauf *getJavaEntrypoints()* (Quelle: Eigene Darstellung)

6.2.4 Method Class & Method Names

Bei der Auswahl einer *Method Class* (Anforderung AF10) wird wie beim *Entrypoint Class* der Quellcode der Java-Datei angezeigt (Anforderung AF14). Wird bei der Auswahl aber die gleiche Klasse ausgewählt, wird kein zusätzlicher Quellcode angezeigt. Damit soll verhindert werden, dass es nicht zu Verwirrungen beim Benutzer kommt, wenn der gleiche Quellcode zwei Mal angezeigt wird. Dies sollte die Benutzbarkeit steigern. In Abbildung 6.9 ist ein Beispiel zu erkennen, in dem die *Method Class* genau die gleiche Klasse wie die *Entrypoint Class* aus der Abbildung 6.7 hat.
Rechts neben den Feldern gibt es wieder entweder nur den *Clear*-Button oder den *Add All*- und *Clear*- Button. Der Benutzer kann an dieser Stelle *Method Names* hinzufügen (Anforderung AF12), welche durch den Slicer gefiltert worden sind (Anforderung AF13).

Abbildung 6.9: Method Class und Method Names (Quelle: Eigene Darstellung)

Zum Ermitteln der "Methoden für das Seed-Statement" wird nicht mehr zwischen den Slicemodes Java und Android differenziert, denn es gibt keinen Grund, warum an dieser Stelle zwischen den beiden unterschieden werden müsste.

In die neue Klasse *MethodNodesAlgorithms* wurde die Methode *findMethodNodes* ausgelagert und erweitert. Die Methode *getInnerMethodNames*, welche alle Methodennamen einer Klasse liefert, wurde im Zuge der Änderungen verworfen. Denn in der Methode *findMethodNodes* wird schon durch den Aufrufgraphen iteriert und somit werden alle Methoden besucht und ggf. als gefundene Methoden zurückgegeben.

Der folgende Algorithmus ist im Programmcode 6.10 zu sehen. Der alte Code wird im Programmcode 6.6 angezeigt. Als Erstes wird der String *methodClass* festgelegt. Entweder ist dieser *null*, wenn der Benutzer keine *Method Class* angegeben hat, oder der Klassenname wird in WALA-Form gebracht. Dies bedeutet, dass zum einen die Extension der Datei entfernt wird und dass zum anderen ein "*L*" vor den Klassennamen kommt.
Als Nächstes wird über den Aufrufgraphen iteriert. Ein Knoten (Methode) aus dem Aufrufgraphen wird nur dann weiter betrachtet, wenn eine der folgenden Bedingungen zutrifft:

1. *methodClass* ist *null*.
2. Der Klassenname der Methode ist gleich *methodClass*.
3. Der Klassenname der Methode fängt mit *methodClass* gefolgt von dem "$" Symbol an.
4. *methodClass* fängt mit dem Klassenname der Methode an und *methodClass* beinhaltet das "$" Symbol.

Punkt 1 und Punkt 4 sind neu im Vergleich zum alten Code. Der 3. und der 4. Punkt sind die interessanten Fälle. Wenn die angegebene *methodClass* ein oder mehrere Innere-Klassen hat, dann werden durch den 3. Punkt die Methoden der Inneren-Klassen berücksichtigt. Durch den 4. Punkt ist es nun möglich, auch eine Innere-Klasse als *methodClass* anzugeben und dabei werden die Methoden der Äußeren-Klasse berücksichtigt. Dies war vorher nicht möglich.

Wenn dann eine Methode eine der vier Bedingungen erfüllt, wird zwischen zwei Fällen unterschieden. Entweder ist *methodNames null* bzw. leer, dann werden alle Methoden der Klasse den *methodNodes* hinzugefügt, oder *methodNames* ist nicht leer, dann werden nur die Methoden hinzugefügt, wenn der Methodennamen mit einem Namen aus *methodNames* übereinstimmt. Die gefundenen *methodNodes* werden dann geloggt, wenn die Anzahl von *methodNodes* nicht größer als *WalaSlicer.MAX_DETAILED_LOG* ist. *WalaSlicer.MAX_DETAILED_LOG* beträgt 300. Wenn keine *methodNodes* gefunden worden sind, wird eine *WalaException* geworfen, sonst werden zum Schluss die *methodNodes* zurückgegeben.

```
1 public static Set<CGNode> findMethodNodes(final Slice slice,
2                     final CallGraph callGraph,
```

```
3                       final Set<CGNode> methodNodes,
4                       final SliceLogger logger) throws WalaException {
5
6   // add "L" to class name and remove .java extension
7   // e.g. com/android/server/AlarmManagerService.java
8   // -> Lcom/android/server/AlarmManagerService
9   String methodClass = slice.getMethodClass() == null ? null : "L" + FilenameUtils.removeExtension(slice.
        getMethodClass());
10  Set<String> methodNames = slice.getMethodNames();
11
12  for (CGNode node : callGraph) {
13    // get the method name
14    Atom methodNodeName = node.getMethod().getName();
15    // get the class name
16    TypeName declaringClassName = node.getMethod().getDeclaringClass().getName();
17
18    // check if class name of current node equals class name ...
19    if (methodClass == null || declaringClassName.equals(TypeName.findOrCreate(methodClass))
20      // ... or inner class name
21      || declaringClassName.toString().startsWith(methodClass + "$")
22      || (methodClass.startsWith(declaringClassName.toString()) && methodClass.contains("$"))) {
23
24      // check all given method names
25      if (methodNames == null || methodNames.isEmpty()) {
26        methodNodes.add(node);
27      } else {
28        for (String methodName : methodNames) {
29          // compare method name
30          if (methodNodeName.equals(Atom.findOrCreateUnicodeAtom(methodName))) {
31            // add node if not already in set
32            methodNodes.add(node);
33          }
34        }
35      }
36    }
37  }
38  if (!methodNodes.isEmpty()) {
39    if (methodNodes.size() > WalaSlicer.MAX_DETAILED_LOG) {
40      logger.log("Number of Method Nodes for Seed Statements exceeded maximum Statements to log(" + WalaSlicer.
            MAX_DETAILED_LOG + "). Therefore Method Nodes will not be logged in detail.");
41    } else {
42      for (CGNode node : methodNodes) {
43        logger.log("~ Found call graph method: " + node.getMethod().getName() + "() with object class: " + node
              .getMethod().getDeclaringClass().getName() + " in " + node + ".");
44      }
45    }
46  } else {
47            logger.log("WARNING: Failed to find any method nodes from the method class " + slice.getMethodClass
                () + " for the method names " + methodNames + "!\n\n");
48            throw new WalaException("Failed to find any methods from the method class " + slice.getMethodClass
                () + " for the method names " + methodNames + "!");
49  }
50  return methodNodes;
51 }
```

Programmcode 6.10: findMethodNodes - neu (Quelle: Eigene Darstellung)

6.2.5 Seed Statements & Seed Statement Classes

Im Slicemode Java in der Benutzeroberfläche konnte vorher nur ein Seed-Statement angegeben werden, es fehlten die Buttons *Add All* und *Clear*. Außerdem wurde das Feld nicht als benötigt markiert. Das bedeutet, dass links die rote Markierung sowie der Text "*This field is required.*" dann gefehlt haben, wenn kein Seed-Statement angegeben worden ist. Wie in der Abbildung 6.2 zu erkennen ist, wurde dies behoben (Anforderung AF15 & AF16).

Das Feld *Seed Statement Classes* ist neu. Die Idee und ein Teil des Algorithmus wurden

dafür von Nguyen übernommen. Durch *Seed Statement Classes* kann genauer angegeben werden, aus welchen Klassen die Methoden (Seed-Statements) stammen (Anforderung AF17). An dem im Programmcode 6.11 gezeigten Quellcode wird ein Beispiel für die Benutzung der *Seed Statement Classes* gezeigt.

```
1 public void main(){
2   ClassA classA = new ClassA();
3   ClassB classb = new ClassB();
4
5   classA.doSomething():
6   classB.doSomething():
7 }
```

Programmcode 6.11: Beispiel - Seed Statement Classes (Quelle: Eigene Darstellung)

Als Seed-Statement wird *doSomething* verwendet. Wird kein *Seed Statement Classes* angegeben, würden beide Aufrufe von *doSomething* (Zeile 5 + 6) im Slice-Ergebnis enthalten sein. Wird aber *ClassA* als *Seed Statement Classes* angegeben, würde nur der Aufruf *doSomething* (Zeile 5) von der Klasse *ClassA* im Slice-Ergebnis enthalten sein.

Ein *Seed Statement Class* kann in einer von drei Varianten angegeben werden:

- Nur der Klassenname. z.B.: *Signature*
- Package (Namen getrennt durch das ”.” Symbol) + Klassenname.
  z.B.: *java.security.Signature*
- Package (Namen getrennt durch das ”/” Symbol) + Klassenname.
  z.B.: *java/security/Signature*

In der neuen Klasse *SeedStatementAlgorithms* wurde die Methode *findSeedStatements* ausgelagert und erweitert. Im Programmcode 6.12 ist der alte Teilalgorithmus und im Programmcode 6.13 ist der neue Teilalgorithmus der Methode *findSeedStatements* zu sehen. Der Rest der Methode wurde entweder nicht angepasst oder ist deshalb nicht relevant, weil dort die Seed-Statements geloggt werden.

```
1  // add seed statements with method invoke instructions
2  if (instruction instanceof SSAAbstractInvokeInstruction) {
3    SSAAbstractInvokeInstruction call = (SSAAbstractInvokeInstruction) instruction;
4    String abstractInvokeInstructionName = call.getCallSite().getDeclaredTarget().getName().toString();
5
6    // check all seed statements
7    for (String seedStatementName : seedStatementNames) {
8      // mcyl: use regex for seed statements
9      if (Pattern.matches(seedStatementName, abstractInvokeInstructionName)) {
10       IntSet indices = ir.getCallInstructionIndices(call.getCallSite());
11       statements.add(new NormalStatement(node, indices.intIterator().next()));
12       logger.log("~ Found seed statement: " + call.getCallSite().getDeclaredTarget().getName().toString() + "
             in " + node + ".");
13     }
14   }
15 }
```

Programmcode 6.12: findSeedStatements - alt (Quelle: Eigene Darstellung)

```
1  // add seed statements with method invoke instructions
2  if (instruction instanceof SSAAbstractInvokeInstruction) {
3    SSAAbstractInvokeInstruction call = (SSAAbstractInvokeInstruction) instruction;
4    String abstractInvokeInstructionName = call.getCallSite().getDeclaredTarget().getName().toString();
5
6    // check all seed statements
7    for (String seedStatementName : seedStatementNames) {
8      // mcyl: use regex for seed statements
9      // note inner-classes with the symbol $. $ in Regex is a special character therefore must be replaced by \\
           $
10     if (Pattern.matches(seedStatementName.replace("$", "\\$"), abstractInvokeInstructionName)) {
11       if (seedStatementClasses == null || seedStatementClasses.isEmpty()) {
12         IntSet indices = ir.getCallInstructionIndices(call.getCallSite());
13         statements.add(new NormalStatement(node, indices.intIterator().next()));
14         logger.log("~ Found seed statement: " + abstractInvokeInstructionName + " in " + node + ".");
15       } else {
16         // check seed statement classes
17         for (String seedStatementClass : seedStatementClasses) {
18           if(seedStatementClass.contains(".")){
19             seedStatementClass = seedStatementClass.replace(".", "/");
20           }
21           // taken from nguyen and modified.
22           String abstractInvokeInstructionClassName;
23           if (seedStatementClass.contains("/")) {
24             abstractInvokeInstructionClassName = call.getCallSite().getDeclaredTarget().getDeclaringClass().
                   getName().toString();
25             abstractInvokeInstructionClassName = abstractInvokeInstructionClassName.substring(1); // remove L
26           } else {
27             abstractInvokeInstructionClassName = call.getCallSite().getDeclaredTarget().getDeclaringClass().
                   getName().getClassName().toString();
28           }
29           if (seedStatementClass.equals(abstractInvokeInstructionClassName)) {
30             IntSet indices = ir.getCallInstructionIndices(call.getCallSite());
31             statements.add(new NormalStatement(node, indices.intIterator().next()));
32             logger.log("~ Found seed statement: " + abstractInvokeInstructionName + " in " + node + ".");
33           }
34         }
35       }
36     }
37   }
38 }
```

Programmcode 6.13: findSeedStatements - neu (Quelle: Eigene Darstellung)

Der erste wichtige Unterschied ist bei der Überprüfung des Namens der aufgerufenen Methode und der Seed-Statements (Zeile 10) festzustellen. Für Regex ist "$" ein Sonderzeichen, daher muss das Symbol "$" durch "\\$" ersetzt werden. Dadurch ist es möglich, Methodenaufrufe als Seed-Statement anzugeben, welche ein "$" Symbol beinhalten. Diese speziellen Methodenaufrufe werden von WALA dann generiert, wenn es Klassen mit Inneren-Klassen gibt. Diese Methodenaufrufe werden aber nicht immer erzeugt. An dieser Stelle konnte nicht herausgefunden werden, in welcher Konstellation dies passiert.
Ein Beispiel für so eine erzeugte Methode namens *access$000* wird in Abbildung 6.10 veranschaulicht. Dabei verweist *access$000* auf die *print*-Methode. Wenn also entweder *access$000* oder *print* als Seed-Statement angegeben werden, wird das gleiche Slice-Ergebnis in diesem Fall erzeugt.

Abbildung 6.10: Beispiel einer erzeugten Methode durch WALA (Quelle: Eigene Darstellung)

Der zweite wichtige Unterschied zum alten Algorithmus besteht darin, dass wieder *Seed Statement Classes* berücksichtigt werden. Nguyen bezeichnete dies als *src_callee_class*. Wird kein *seedStatementClasses*-Attribut angeben, werden alle Methodenaufrufe als Seed-Statement genommen, welche die Überprüfung des Namens bestanden haben (Zeile 12-14). Wurden aber ein oder mehrere *seedStatementClasses* angegeben, wird als Erstes über diese iteriert und ggf. werden alle "." im Namen durch "/" ersetzt (Zeile 18-20).

Als zweiter Schritt muss der richtige Klassenname des Objekts, auf dem die Methode aufgerufen wird, in Abhängigkeit von der *seedStatementClass* ermittelt werden, dieser wird als *abstractInvokeInstructionClassName* gespeichert. Dies passiert in den Zeilen 22 bis 28. Wurden beim *seedStatementClass* der Packagename und der Klassenname angegeben, - daher beinhaltet *seedStatementClass* das "/" Symbol - , werden auch der Packagename und der Klassenname vom Objekt ermittelt.
Durch den Aufruf von *call.getCallSite().getDeclaredTarget()* erhält man die Methodenreferenz (*MethodReference*) und auf dieser erhält man mit dem Aufruf von *getDeclaringClass*, die "Klassenreferenz" (*TypeReference*).
Durch *getName().toString()* auf der *TypeReference* wird der Packagename + Klassenname zurückgegeben (Zeile 23-35). Wird *getName().getClassName().toString()* aufgerufen, wird nur der Klassenname zurückgegeben (Zeile 27).
Zum Schluss wird nur noch überprüft, ob der ermittelte Klassenname *abstractInvokeInstructionClassName* mit dem *seedStatementClass* übereinstimmt. Wenn dies der Fall ist, wird der Methodenaufruf als Seed-Statement hinzugefügt (Zeile 29-33).

6.2.6 Object & Parameter Tracking

Die Algorithmen für das Objekt- und das Parameter Tracking wurden in die Klasse *TrackingAlgorithms* ausgelagert (Anforderung AF18 & AF19). Die Methode *addObjectTracking* von Möhlmann wurde zu *interprocedural* umbenannt und erweitert. Dies liegt daran, dass zum einen sein Algorithmus interprozedural funktioniert und zum anderen dass eine neue Methode für das intraprozedurale Tracking mit dem Methodennamen *intraprocedural* gebaut wurden ist. Die *interprocedural*-Methode trackt zwar auch intraprozedural, aber es gibt Situationen, in denen diese Methode durch die *intraprocedural*-Methode unterstützt werden muss. Diese Methoden werden später genauer beschrieben. Zusätzlich wurde die Methode *addParameterTracking* verworfen, denn die Methoden für *interprocedural* und *intraprocedural* wurden so angepasst, dass diese auch das Parameter-Tracking unterstützen.

Bevor die einzelnen Methoden der *TrackingAlgorithms*-Klasse beschrieben werden, wird an einem Beispiel die Veränderung des Ergebnisses durch die Algorithmen verdeutlicht. Erzeugt wurde ein Slice mit den folgenden Einstellungen:

Parameter	Wert
Java Jar Name	LocalParameter
Entrypoint Class	objectTracking/InterproceduralComplex
EntryPoint Methods	main
Seed Statements	fooMain
CFA Options	ZERO_ONE_CFA
Data Dependence Options	FULL
Control Dependence Options	NO_EXCEPTIONAL_EDGES

In den gezeigten Programmcodes 6.14 und 6.15 sind zum einen die *InterproceduralComplex*- und zum anderen die *ClassDMoreMethods*-Klasse zu sehen. Die *main*-Methode der *InterproceduralComplex*-Klasse wird als Einstiegspunkt genutzt. Das Seed-Statement *fooMain* befindet sich in der *ClassDMoreMethods*-Klasse. Die restlichen Klassen werden in diesem Beispiel nicht weiter betrachtet. Um die Veränderungen und den Zweck der Methoden *interprocedural* und *intraprocedural* zu verdeutlichen, wird weiterhin nur die Klasse *ClassDMoreMethods* fokussiert.

```
1  package objectTracking;
2
3  public class InterproceduralComplex {
4
5      public static void main(String[] args) {
6          A classA = new A();
7          classA.init();
8
9          ClassBMoreMethods classB = new ClassBMoreMethods();
10         classB.doSomethingOne(classA);
11
12         ClassCMoreMethods.doSomethingTwo(classA);
13
14         classB.doSomethingThree(classA);
15         ClassDMoreMethods.doFooMain(classA);
16         classA.fooNot();
```

```
17      }
18  }
```

Programmcode 6.14: InterproceduralComplex - Klasse (Quelle: Eigene Darstellung)

```
1  package objectTracking;
2
3  public class ClassDMoreMethods {
4
5      public static void doFooMain(A classA){
6          classA.fooMain();
7          ClassB classB = new ClassB();
8          classB.doSomethingThree(classA);
9      }
10 }
```

Programmcode 6.15: ClassDMoreMethods - Klasse (Quelle: Eigene Darstellung)

Als Allererstes wurde ein Slice mit den gezeigten Einstellungen von dem fertigen Slicer erstellt, nur wurden die alten Objekt-Tracking Methoden von Möhlmann benutzt. Das Ergebnis ist in Abbildung 6.11 zu erkennen. Es ist zu sehen, dass die Zeilen 7 und 8 fehlen. In der gezeigten *Diff*-Darstellung fehlen alle rote Zeilen im Slice-Ergebnis. Zur Erinnerung sei gesagt, das erzeugte Slice-Ergebnis ist dann nicht falsch, wenn kein Objekt-Tracking aktiviert ist, nur für den Benutzer könnten wichtige Informationen fehlen, wie und wo das Objekt benutzt wird. In diesem Fall wurde das Objekt *classA* in der Methode *doSomethingThree* benutzt. Daher sollte das Objekt-Tracking mindestens Zeile 8 zusätzlich erkennen. Weil wahrscheinlich auch interessant sein könnte, woher das Objekt *classB* kommen könnte oder wo dieses noch benutzt wird, müsste noch Zeile 7 erkannt werden, denn dort wird das Objekt *classB* erzeugt. Dafür ist die Methode *intraprocedural* zuständig. Wichtig ist zu wissen, dass das Ergebnis der Methode *interprocedural* an *intraprocedural* übergeben wird. Das bedeutet, dass das bekannte Objekt-Tracking (das Tracking des Objekts vom Seed-Statement) nur von der *interprocedural*-Methode genutzt wird. Mit der *intraprocedural*-Methode werden alle Objekte von den Ergebnissen von *interprocedural* getrackt. Diese Art von Tracking ist neu und gilt auch für das Parameter-Tracking.

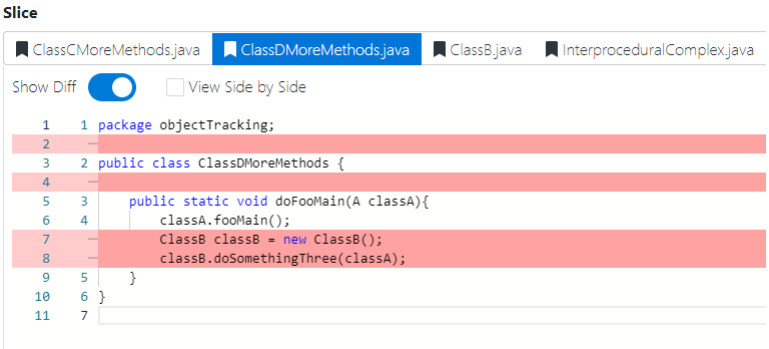


Abbildung 6.11: Slice-Ergebnis - ClassDMoreMethods - Alte Objekt-Tracking Methoden (Quelle: Eigene Darstellung)

In Abbildung 6.12 ist das Slice-Ergebnis mit der angepassten *interprocedural*-Methode zu

erkennen. Die Zeile 8 wurde richtig erkannt. Die Zeile 7 wurde dann mithilfe der *intraprocedural*-Methode identifiziert. Das Ergebnis ist der Abbildung 6.13 zu entnehmen.

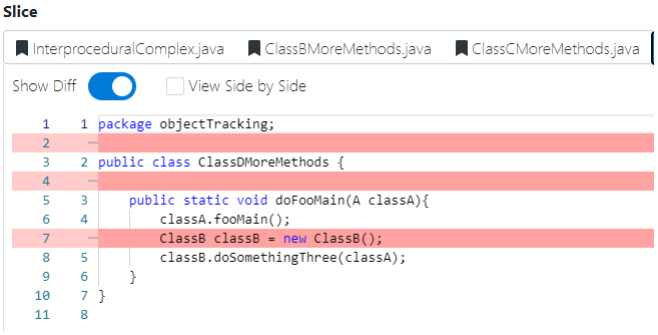


Abbildung 6.12: Slice-Ergebnis - ClassDMoreMethods - nur interprozedural (Quelle: Eigene Darstellung)

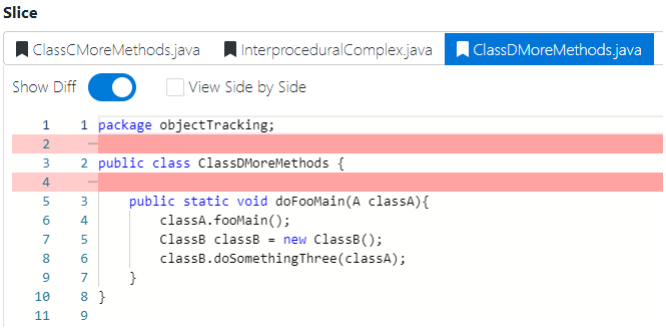


Abbildung 6.13: Slice-Ergebnis - ClassDMoreMethods - interprozedural und intraprozedural (Quelle: Eigene Darstellung)

Weil der Gedanke "das könnte auch interessant sein" immer weiter gespielt werden könnte, wurde darauf verzichtet, dass zum einen diese Art des Objekttrackings rekursiv ist und dass diese nur intraprozedural ist. Würde dies z.B. rekursiv ablaufen, hätte man sehr schnell ungewollte Ergebnisse im Slice-Ergebnis. Als Beispiel für ungewollte Ergebnisse, wurde der Tracking-Algorithmus mit den folgenden Einstellungen getestet:

Parameter	**Wert**
Java Jar Name	LocalParameter
Entrypoint Class	objectTracking/InterproceduralMoreObjects
EntryPoint Methods	main
Seed Statements	fooMain
CFA Options	ZERO_ONE_CFA
Data Dependence Options	FULL
Control Dependence Options	NO_EXCEPTIONAL_EDGES
Object Tracking	true

Abbildung 6.14: Slice-Ergebnis - InterproceduralMoreObjects - ohne rekursiven Aufruf (Quelle: Eigene Darstellung)

Abbildung 6.15: Slice-Ergebnis - InterproceduralMoreObjects - mit rekursiven Aufruf (Quelle: Eigene Darstellung)

In der Abbildung 6.14 ist das Slice-Ergebnis ohne den rekursiven Aufruf zu erkennen. Dass Seed-Statement *fooMain* befindet sich in der Methode *doFooMain*. Die *fooMain*-Methode wird daher auf das Objekt *classA* aufgerufen. Es ist in der Abbildung zu erkennen, dass alle Vorkommnisse des Objektes *classA* im Slice-Ergebnis enthalten sind. Dafür befindet sich das Objekt *classA1* richtigerweise nicht einmal im Slice-Ergebnis, weil darauf nie die Methode *fooMain* aufgerufen worden ist.

In der Abbildung 6.15 sind aber genau das Objekt *classA1* und alle Vorkommnisse des Objektes im Slice-Ergebnis erhalten. Dies passierte durch den rekursiven Aufruf.

Zusätzlich wurde überprüft, ob ein nachträgliches Tracking nach dem Slicing sinnvoll wäre und das Slice-Ergebnis verbessern würde. Es geht also darum, ob die *interprocedural*- und/oder *intraprocedural*-Methode nach der *computeBackwardSlice*-Methode verwendet werden kann. Es hat sich sehr schnell gezeigt, dass durch dieses Vorgehen sehr schnell gewünschte Zeilen im Slice-Ergebnis erscheinen. Dafür wurde zwei Slices mit folgenden Einstellungen erstellt:

Parameter	**Wert**
Java Jar Name	isCallerInRole
Entrypoint Class	clinic/ClinicExampleBean
Method Names	encryptData
Seed Statements	doFinal
CFA Options	ZERO_ONE_CFA
Data Dependence Options	FULL
Control Dependence Options	NO_EXCEPTIONAL_EDGES
Object Tracking	true
Tracking To Slicing Criterion	true

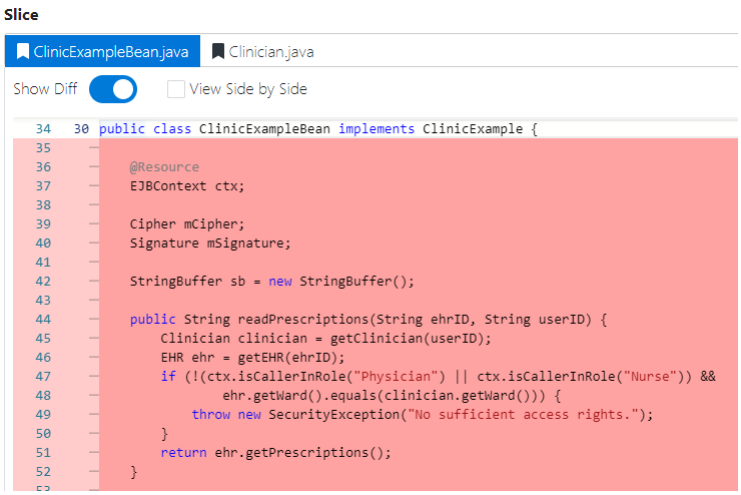


Abbildung 6.16: Slice-Ergebnis - ClinicExampleBean - ohne Tracking nach dem Slicing (Quelle: Eigene Darstellung)

Abbildung 6.17: Slice-Ergebnis - ClinicExampleBean - mit Tracking nach dem Slicing (Quelle: Eigene Darstellung)

In den Abbildungen 6.16 und 6.17 sind Ausschnitte aus den Slice-Ergebnisse ohne und mit Tracking nach dem Slicing zu erkennen. Die *encryptData*-Methode mit den Seed-Statement *doFinal* wird an dieser Stelle nicht weiter in den Blick genommen. Es ist zu erkennen, dass die Methode *readPrescriptions* fast vollständig im Slice-Ergebnis mit dem Tracking nach dem Slicing erhalten ist. Diese dürfte aber deshalb nie im Slice-Ergebnis sein, weil die Methode zum einen nie Aufgerufen wird und zum anderen kein Bezug zum Seed-Statement hat.

Im Folgenden werden alle Methoden beschrieben, welche für das Tracking zuständig sind.

6.2.6.1 tracking-Methode

Im Programmcode 6.16 ist die *tracking*-Methode zu sehen. Diese wird durch die *WalaSlicer*-Klasse aufgerufen. Wenn weder Objekt- noch Parameter-Tracking aktiviert ist, wird nur ein leeres *Set* zurückgegeben. Die *Slice*-Klasse bekam für das Tracking zwei neue Attribute, *interproceduralTracking* und *intraproceduralTracking*. Diese haben den Standardwert *true*. Der Benutzer kann diese nicht verändern, denn die beiden Attribute sind aktuell nur für Testzwecke gedacht. Über diese beiden Attribute kann dahin gehend gesteuert werden, ob nur interprozedural, nur intraprozedural oder ob beide Arten von Tracking genutzt werden sollen. Da die ersten beiden Fälle trivial sind, wird nur der Fall dann genauer erklärt, wenn beide Tracking-Arten genutzt werden. Dieser Teilalgorithmus besteht aus vier Teilen:

1. Es wird ein neues Set *interSliceList* aus den *seedStatements* erstellt.
2. Das Set *interSliceList* und andere Parameter werden an die Methode *interprocedural* übergeben. Die Methode befüllt *interSliceList* mit den gefundenen Statements.

3. Eine Kopie des Sets *interSliceList*, das Set *sliceList* und andere Parameter werden an die Methode *intraprocedural* übergeben. Die Methode befüllt *sliceList* mit gefundenen Statements.

4. Alle Statements aus *interSliceList* werden *sliceList* hinzugefügt.

Es hat einen besonderen Grund, warum entweder eine Kopie von einem Set erstellt wird (1. und 3. Punkt) oder ein Set alle Statements von einem anderen Set übernimmt (4. Punkt). Durch die Methodenaufrufe von *interprocedural* und *intraprocedural* werden die übergebenen Statements nicht mit in das Set übergeben, in welchem die gefundenen Statements gespeichert werden. Dadurch geht zum einen kein Ergebnis verloren und zum anderen wird genau an dieser Stelle die neue Art des Objekt-Trackings möglich, weil das Ergebnis weitergegeben wird. Würde aber nur *interprocedural* oder *intraprocedural* getrackt werden, dann würde nur das alte Tracking Verfahren benutzt werden, daher das Tracking des Objekts und/oder der Parameter der Seed-Statements.

```
1  public static Set<Statement> tracking(final SDG<InstanceKey> sdg, final Slice slice, final Set<Statement>
       seedStatements, final SliceLogger logger) {
2    Set<Statement> sliceList = new HashSet<>();
3    if (slice.isObjectTracking() || slice.isParameterTracking()) {
4      if (slice.isInterproceduralTracking() && slice.isIntraproceduralTracking()) {
5        Set<Statement> interSliceList = new HashSet<>(seedStatements);
6        interprocedural(sdg.getPointerAnalysis(), seedStatements, interSliceList, slice.isObjectTracking(), slice
             .isParameterTracking(), logger);
7        intraprocedural(sdg, new HashSet<>(interSliceList), sliceList, slice.isObjectTracking(), slice.
             isParameterTracking(), logger);
8        sliceList.addAll(interSliceList);
9      } else if (slice.isInterproceduralTracking() && !slice.isIntraproceduralTracking()) {
10       interprocedural(sdg.getPointerAnalysis(), seedStatements, sliceList, slice.isObjectTracking(), slice.
             isParameterTracking(), logger);
11     } else {
12       intraprocedural(sdg, seedStatements, sliceList, slice.isObjectTracking(), slice.isParameterTracking(),
             logger);
13     }
14   }
15   return sliceList;
16 }
```

Programmcode 6.16: Tracking Methode (Quelle: Eigene Darstellung)

6.2.6.2 interprocedural-Methode

Diese Methode wurde von Möhlmann übernommen, refaktoriert und nur ein wenig verändert. Deswegen wird nicht der gesamte Algorithmus erklärt, sondern nur der veränderte Teil. Der gesamte Algorithmus ist zu lang und ist daher im Anhang A.1 zu finden. Für die Erklärung des gesamten Algorithmus wird auf die Arbeit von Möhlmann verwiesen [Möhlmann, 2020]. Der veränderte Teil ist im Programmcode 6.17 zu sehen. Vorher hat der Algorithmus nur Objekt-Tracking unterstützt und mit der Änderung ist nun auch Parameter-Tracking möglich. Denn für das eigentliche Tracking werden die *Valuenumbers* von den *SSAAbstractInvokeInstruction* der Statements gebraucht.
Mithilfe einer Valuenumber kann innerhalb einer Methode ein Objekt genau verfolgt werden. Denn WALA benutzt diese als Identifikation der Objekte. Außerhalb der Methode

sind die Valuenumbers nutzlos. Von den gefundenen Valuenumbers werden die entsprechenden *LocalPointerKeys* gefunden, welche aus den Heapgraph der Pointeranalyse kommen. Ein *LocalPointerKey* beinhaltet den Knoten (Methode) und die Valuenumber des Objektes. Werden zwei *LocalPointerKeys* aus unterschiedlichen Zeilen miteinander verglichen und sind diese gleich, sind daher die Methode und die Valuenumber gleich, dann handelt es sich um das gleiche Objekt. Die *LocalPointerKeys* werden dabei durch die PointerAnalyse erzeugt.
Ein *SSAAbstractInvokeInstruction* hat nur eine Valuenumber für das Objekt, wenn die Instruktion nicht statisch ist. Wenn der Methodenaufruf statisch wäre, gäbe es auch kein Objekt, auf dem die Methode aufgerufen worden ist. Ist *objectTracking* aktiviert und die Instruktion nicht statisch, dann wird die Valuenumber über *getReceiver()* auf der Instruktion in *valueNumbers* abgespeichert.
Wenn das Parameter-Tracking aktiviert ist, dann werden alle Valuenumbers der benutzen Parameter hinzugefügt. Dies passiert, indem über die Anzahl der *numberOfUses* der Instruktion iteriert wird und für jede Zahl (i = 0 bis numberOfUses - 1) die Methode *getUse(i)* aufgerufen wird. *getUse* liefert die Valuenumber für die einzelnen Parameter. Dafür ist wichtig zu wissen, dass dann, wenn die Instruktion nicht statisch ist, die Iteration erst bei 1 und nicht bei 0 starten muss. Der Aufruf von *getUse(0)* würde bei einer nicht statischen Instruktion die Valuenumber des Objekts und nicht eines Parameters liefern. Mit den dann gefundenen *valueNumbers* wird das restliche Tracking fortgeführt.
Am Ende wird dann eine Map generiert, welche aus Knoten (*CGNode*) und einem Set aus Integer-Werten besteht. Die Map repräsentiert die vom Tracking gefundenen Knoten und die dazugehörigen Valuenumbers. Diese Valuenumbers sind Aliases der Valuenumbers, welche zuvor durch den hier gezeigten Teilalgorithmus gefunden worden sind. Zum Schluss wird diese Map an die Methode *addAliasesToSlice* übergeben. Anhand dieser Map werden die passenden Instruktionen der Knoten gefunden und als Statement abgespeichert.

```
1  if (objectTracking) {
2    // static method call have no receiver
3    if (!ssaAbstractInvokeInstruction.isStatic()) {
4      valueNumbers.add(ssaAbstractInvokeInstruction.getReceiver());
5    }
6  }
7
8  if (parameterTracking) {
9    int i = 0;
10   // If the method call is not static, then getUse(i) and i = 0 is always the object on which the method is
         called. But used are the parameters i>=1.
11   if (!ssaAbstractInvokeInstruction.isStatic()) {
12     i = 1;
13   }
14   for (; i < ssaAbstractInvokeInstruction.getNumberOfUses(); i++) {
15     valueNumbers.add(ssaAbstractInvokeInstruction.getUse(i));
16   }
17 }
```

Programmcode 6.17: interprocedural-Methode-Teilalgorithmus (Quelle: Eigene Darstellung)

6.2.6.3 addAliasesToSlice & isValueUsedByInstruction-Methoden

Bei den Methoden *addAliasesToSlice* und *isValueUsedByInstruction* handelt es sich um Helfer-Methoden von Möhlmann, welche für das interprozedurale-Tracking genutzt werden. Die Methode *addAliasesToSlice*, siehe Programmcode 6.18, bekommt unter anderem eine Map *aliasValueNumbers* und eine Collection *sliceList* übergeben. Die Methode iteriert über diese Map und überprüft für jede Instruktion des Knotens, ob eine Valuenumber von dieser Instruktion mit den gefundenen Alias-Valuenumbers übereinstimmt. Dies passiert mit der Hilfe der *isValueUsedByInstruction*-Methode. Wenn diese Übereinstimmung erfolgreich war, wird ein *NormalStatement* erzeugt und dieses wird der *sliceList* hinzugefügt. Aus dieser Methode wurde das "*Tracking to slicing criterion*" entfernt und in die Extramethode *trackingToSliceCriterion* ausgelagert. Diese wird aber nicht durch das Tracking selbst aufgerufen. Die Methode wird im Abschnitt 6.2.7 detailliert erklärt.

```
1  private static void addAliasesToSlice(HashMap<CGNode, Set<Integer>> aliasValueNumbers, Collection<Statement>
       sliceList, SliceLogger logger) {
2    Set<NormalStatement> loggerStatements = new HashSet<>();
3    // find the statements, that use these aliases
4    aliasValueNumbers.forEach((nodeWithAliasStatements, integers) -> {
5      if (nodeWithAliasStatements == null || nodeWithAliasStatements.getIR() == null) {
6        return;
7      }
8
9      SSAInstruction[] irInstructions = nodeWithAliasStatements.getIR().getInstructions();
10     for (int i = 0; i < irInstructions.length; i++) {
11       SSAInstruction ssaInstruction = irInstructions[i];
12       if (ssaInstruction == null) {
13         continue;
14       }
15       if (isValueUsedByInstruction(ssaInstruction, integers)) {
16         loggerStatements.add(new NormalStatement(nodeWithAliasStatements, i));
17         sliceList.add(new NormalStatement(nodeWithAliasStatements, i));
18       }
19     }
20   });
21
22   if (loggerStatements.size() > WalaSlicer.MAX_DETAILED_LOG) {
23     logger.log("Number of aliases statements for Seed Statements exceeded maximum Statements to log(" +
         WalaSlicer.MAX_DETAILED_LOG + "). Therefore aliases statements will not be logged in detail.");
24   } else {
25     loggerStatements.forEach(normalStatement -> logger.log("Addes Statement from class "
26       + normalStatement.getNode().getMethod().getDeclaringClass().getName()
27       + " in method " + normalStatement.getNode().getMethod().getName()
28       + ", instruction index " + normalStatement.getInstructionIndex()
29       + ", toString: " + normalStatement.getInstruction().toString()));
30   }
31 }
```

Programmcode 6.18: addAliasesToSlice-Methode (Quelle: Eigene Darstellung)

Die Methode *isValueUsedByInstruction* wird im Programmcode 6.19 dargestellt. Möhlmann konnte die Methode wegen der Corona-Pandemie nicht fertigstellen, weswegen ein paar Objekte nicht richtig getrackt wurden sind. Die Methode überprüft, ob die Valuenumbers der Instruktion mit einer Valuenumber aus dem übergebenen Set übereinstimmen. Dabei wird in drei Fälle differenziert:

1. (Zeile 2-7) Die Instruktion definiert eine neue Variable, dann wird über die *numberOfDefs* iteriert, mit i = 0 bis numberOfDefs - 1. Über *getDef(i)* wird die Valuenumber

der Instruktion für das Definieren geholt und überprüft, ob diese im Set vorhanden ist.

2. (Zeile 9-20) Die Instruktion ist eine *SSAAbstractInvokeInstruction*. Es wird wie bei der *interprocedural*-Methode über die *numberOfUses* iteriert, ob die Instruktion dabei statisch ist, ist unwichtig. Die gefundenen Valuenumbers werden genauso überprüft.

3. (Zeile 22-31) Die Instruktion ist eine *SSAPutInstruction*. Hier wird zwischen statisch und nicht-statisch unterschieden. Wenn die Instruktion statisch ist, wird die Valuenumber von *getVal()* genommen. Wenn die Instruktion nicht-statisch ist, wird die Valuenumber von *getVal()* und von *getRef()* genommen und überprüft. Zusätzlich darf die Valuenumber von *getRef()* nicht *1* sein, denn dann, wenn sie *1* wäre, würde die Instruktion außerhalb der Methode auf ein Attribut zeigen. Dieses gilt für alle Attribute. Werden z.B. in einer Methode zwei Attributen zwei unterschiedliche Objekte zugewiesen, dann ist *getRef()* bei beiden *SSAPutInstruction* eine *1*.

Ist die Überprüfung einmal wahr, wird *true* zurückgeben und damit stimmt mindestens eine Valuenumber der Instruktion mit einer Valuenumber aus dem Set überein. Sonst wird zum Schluss *false* zurückgegeben. Würde die Valuenumber *1* dann im Set vorhanden sein, würden beide Instruktionen bei der Methode *isValueUsedByInstruction true* zurückgeben, welches nicht richtig wäre. Deswegen darf *getRef()* nicht *1* sein.

```
1  private static boolean isValueUsedByInstruction(SSAInstruction ssaInstruction, Set<Integer> integers) {
2    // check, if v is defined by ssaInstruction
3    for (int i = 0; i < ssaInstruction.getNumberOfDefs(); i++) {
4      if (integers.contains(ssaInstruction.getDef(i))) {
5        return true;
6      }
7    }
8
9    // check, if ssaInstruction is a method call on v
10   if (ssaInstruction instanceof SSAAbstractInvokeInstruction) {
11     SSAAbstractInvokeInstruction invokeInstruction = (SSAAbstractInvokeInstruction) ssaInstruction;
12     if (!invokeInstruction.isSpecial()) { // ignore special methods
13       // check, if ssaInstruction is the object or the parameters from the method call
14       for (int i = 0; i < invokeInstruction.getNumberOfUses(); i++) {
15         if (integers.contains(invokeInstruction.getUse(i))) {
16           return true;
17         }
18       }
19     }
20   }
21
22   // check, if v is defined by ssaPutInstruction
23   if (ssaInstruction instanceof SSAPutInstruction) {
24     SSAPutInstruction putInstruction = (SSAPutInstruction) ssaInstruction;
25     if (putInstruction.isStatic()) {
26       return integers.contains(putInstruction.getVal());
27     } else {
28       // ignore ref == 1, ref 1 always points to an attribute
29       return (integers.contains(putInstruction.getUse(0)) && putInstruction.getRef() != 1) || integers.contains
             (putInstruction.getVal());
30     }
31   }
32
33   return false;
34 }
```

Programmcode 6.19: isValueUsedByInstruction-Methode (Quelle: Eigene Darstellung)

6.2.6.4 intraprocedural-Methode

Die *intraprocedural*-Methode arbeitet, wie der Name der Methode sagt, nur intraprozedural. Dafür werden nicht nur die Valuenumbers und die *LocalPointerKeys* der Instruktionen verwendet, sondern auch die Namen der *SSAGetInstruction* sowie *SSAPutInstruction*. Bevor der eigentlich Algorithmus erklärt wird, wird erklärt, warum die Namen der Instruktionen gebraucht werden. Wie bereits bei der *isValueUsedByInstruction*-Methode erklärt, weist eine *SSAPutInstruction* auf ein Attribut hin, wenn *getRef()* eine *1* liefert. Das Gleiche gilt auch für die *SSAGetInstruction*. Besteht eine Methode nur aus Zuweisungen und Methodenaufrufe von Attributen, dann ist *getRef()* immer *1*. Um eine Methode auf einem Attribut aufzurufen, wird durch WALA nicht nur ein *SSAAbstractInvokeInstruction* erzeugt, sondern auch eine *SSAGetInstruction* nur für die *InvokeInstruction* erzeugt. *SSAGetInstruction* liefert zwar eine Valuenumber durch *getDef()*, aber diese Valuenumber wird nur für die *InvokeInstruction* benutzt. Das Problem, welches dadurch entsteht, besteht darin, dass mit den Valuenumbers und den *LocalPointerKeys* in bestimmten Situationen kein Tracking mehr möglich ist. Insbesondere das interprozedurale Tracking kann dann nichts tracken. Diese Situation tritt dann ein, wenn auf ein Attribut zugegriffen wird und dieses Attribut nicht in der Methode oder als Attribut selbst instanziiert wird. Um das Problem deutlicher zu machen, wird dies an einem Beispiel mit den folgenden Einstellungen gezeigt:

Parameter	Wert
Java Jar Name	LocalParameter
Entrypoint Class	objectTracking/ ObjectTrackingPutMoreObjectNewInFullConstructor
Seed Statements	fooMain
CFA Options	ZERO_ONE_CFA
Data Dependence Options	FULL
Control Dependence Options	NO_EXCEPTIONAL_EDGES
Object Tracking	true

Im Programmcode 6.20 ist ein Teil des Quellcodes aus der Klasse *ObjectTrackingPutMoreObjectNewInFullConstructor* zu sehen, welcher für dieses Beispiel relevant ist. In Abbildung 6.18 ist das Slice-Ergebnis nur mit dem interprozeduralen Tracking zu erkennen. Es wurden zwar die beiden Seed-Statements *fooTwo.fooMain()* und *fooThree.fooMain()* gefunden, aber alle anderen Zeilen, welche mit den Objekten zu tun haben, fehlen komplett.

```
1  private String ne;
2
3  private Foo fooWow;
4  private Foo fooTwo;
5  private Foo fooThree;
6
7  private Foo fui;
8
9  public ObjectTrackingPutMoreObjectNewInFullConstructor(final Foo fo, final Foo foo, final Foo fooo, final Foo
       foooo){
10   fooWow = fo;
11   fooTwo = foo;
12   fooThree = fooo;
13   fui = foooo;
```

```
14 }
15
16 public void main(String[] args) {
17   fooTwo.fooNot();
18   fooTwo.init();
19   fooTwo.fooMain();
20
21   fooWow.fooNot();
22
23   fooThree.fooNot();
24   fooThree.init();
25   fooWow = fooThree;
26   fooWow.initTwo();
27   fooWow = fooTwo;
28   ne = "ne";
29   fooWow.init();
30   fooWow = fui;
31   fooWow.init();
32   fooThree.fooMain();
33 }
```

Programmcode 6.20: Teil Quellcode aus der ObjectTrackingPutMoreObjectNewInFullConstructor - Klasse (Quelle: Eigene Darstellung)

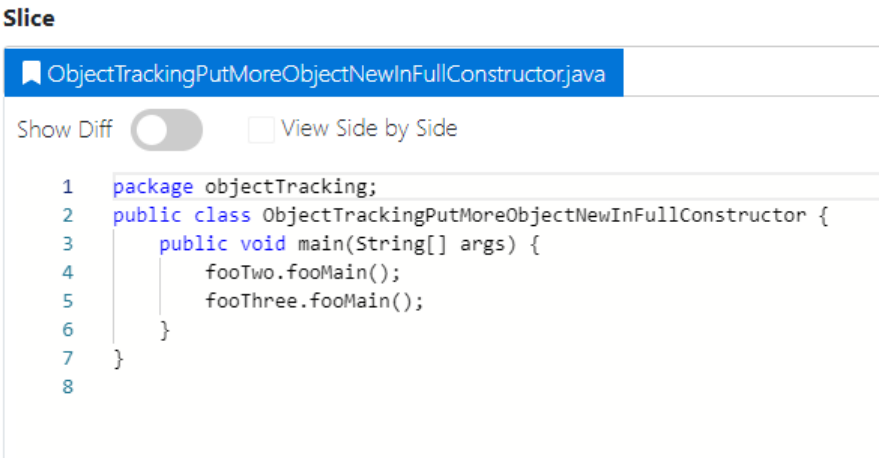


Abbildung 6.18: Slice-Ergebnis - ObjectTrackingPutMoreObjectNewInFullConstructor - interprozedural (Quelle: Eigene Darstellung)

Durch die Benutzung des intraprozeduralen Trackings werden alle Zeilen gefunden, in denen die Objekte benutzt worden sind. Der Abbildung 6.19 ist dieses Ergebnis zu entnehmen. Die roten Zeilen sind genau die Zeilen, welche nicht im Ergebnis sein dürfen. Die Attribute sind in der Abbildung nicht zu sehen, diese wurden auch nicht gefunden. Dies ist ein anderes Problem und wird mit dem *Local variable & Attribute Tracking* gelöst, welches im Abschnitt 6.2.8 fundiert beschrieben wird.

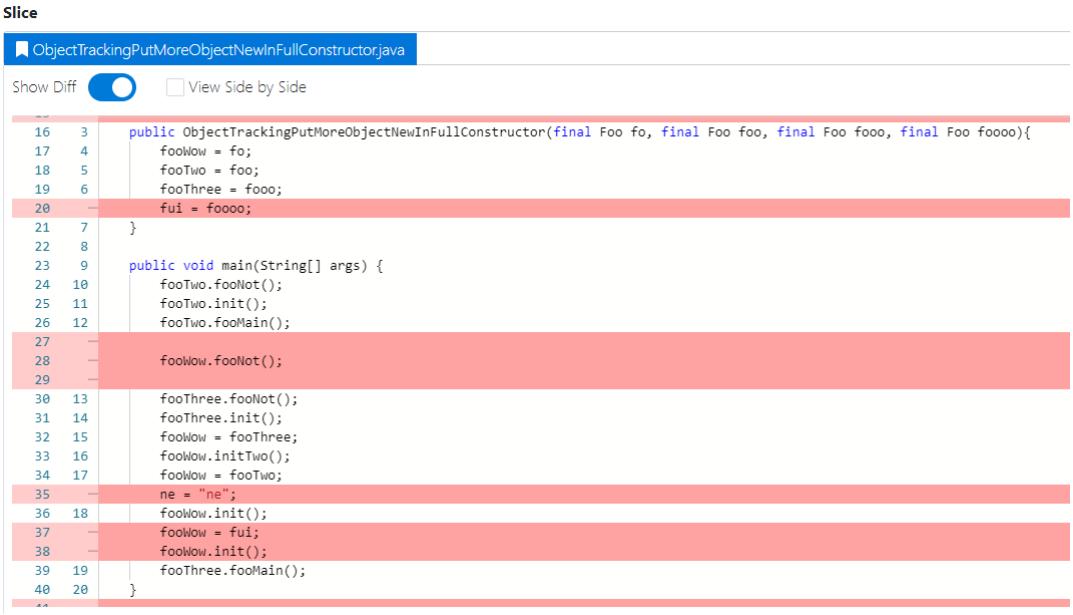


Abbildung 6.19: Slice-Ergebnis - ObjectTrackingPutMoreObjectNewInFullConstructor - intraprozedural (Quelle: Eigene Darstellung)

Der Algorithmus für die *intraprocedural*-Methode ist in fünf Schritte unterteilt. Weil der Algorithmus sehr lang ist, befindet sich dieser im Anhang A.2.

- **Schritt 1: Finden des *LokalPointerKey***
  Es wird, wie dies bei der *interprocedural*-Methode der Fall ist, für jedes übergebene *seedStatament* aus dem Set *seedStatements* die entsprechenden Valuenumber genommen und mithilfe des *HeapModel* wird der *LokalPointerKey* ermittelt. Dabei wird auf *objectTracking* und *parameterTracking* geachtet. An dieser Stelle werden nur die *SSAAbstractInvokeInstructions* betrachtet. Die gefundenen *LokalPointerKeys* werden in *localPointerKeysSeedStatement* gespeichert.

- **Schritt 2: Finden der Namen und der dazugehörigen Valuenumbers & Finden des *LocalPointerKeys* von *getRef()***
  Es wird über den Aufrufgraphen iteriert. Für jeden Knoten des Aufrufgraphen wird über seine Instruktionen iteriert. Somit wird jede Instruktion des Aufrufgraphen besucht. Ein Ziel ist es, die HashMap *nameWithValues* zu befüllen, dafür wird zwischen *SSAGetInstruction* und *SSAPutInstruction* unterschieden. Die Map besteht aus einem Knoten und der neuen Klasse *TrackingInfos*. Die Klasse dient dazu, Objekte innerhalb einer Methode zu tracken. Diese wiederum beinhaltet ein Set von der neuen Klasse *Info*. Da die Beschreibung von Info sehr komplex ausfallen würde, wird versucht, dies so einfach wie möglich zu halten. Ein *Info* Objekt spiegelt einen Namen (*Atom*) von Objekten aus einer Methode wider. Weil ein Objekt bzw. der Name in Java durch Zuweisungen im Laufe der Zeit mit unterschiedlichen Objekten belegt werden kann, speichert ein *Info* Objekt zusätzlich die Instruktionen und Va-

luenumbers zu den Namen. Diese werden später auch für den im Schritt 5 gemachten Vergleich gebraucht.
Im Fall einer *SSAPutInstruction* wird immer davon ausgegangen, dass dem Namen *xy* gerade ein neues Objekt zugewiesen worden ist. Zum Beispiel werden *xy* zwei Auto Objekte zugewiesen:

xy = new Auto(); (*getVal() = 2*)
xy = new Auto(); (*getVal() = 3*)

Dann würde es zwei *SSAPutInstruction* geben mit zwei unterschiedlichen Valuenumbers. Daher würde im *Info* Objekt, welches das *Atom xy* inne hat, jeweils ein neues Set angelegt werden. Dies passiert mit der Methode *addValueNumberWithNewSet*. Genau dieses passiert auch im 2. Schritt des Algorithmus.
Das bedeutet, dass das *xy Info* Objekt abstrakt wie folgt aussehen würde:

Set	**Instruktion**	**Valuenumber**
1	xy = new Auto();	2
2	xy = new Auto();	3

Im Fall eines *SSAGetInstruction* wird davon ausgegangen, dass sich das Objekt für den Namen *xy* nicht geändert hat, deswegen werden die Instruktion und die Valuenumber von *getDef()* mit der Methode *addValueNumberAtLastSet* hinzugefügt. Nur falls es vorher kein Set für den Namen gab, wird intern ein neues angelegt. Das oben genannte Beispiel wird durch Methodenaufrufe erweitert:

xy = new Auto(); (*getVal() = 2*)
xy.fahre(); (*getDef() = 3*)
xy = new Auto(); (*getVal() = 4*)
xy.bremse(); (*getDef() = 5*)
xy.explodiere(); (*getDef() = 6*)

Wie bereits erwähnt, wird bei einer *SSAAbstractInvokeInstruction* auf ein Attribut immer zusätzlich eine *SSAGetInstruction* generiert. Dies bedeutet, in den fünf Zeilen gibt es zwei Objekte mit fünf unterschiedlichen Valenumbers. Das *xy Info*-Objekt würde abstrakt so aussehen:

Set	**Instruktion**	**Valuenumber**
1	xy = new Auto();	2
	xy.fahre();	3
2	xy = new Auto();	4
	xy.bremse();	5
	xy.explodiere();	6

Im zweiten Schritt wird also für jede *SSAPutInstruction* und *SSAGetInstruction* die passenden *Info*-Objekte erzeugt und befüllt, welche wiederum in *TrackingInfos* für die passende Methode gespeichert werden.

Zusätzlich wird für jede *SSAPutInstruction* und *SSAGetInstruction* deren *LokalPointerKey* von der Valuenumber *getVal()* oder *getDef()* bestimmt und mit den gefundenen *LokalPointerKeys* aus dem 1. Schritt verglichen. Falls die Überprüfung erfolgreich war, wird der *LokalPointerKey* von der Valuenumber *getRef()* bestimmt und in *localPointerKeysSeedStatement* gespeichert. Diese Instruktionen dürfen dafür weder statisch sein noch auf ein Attribut zeigen (*getRef() = 1)*. Somit werden alle *LokalPointerKeys* gefunden, welche mit den Seed-Statements direkt zu tun haben.

- **Schritt 3: Nur Namen hinzufügen, bei denen die Valuenumbers gleich der Valuenumber aus dem *localPointerKeysSeedStatement* sind.**
  Die *Info*-Klasse beinhaltet nicht nur eine Liste aus Sets der gefundenen Instruktionen und Valuenumbers, sondern auch ein sortiertes Set aus Instruktionen und Valuenumbers. Dieses Set wird in diesem Schritt befüllt. Dafür werden die Valuenumbers aus den *Info*-Objekten mit den gefundenen Valuenumbers von den *LokalPointerKeys* aus *localPointerKeysSeedStatement* verglichen. Zusätzlich muss der Knoten (Methode) des *LokalPointerKeys* mit dem zugehörigen Knoten des *TrackinInfos* übereinstimmen. Wird also ein Set mit einer entsprechenden Valuenumber und Knoten gefunden, wird das gesamte Set dem sortierten Set hinzugefügt. Wäre als Beispiel *fahre* ein Seed-Statement, dann würde das sortierte Set für *xy* so aussehen:

Set	Instruktion	Valuenumber
sortiert	xy = new Auto();	2
	xy.fahre();	3

  Wäre zusätzlich *bremse()* ein Seed-Statement, dann wären alle Zeilen im sortierten Set:

Set	Instruktion	Valuenumber
sortiert	xy = new Auto();	2
	xy.fahre();	3
	xy = new Auto()	4
	xy.bremse();	5
	xy.explodiere();	6

- **Schritt 4: Namen hinzufügen, bei denen die Valuenumbers gleich den Valuenumbers aus Schritt 2 sind**
  Es wird wieder über die Knoten der Map *nameWithValues* iteriert. Für jeden Knoten wird die *TrackingInfos* aus der Map ermittelt. Über die *Infos* des *TrackingInfos* wird danach iteriert und dann wird über jedes einzelne Set des *Infos* iteriert. Für jedes Set wird nochmals über alle *Infos* und über alle einzelnen sortierten Sets iteriert.
  An dieser Stelle wird dann verglichen, ob eine Valuenumber aus einen normalen Set mit einer Valuenumber aus dem sortierten Set übereinstimmt. Ist dies der Fall, wird das normale Set zum sortierten Set hinzugefügt. Diese Überprüfung hat den Grund, dass ein Objekt in mehreren Namen gespeichert werden könnte. Denn dann würde

eine Valuenumber in zwei *Info*-Objekten vorkommen und in diesem Schritt werden diese zusammengefügt. Dafür wird das Auto-Beispiel weiter erweitert:
xy = new Auto(); (*getVal() = 2*)
xy.fahre(); (*getDef() = 3*)
xy = new Auto(); (*getVal() = 4*)
xy.bremse(); (*getDef() = 5*)
xy.explodiere(); (*getDef() = 6*)
ab = xy; (*getDef() = 7, getVal() = 7*)
ab.reparieren(); (*getDef() = 8*)
Nach dem 3. Schritt würde das *xy Info*-Objekt nur mit dem Seed-Statement *bremse* so aussehen.

Set	**Instruktion**	**Valuenumber**
1	xy = new Auto();	2
	xy.fahre();	3
2	xy = new Auto()	4
	xy.bremse();	5
	xy.explodiere();	6
	ab = xy;	7 (GET)
sortiert	xy = new Auto()	4
	xy.bremse();	5
	xy.explodiere();	6
	ab = xy;	7 (GET)

Und das *ab Info*-Objekt würde so aussehen:

Set	**Instruktion**	**Valuenumber**
1	ab = xy;	7 (PUT)
	ab.reparieren();	8

Nach dem 4. Schritt sehen die Sets wie folgt aus.
Info für *xy*:

Set	Instruktion	Valuenumber
1	xy = new Auto();	2
	xy.fahre();	3
2	xy = new Auto()	4
	xy.bremse();	5
	xy.explodiere();	6
	ab = xy;	7 (GET)
sortiert	xy = new Auto()	4
	xy.bremse();	5
	xy.explodiere();	6
	ab = xy;	7 (GET)
	ab = xy;	7 (PUT)
	ab.reparieren()	8

Info für *ab*:

Set	Instruktion	Valuenumber
1	ab = xy;	7 (PUT)
	ab.reparieren();	8

- **Schritt 5: Statements zu *sliceList* hinzufügen, wenn *LocalPointerKey* aus dem Statement gleich dem gefundenen *LocalPointerKey* ist oder die Valunumbers gleich den gefundenen Valuenumbers sind**
  An dieser Stelle wird wieder über den Aufrufgraphen und alle Instruktionen iteriert. Für jede Instruktion werden die *LokalPointerKeys* der *Uses* und *Defs* bestimmt und mit den gefundenen *LokalPointerKey* aus *localPointerKeysSeedStatement* verglichen. Daher wird über die *NumberOfUses* und *NumberOfDefs* iteriert. Falls die Überprüfung erfolgreich ist, wird aus dem Knoten und den *InstructionIndex* des Statements ein *NormalStatement* erzeugt und dieses wird in die *sliceList* gespeichert. Falls die Überprüfung nicht erfolgreich war, werden die passenden *TrackingInfos* zum aktuellen Knoten ermittelt. Es wird über die gefundenen *Atom* des *TrackingInfos* iteriert und für jedes *Atom* wird über das sortierte Set iteriert. Wenn dann eine Valuenumber des Statements mit einer Valuenumber aus einem sortierten Set übereinstimmt, wird ein *NormalStatement* erzeugt und zur *sliceList* hinzugefügt.
  Nur bei der Iteration über die *NumberOfUses* wird zusätzlich nochmals über den Aufrufgraphen iteriert. Dieses Mal wird aber nicht sofort über jede Instruktion iteriert. Wenn der Knoten (Methode) entweder eine *Init*- oder eine *CInit*-Methode ist, wird über alle *Infos* über alle sortierten Sets iteriert. Für jede Instruktion aus den einzelnen Sets wird überprüft, ob der Klassenname mit dem der Methode übereinstimmt, erst dann wenn dies der Fall ist, wird über alle Instruktionen der Methode iteriert. An dieser Stelle werden nur *SSAPutInstructions* weiter betrachtet. Wenn der *DeclaredField*-Name der *SSAPutInstruction* mit dem *DeclaredField*-Name der Instruktion, welche die Überprüfung des Klassennamens bestanden hatte, überein-

stimmt, werden dieser Knoten und der *InstructionIndex* der *SSAPutInstruction* zu einem *NormalStatement* und abgespeichert. Dieses zusätzliche Iterieren über den Aufrufgraphen für jede *NumberOfUses* wird deshalb vorgenommen, um Zuweisungen direkt am Attribut oder im Konstruktor ausfindig zu machen. Das bedeutet, dass das intraprozedurale Tracking nicht nur innerhalb der Methode trackt, sondern auch schaut, ob das Objekt bzw. der Name im Konstruktor der gleichen Klasse bereits benutzt worden ist. Deswegen wurde auch im Slice-Ergebnis in Abbildung 6.19 der Konstruktor mit drei Zuweisungen veranschaulicht.
Zum Schluss würden die gefundenen Statements geloggt, solange die Anzahl der Statements nicht *WalaSlicer.MAX_DETAILED_LOG* übersteigt und alle Statements aus *sliceList* werden dem übergebenen Set *rawSliceList* hinzugefügt.
Für das Beispiel mit den beiden Autos und dem Seed-Statement *bremse* würde dies bedeuten, dass nach dem 5. Schritt alle fünf Instruktionen aus dem sortierten Set in der *rawSliceList* gespeichert worden sind. Würde es zusätzlich einen Konstruktor geben, bei dem z.B. *xy* mit einen Objekt initialisiert werden würde, würde dieses auch in der *rawSliceList* sein.

6.2.7 Tracking to slicing criterion

Der Algorithmus für die Anforderung AF20, welcher im Programmcode 6.21 zu sehen ist, wurde in die *trackingToSliceCriterion*-Methode der *TrackingAlgorithms*-Klasse erkennen und angepasst. Für jedes Statement aus der *sliceList* wird über die *seedStatements* iteriert. Wenn der Knoten beider Statements gleich ist, wird das Statement aus der *sliceList* zu den *seedStatements* hinzugefügt.

```
1  public static void trackingToSliceCriterion(final Slice slice, final Set<Statement> seedStatements, final
       Collection<Statement> sliceList, final SliceLogger logger) {
2    if (slice.isTrackingToSlicingCriterion()) {
3      for (Statement statement : sliceList) {
4        for (Statement seedStatement : new HashSet<>(seedStatements)) {
5          if (statement.getNode().equals(seedStatement.getNode())) {
6            seedStatements.add(statement);
7            logger.log("Add statement to SedStatements: " + statement);
8            break;
9          }
10       }
11     }
12   }
13 }
```

Programmcode 6.21: trackingToSliceCriterion Algorithmus (Quelle: Eigene Darstellung)

Vorher wurde durch die Aktivierung von *trackingToSliceCriterion* jedes gefundene Statement des Objekttrackings zu den *seedStatements* hinzugefügt. Durch die Änderungen am Objekttracking würde dies zu ungewollten Ergebnissen führen. Deswegen wird das *trackingToSliceCriterion* auf die Methoden der Seed-Statements beschränkt. Um das Problem deutlicher zu machen, wird dies an einem Beispiel mit den folgenden Einstellungen gezeigt:

Parameter	Wert
Java Jar Name	LocalParameter
Method Class	objectTracking/Interprocedural
Method Names	main
Seed Statements	fooNot
CFA Options	ZERO_ONE_CFA
Data Dependence Options	FULL
Control Dependence Options	NO_EXCEPTIONAL_EDGES
Object Tracking	true
Tracking To Slicing Criterion	true

In Abbildung 6.20 ist das Slice-Ergebnis ohne die Methoden-Einschränkung zu erkennen. Die Klassen *InterproceduralComplex*, *ClassDMoreMethods* und *InterproceduralMoreObjects* sind zusätzlich im Slice enthalten, obwohl diese nichts mit der Klasse *Interprocedural* zu tun haben. Diese sind zwar im Aufrufgraphen enthalten, weil keine *Entrypoint Class* angegeben worden ist, aber wegen dem "*Tracking To Slicing Criterion*" wurden ein oder mehrere Zeilen der Klassen *ClassB*, *ClassC* und *ClassD* als Seed-Statement genommen. Nur deshalb weil diese als Seed-Statement verwendet worden sind, sind die falschen Klassen im Slice-Ergebnis enthalten.
Mit der Einschränkung, dass "*Tracking To Slicing Criterion*" nur Statements der gleichen Methode zu Seed-Statements macht, sind die genannten Klassen nicht im Slice-Ergebnis enthalten, siehe Abbildung 6.21.

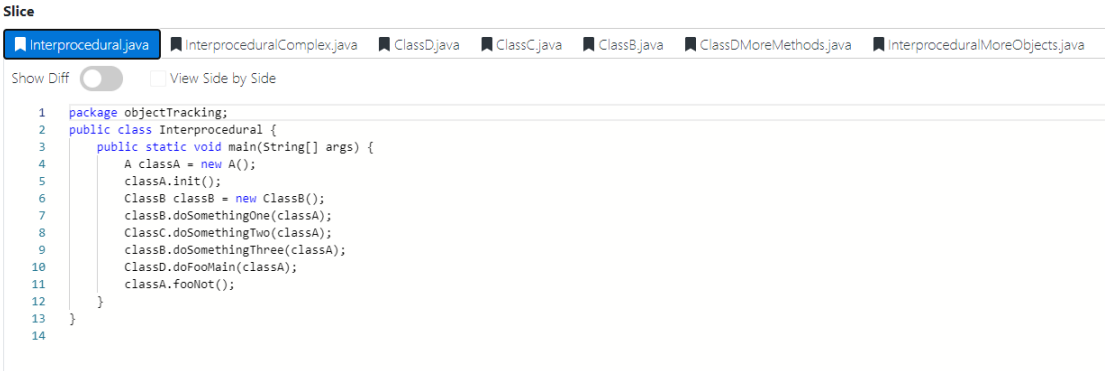


Abbildung 6.20: Slice-Ergebnis - interprocedural - ohne Methoden-Einschränkung (Quelle: Eigene Darstellung)

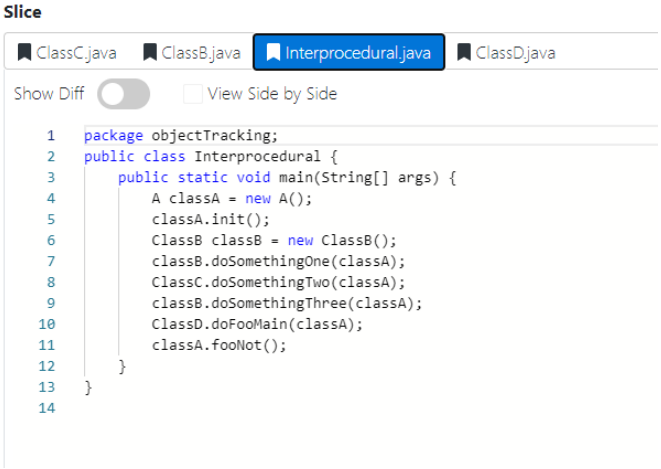


Abbildung 6.21: Slice-Ergebnis - interprocedural - mit Methoden-Einschränkung (Quelle: Eigene Darstellung)

6.2.8 Local Variable & Attribute Tracking

Das Problem, dass lokale Variablen oder Attribute nicht immer im Slice-Ergebnis erscheinen, ist länger bekannt und zeigte sich bei den meisten Vorarbeiten. Das Problem und damit die Anforderung AF21 konnten zwar durch diese Arbeit gelöst werden, aber wann genau das Problem auftritt, ist auch nach der Evaluation nicht vollkommen klar geworden. Es hat sich aber gezeigt, dass ein Attribut oder eine lokale Variable dann nicht im Slice-Ergebnis erscheint, wenn diesem direkt nichts zugewiesen wird (*z.B. private String hallo;*). Außerdem gibt es dann keine Statements für diese Attribute oder Variablen. Weil es keine Statements gibt, können durch das Rekonstruieren der Java-Dateien die Zeilen für diese nicht gefunden werden. Dazu kommt, dass einfache Zuweisungen durch WALA wegoptimiert werden, falls dies nicht schon durch den Java Compiler passiert ist. Dazu sei ein Zitat von der WALA Webseite gegeben.

> *„Because the SSA IR has already been somewhat optimized, some statements such as simple assignments (x=y, y=z) do not appear in the IR, due to copy propagation optimizations done automatically during SSA construction by the SSABuilder class. In fact, there is no SSA assignment instruction; additionally, a javac compiler is free to do these optimizations, so the statements may not even appear in the bytecode. Thus, these Java statements will never appear in the slice.“ (Zitat: [WALA, 2021b])*

In der Abbildung 6.22 ist ein Beispiel mit Objekttracking zu erkennen, in dem das Attribut *hallo* nicht im Slice-Ergebnis zu sehen ist. Das Ergebnis wurde mit den folgenden Einstellungen erzeugt:

Parameter	Wert
Java Jar Name	LocalParameter
Entrypoint Class	localParameterAttributeTracking/ attribute/AttributeNoAssignment
EntryPoint Methods	main
Seed Statements	print
CFA Options	ZERO_ONE_CFA
Data Dependence Options	FULL
Control Dependence Options	NO_EXCEPTIONAL_EDGES
Object Tracking	true

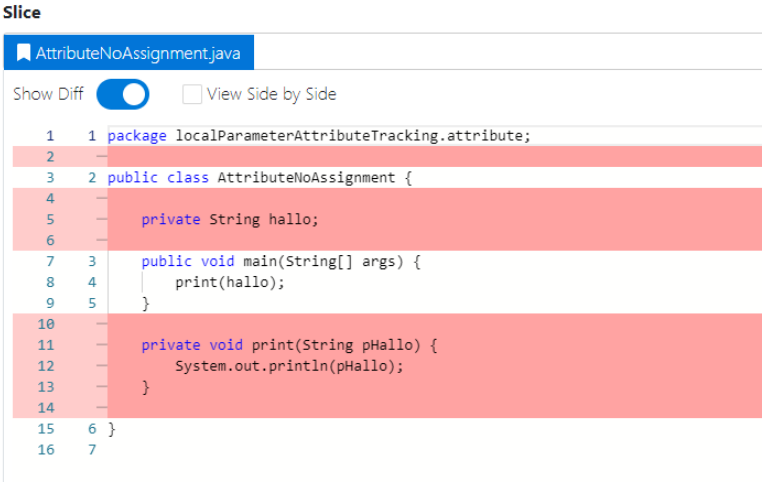


Abbildung 6.22: Slice-Ergebnis - AttributeNoAssignment (Quelle: Eigene Darstellung)

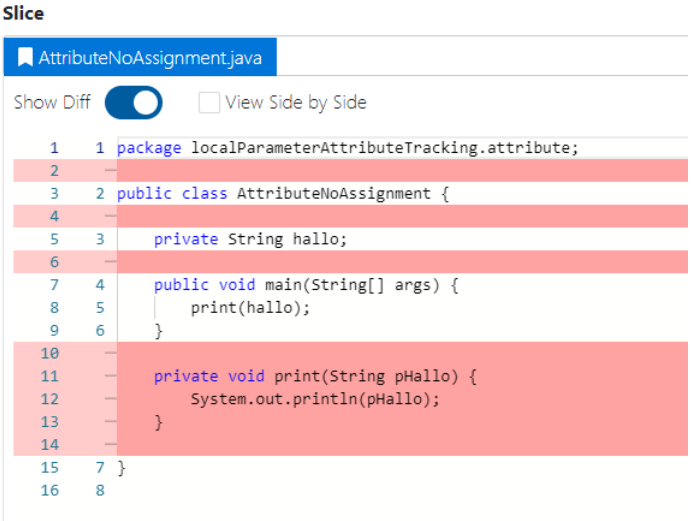


Abbildung 6.23: Slice-Ergebnis - AttributeNoAssignment - mit Attribut-Tracking (Quelle: Eigene Darstellung)

Wenn im Folgenden von Attributen gesprochen wird, sind auch immer lokale Variablen damit gemeint, außer es wird explizit anders formuliert. Das Attribut-Tracking passiert in zwei großen Schritten, zum einen durch das Finden der Namen für die Attribute und zum anderen durch das Rekonstruieren der Java-Dateien mithilfe der gefundenen Namen.
Der erste Schritt passiert mit der neuen *getParameterNamesGroupedBySourceFiles*-Methode, welche sich in der *ParameterNameAlgorithms*-Klasse befindet. Das Rekonstruieren passiert innerhalb der *MethodVisitorParameterNames*-Klasse. Das Ergebnis mit diesem Tracking zum gezeigten Beispiel ist in Abbildung 6.23 veranschaulicht.

6.2.8.1 getParameterNamesGroupedBySourceFiles

Die Methode *getParameterNamesGroupedBySourceFiles*, welche im Programmcode 6.22 zu sehen ist, bekommt unter anderem ein Set aus Statements (*sliceList*) übergeben. Die Statements in der *sliceList* sind die fertigen Statements nach dem Slicen. Die Methode versucht, alle verwendeten Namen der Attribute zu finden, und gibt diese dann als eine Map zurück. Diese Map hat als Key den Dateinamen als String und als Value ein Set aus *Pair* aus zwei Strings. Ein *Pair* spiegelt den Klassennamen und den gefundenen Namen eines Attributs wider. Weil eine Klasse auch Innere-Klassen haben kann, muss an dieser Stelle der Attributname an den Klassennamen gebunden werden, welcher an den Dateinamen gebunden wird.

```
1  public static Map<String, Set<Pair<String, String>>> getParameterNamesGroupedBySourceFiles(final Set<Statement>
       sliceList, final SliceLogger logger) {
2    Map<String, Set<Pair<String, String>>> map = new HashMap<>();
3
4    for (Statement statement : sliceList) {
5      if (statement.getKind() != null &&
6        statement instanceof StatementWithInstructionIndex &&
7        (statement.getKind() == Statement.Kind.NORMAL |
8          statement.getKind() == Statement.Kind.NORMAL_RET_CALLEE |
9          statement.getKind() == Statement.Kind.NORMAL_RET_CALLER)) {
10
11       StatementWithInstructionIndex statementIndex = (StatementWithInstructionIndex) statement;
12       CGNode cgNode = statementIndex.getNode();
13       IMethod method = cgNode.getMethod();
14       if (method instanceof FakeRootMethod || statementIndex.getInstructionIndex() == -1) {
15         continue;
16       }
17
18       SSAInstruction ssa = statementIndex.getInstruction();
19       String declaringClass;
20       String className;
21       if (ssa instanceof SSAFieldAccessInstruction) {
22         declaringClass = ((SSAFieldAccessInstruction) ssa).getDeclaredField().getDeclaringClass().getName().
             toString();
23       } else {
24         declaringClass = method.getDeclaringClass().getName().toString();
25       }
26       className = declaringClass;
27       if (declaringClass.contains("$")) {
28         declaringClass = declaringClass.substring(0, declaringClass.indexOf("$"));
29         className = className.substring(className.indexOf("$") + 1);
30       } else {
31         className = className.substring(1); //remove L
32         if (className.contains("/")) {
33           className = className.substring(className.lastIndexOf("/") + 1);
34         }
35       }
36       String declaringClassFile = declaringClass.substring(1) + ".java";
37       for (int parameterIndex = 0; parameterIndex < ssa.getNumberOfUses(); parameterIndex++) {
```

```
38          int valueNumber = ssa.getUse(parameterIndex);
39          String[] localNamesArray = cgNode.getIR().getLocalNames(statementIndex.getInstructionIndex(),
                valueNumber);
40          addLocalNamesToMap(localNamesArray, map, declaringClassFile, className, ssa);
41          for (SSAInstruction ssaInstruction : cgNode.getIR().getInstructions()) {
42            if (ssaInstruction instanceof SSAGetInstruction) {
43              SSAGetInstruction getInstruction = (SSAGetInstruction) ssaInstruction;
44              if (getInstruction.getDef() == valueNumber) {
45                String[] nameArray = new String[]{getInstruction.getDeclaredField().getName().toString()};
46                String getDeclaringClass = getInstruction.getDeclaredField().getDeclaringClass().getName().
                      toString();
47                String getClassName = getDeclaringClass;
48                if (getDeclaringClass.contains("$")) {
49                  getDeclaringClass = getDeclaringClass.substring(0, getDeclaringClass.indexOf("$"));
50                  getClassName = getClassName.substring(getClassName.indexOf("$") + 1);
51                } else {
52                  getClassName = getClassName.substring(1); //remove L
53                  if (getClassName.contains("/")) {
54                    getClassName = getClassName.substring(getClassName.lastIndexOf("/") + 1);
55                  }
56                }
57                String getDeclaringClassFile = getDeclaringClass.substring(1) + ".java";
58                addLocalNamesToMap(nameArray, map, getDeclaringClassFile, getClassName, ssa);
59              }
60            }
61          }
62        }
63
64        for (int parameterIndex = 0; parameterIndex < ssa.getNumberOfDefs(); parameterIndex++) {
65          int valueNumber = ssa.getDef(parameterIndex);
66          String[] localNamesArray = cgNode.getIR().getLocalNames(statementIndex.getInstructionIndex(),
                valueNumber);
67          addLocalNamesToMap(localNamesArray, map, declaringClassFile, className, ssa);
68        }
69      }
70    }
71    if (!map.isEmpty()) {
72      map.forEach((key, values) -> {
73        Set<String> names = new TreeSet<>();
74        Set<String> classes = new TreeSet<>();
75        values.forEach(par -> {
76          names.add(par.two);
77          classes.add(par.one);
78        });
79        logger.log("~ Found parameter/attribute names: " + names + " from classes " + classes + " in file " + key
              + ".");
80      });
81    } else {
82      logger.log("No parameter/attribute names found!");
83    }
84    return map;
85  }
```

Programmcode 6.22: *getParameterNamesGroupedBySourceFiles Methode* (Quelle: Eigene Darstellung)

Als Allererstes wird über die *sliceList* iteriert (Zeile 4). Es werden nur die Statements weiter betrachtet, welche ein *StatementWithInstructionIndex*, ein *NormalStatement* haben oder *NormalStatement* ähnlich sind und keine *FakeRootMethod* haben (Zeilen 5-16). Insbesondere darf der *InstructionIndex* nicht -1 sein. Mit diesen Statements und deren *Instruction* kann weitergearbeitet werden.
Im nächsten Schritt werden die *declaringClass* und der *className* ermittelt. Dies passiert in den Zeilen 19 bis 35. Dabei wird darauf geachtet, ob die Instruktion in einer Inneren-Klasse liegt. Ist dies der Fall, dann wird *declaringClass* den Namen der Äußeren-Klasse haben und *className* den Namen der Inneren-Klasse (Zeile 27-29). Falls die Instruktion in keiner Inneren-Klasse liegt und wenn *className* noch den Packagenamen im Namen

hat, wird dieser entfernt (Zeilen 31-34). Die *declaringClassFile* wird danach definiert und fungiert als der Dateiname. Daher wird am Ende der *declaringClass* die Endung ".java" hinzugefügt (Zeile 36).
Nachdem die Klassen- und Dateinamen ermittelt worden sind, werden die Namen der Attribute ermittelt. Dafür wird einmal über die *NumberOfUses* (Zeile 37) und einmal über die *getNumberOfDefs* (Zeile 64) des Statements iteriert und die Valuenumber bestimmt. Über den Knoten des Statements gelangt man an die *IR* und darauf kann die Methode *getLocalNames()* angewendet werden. Übergeben werden an *getLocalNames()* der *InstructionIndex* des Statements und die Valuenumber. Über die Helfermethode *addLocalNamesToMap* werden die gefundenen Namen der Map hinzugefügt. Dieser Helfermethode werden die gefundenen Namen, die Map, die *declaringClassFile* und die *className* übergeben. Diese erzeugt das *Pair* und speichert es im richtigen Set ab. Zusätzlich wird "*this*" als Name ignoriert, da dieser nie als Name vorkommen kann. WALA gibt diesen aber durch die *getLocalNames()* zurück, falls "*this*" benutzt worden ist. Zusätzlich wird, falls die Instruktion ein *SSAPutInstruction* ist, der *DeclaredField*-Name der Instruktion als Name mit abgespeichert. Dies hat folgenden Grund: Es kann vorkommen, dass *getLocalNames()* z.B. nur "*this*" liefert. Dies passiert dann, wenn auf wirkliche Attribute zugegriffen wird. Deswegen wird der Name des Attributs genommen und gespeichert.
Wenn ein *SSAAbstractInvokeInstruction* auf ein wirkliches Attribut zugreift, dann wird *getLocalNames()* kein Ergebnis liefern. Deswegen müssen die zugehörige *SSAGetInstruction* gesucht und der Name dieser Instruktion bestimmt werden. Dafür wird für jede Valuenumber über die Instruktionen des Knotens iteriert(Zeile 41-60). Wenn diese Instruktion eine *SSAGetInstruction* ist, wird die Valuenumber mit der Valuenumber der *getDef*-Methode der *SSAGetInstruction* verglichen. Wenn die Überprüfung erfolgreich war, werden der Klassenname und der Dateieinnamen vom *SSAGetInstruction* bestimmt. Der Name des Attributs wird über *getDeclaredField* ermittelt. Diese Parameter werden an die Methode *addLocalNamesToMap* weitergegeben, damit der Name des *SSAGetInstruction* hinzugefügt wird.

6.2.8.2 MethodVisitorParameterNames

Die alte *MethodVisitor*-Klasse wurde in *MethodVisitorSourceLines* umbenannt, weil es sonst Verwirrungen mit der neuen Klasse *MethodVisitorParameterNames* geben kann. Die Klasse dient wie die andere Klasse dazu, den ganzen Quellcode der ablegten Java-Dateien durchzugehen und die bereits gefundenen *slicedLineNumbers* mit neuen *sourceLineNumbers* zu füllen. Der große Unterschied liegt aber darin begründet, dass die *MethodVisitorSourceLines*-Klasse darauf spezialisiert ist, die Dateien syntaktisch richtig zu rekonstruieren. Dies geschieht aufgrund der Tatsache, dass alle Klammern und Zeilenumbrüche gefunden werden, welche durch das Slicing nicht gefunden werden können. Die *MethodVisitorParameterNames*-Klasse konzentriert sich hingegen nur auf das Finden von Attribu-

ten, lokalen Variablen und Zuweisungen. Dies wird durch die gefundenen Attribut- und Klassennamen aus *getParameterNamesGroupedBySourceFiles* realisiert, welche im Attribut *parameterNames* gespeichert werden. Diese Klasse ist erst einmal grundlegend gleich aufgebaut mit den Methoden *getSlice*, *isLineInNode*, *areSlicedLineNumbersInNode* und *addAllLinesFromBeginToEnd*. Dazu gibt es die vier *visit*-Methoden für *InitializerDeclaration*, *FieldDeclaration*, *MethodDeclaration* und *ConstructorDeclaration*, welche im Programmcode 6.23 zu sehen sind. Der ganze Quellcode der Klasse befindet sich im Anhang A.3.

```
1  @Override
2  public void visit(final InitializerDeclaration initializerDeclaration, final Object arg) {
3    for (Node node : initializerDeclaration.getBody().getChildNodes()) {
4      addStatementBodyForLocalNames(node);
5    }
6  }
7
8
9  @Override
10 public void visit(FieldDeclaration fieldDeclaration, Object arg) {
11   super.visit(fieldDeclaration, arg);
12   for (VariableDeclarator vde : fieldDeclaration.getVariables()) {
13     addStatementBodyForLocalNames(vde);
14   }
15 }
16
17
18 @Override
19 public void visit(MethodDeclaration methodDeclaration, Object arg) {
20   if (areSlicedLineNumbersInNode(methodDeclaration)) {
21     if (methodDeclaration.getBody().isPresent()) {
22       for (Node node : methodDeclaration.getBody().get().getChildNodes()) {
23         addStatementBodyForLocalNames(node);
24       }
25     }
26   }
27 }
28
29
30 @Override
31 public void visit(ConstructorDeclaration constructorDeclaration, Object arg) {
32   for (Node node : constructorDeclaration.getBody().getChildNodes()) {
33     addStatementBodyForLocalNames(node);
34   }
35 }
```

Programmcode 6.23: visit-Methoden der MethodVisitorParameterNames-Klasse (Quelle: Eigene Darstellung)

Alle vier Methoden iterieren durch ihren *Body* oder durch die Variablen und damit durch den ganzen Sourcecode. Für jeden gefundenen Knoten wird die *addStatementBodyForLocalNames*-Methode aufgerufen. Nur bei der *visit*-Methode für die *MethodDeclaration* wird darauf geachtet, das die bereits gefundenen *slicedLineNumbers* innerhalb der *MethodDeclaration* liegen. Wenn dies nicht gemacht werden würde, würden sonst lokale Variablen in anderen Methoden gefunden werden, welche zwar den gleichen Namen, wie die aus *parameterNames*, aber mit der gesuchten Variable nichts zu tun haben.
Bei den anderen dreien spielt dies deshalb keine Rolle, weil es sich um Attribute und Zuweisungen im Konstruktor handelt, welche auch gefunden werden sollen, damit zu erkennen ist, wie die Attribute initialisiert werden.
Die *addStatementBodyForLocalNames*-Methode iteriert durch die Methoden, Konstrukto-

ren usw. bis zu den einzelnen *ExpressionStmt* und *VariableDeclarator*.
Wenn die *Expression* des *ExpressionStmt* eine *VariableDeclarationExpr* ist, wird durch die einzelnen Variablen der *VariableDeclarationExpr* iteriert und die *addSourceLineNumbersForParameterName*-Methode aufgerufen. An dieser Methode werden der Name der Variablen (*getNameAsString*), der Knoten und die Anfangs- und Endzeilennummern übergeben. Die Methode wird später genauer beschrieben. Wenn die *Expression* des *ExpressionStmt* aber ein *AssignExpr*, also eine Zuweisung ist, wird der Name durch des *Target* des *AssignExpr* bestimmt.
Bei dem *VariableDeclarator* wird direkt der Name über *getNameAsString* geholt und mit den restlichen Parametern an die *addSourceLineNumbersForParameterName*-Methode übergeben.
Die Methode *addSourceLineNumbersForParameterName* überprüft, ob der übergebene Name mit einem gefundenen Namen aus *parameterNames* übereinstimmt. Zusätzlich muss noch der Klassenname des Knotens mit dem gebundenen Klassennamen des gefundenen Namens identisch sein oder die *slicedLineNumbers* liegen innerhalb der *ClassOrInterfaceDeclaration* des Knoten. Daher liegt eine Zeile aus *slicedLineNumbers* in der Klasse des Knotens. Wenn diese Überprüfung erfolgreich war, werden die übergebenen Zeilen mit der *addAllLinesFromBeginToEnd*-Methode hinzugefügt. Der Grund, warum die *slicedLineNumbers* in der *ClassOrInterfaceDeclaration* liegen müssen, ist folgender: Es kann passieren, dass der gefundene Klassenname nicht der echte Klassenname der Klasse ist. Dies passiert dann, wenn ein neues Objekt erzeugt wird, aber noch zusätzlich Methoden überschrieben werden müssen und dort ein Objekt initialisiert wird, wie z.B. die *IAlarmManager*-Klasse, in der die *set*-Methode überschrieben wird. Dort befindet sich die lokale Variable *callingUid*, siehe Abbildung 6.24. Dann ist der Klassenname weder *IAlarmManager*, *IAlarmManager* ähnlich, noch die Klasse *AlarmManagerService*, sondern hat den Namen *2*. An dieser Stelle soll betont werden, dass das Beispiel von Android aus der *AlarmManagerService*-Klasse stammt. Ich kenne mich mit Android nicht aus, aber bei meiner Recherche habe ich keine Klasse mit dem Namen *2* gefunden. Daher gehe ich davon aus, dass entweder WALA den Namen irgendwie erzeugt hat oder *2* eine Innere-Klasse ist.
Daher reicht an dieser Stelle eine Überprüfung auf den Klassennamen nicht aus und es wird geschaut, ob die *slicedLineNumbers* in der *ClassOrInterfaceDeclaration* liegen.

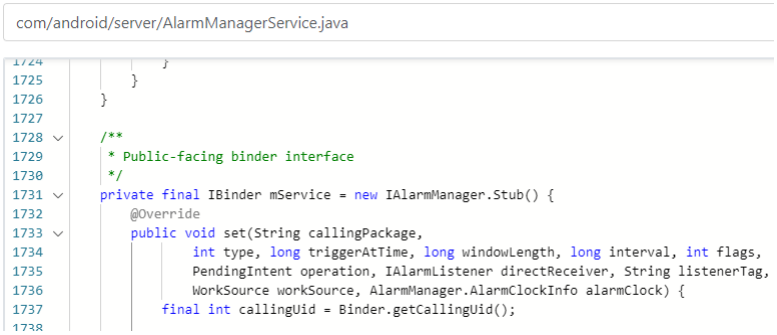


Abbildung 6.24: Beispiel - IAlarmManager (Quelle: Eigene Darstellung)

6.3 Slices-Tab & View a Slice

Die Ansichten zwischen den Slicemodes Android und Java im Slices-Tab und beim Ansehen eines Slices wurden auch vereinheitlicht und genauso benannt wie auf der "*Make a new Slice*"-Seite. So wurde in dieser Ansicht "*Entry Methods*" für Java "*Seed Statement Class*" genannt. Damit gab es drei unterschiedliche Bezeichnungen für "*Entry Methods*". Diese wurden, wie im Abschnitt 6.2 beschrieben, alle verworfen. Das neue "*Seed Statement Classes*" hat daher nicht mit der alten "*Seed Statement Class*" zu tun.
Im Slices-Tab, siehe Abbildung 6.25, werden die einzelnen, in Auftrag gegebenen Slices mit den Einstellungsmöglichkeiten aufgelistet:

- Java Jar Name oder Android Version
- Entrypoint Class
- Entrypoint Methode
- Method Class
- Method Names
- Seed Statements
- CFA Option
- Data Dependence Options
- Controll Dependence Options
- Running

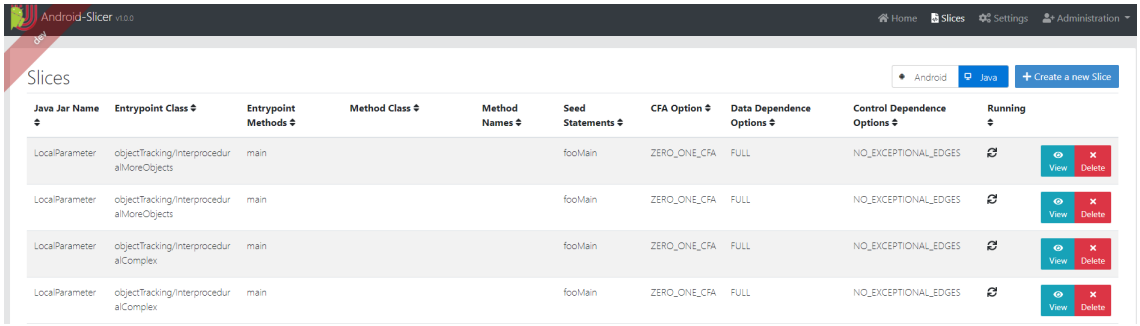


Abbildung 6.25: Slices-Tab - Marcel (Quelle: Eigene Darstellung)

Beim Ansehen eines Slices werden alle Einstellungsmöglichkeiten angezeigt. Wenn eine Einstellung nicht betätigt worden ist, wird dies mit einem "-" markiert. Nur in der Java Ansicht wird die Einstellung "*Only Public Methods*" angezeigt. Ein Beispiel für ein vollständiges Slice-Ergebnis ist den drei Abbildungen 6.26, 6.27 und 6.28 zu entnehmen (Anforderung AF33).

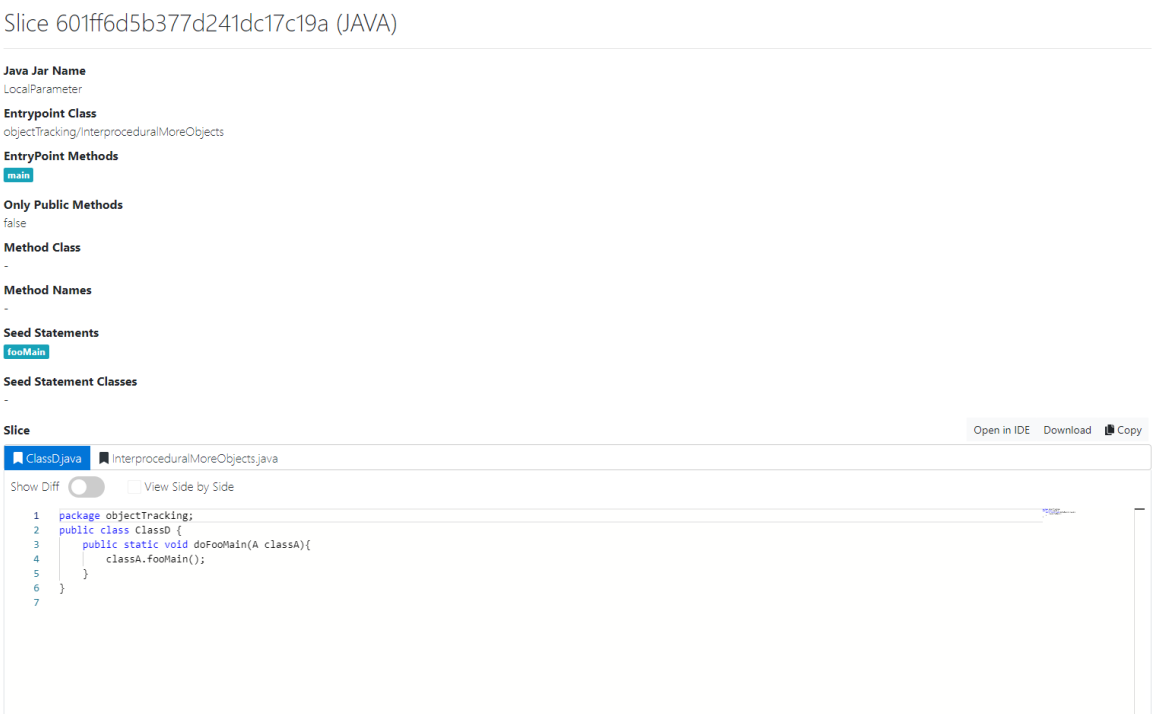


Abbildung 6.26: Ansicht eines fertigen Slices - Teil 1 - Marcel (Quelle: Eigene Darstellung)

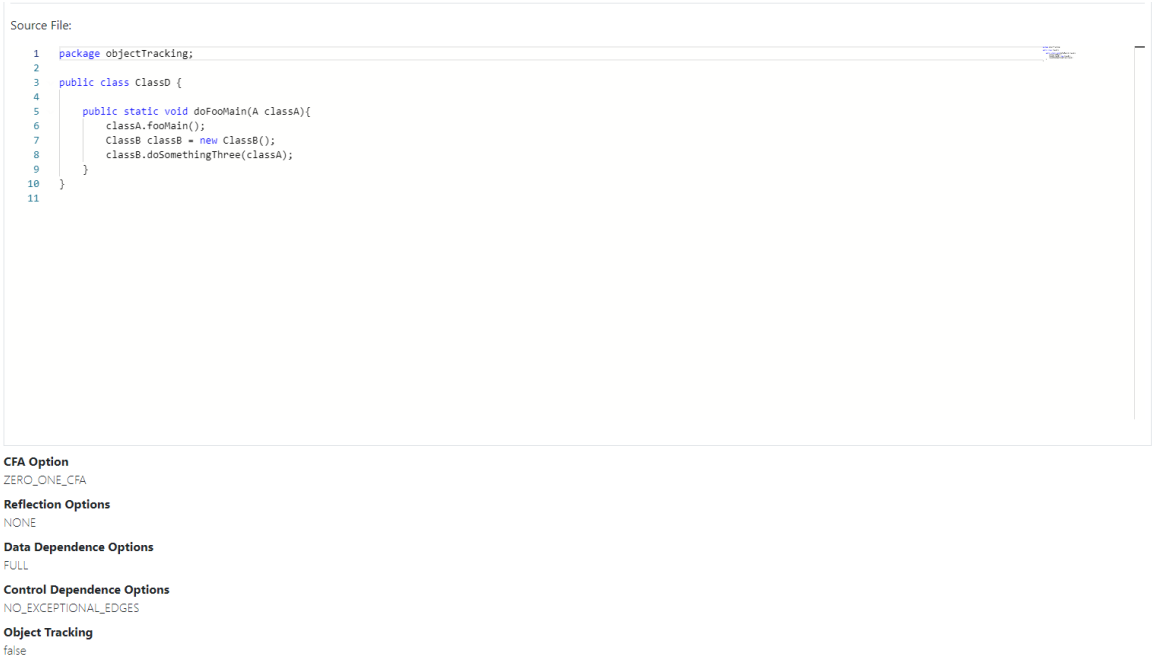


Abbildung 6.27: Ansicht eines fertigen Slices - Teil 2 - Marcel (Quelle: Eigene Darstellung)

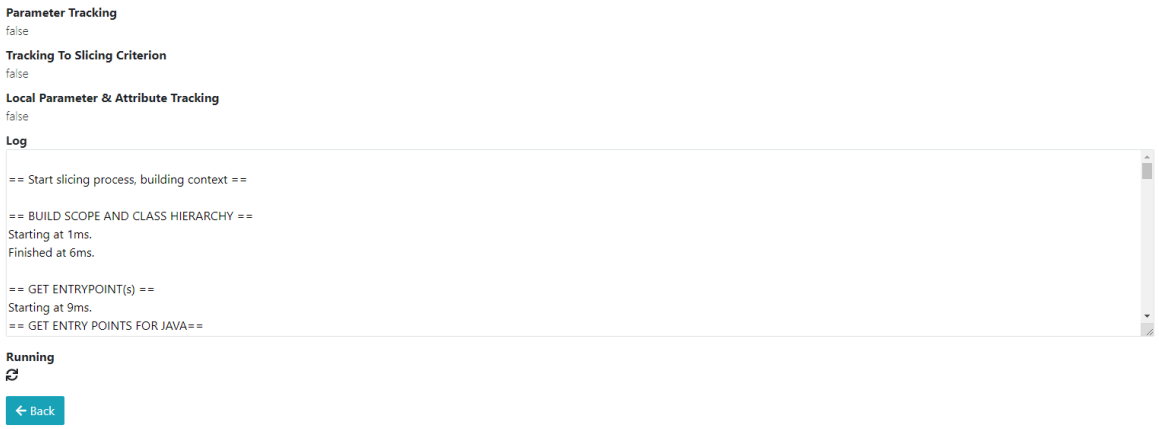


Abbildung 6.28: Ansicht eines fertigen Slices - Teil 3 - Marcel (Quelle: Eigene Darstellung)

Zusätzlich ist in der Abbildung 6.26 auf der rechten Seite über dem Slice der neue *Copy*-Button zu sehen. Bis jetzt war es nicht möglich, einen neuen Slice mit den gleichen Einstellungen wie einen anderen Slice anzupassen und neu zu starten. Dies ist über den neuen *Copy*-Button möglich (Anforderung AF38). Wenn auf diesen Button geklickt wird, wird man auf die "*Make a new Slice*"-Seite weitergeleitet und alle Einstellungen von dem angesehenen Slice werden übernommen. Der Button wird auch dann angezeigt, wenn es noch kein Slice-Ergebnis gibt. Somit ist es möglich, sehr schnell mehrere Slices mit gleichen oder angepassten Einstellungen in Auftrag zu geben. Gerade durch dieses kleine Feature konnten beim manuellen Testen viele Arbeitsstunden eingespart werden.

6.4 Settings-Tab

Im Setting-Tab, siehe Abbildung 6.29, befindet sich der neue Key *Java_Directory_Path*. Als Standardwert ist *java-jars* gesetzt. Über diesen Key kann der Ordner festgelegt werden, in welchem die hochgeladen Zip-Datei für das Erstellen eines neuen Slices entpackt wird.

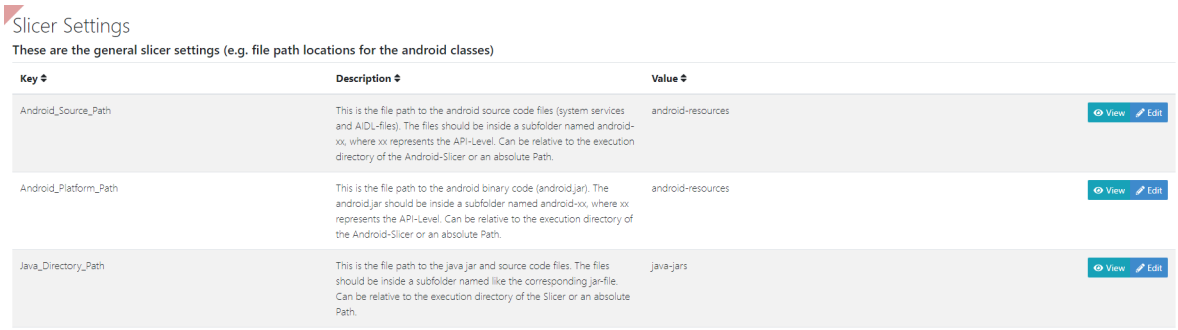


Abbildung 6.29: Settings-Tab - Java_Directory_Path (Quelle: Eigene Darstellung)

Zusätzlich wurden das Ansehen und das Bearbeiten der *Exlusion_List* überarbeitet, um die Benutzbarkeit zu verbessern. Vorher war alles in einen einzelnen Text zu sehen und zu bearbeiten. Weil es aber dort sehr schwierig war, einzelne Einträge zu löschen oder neue hinzuzufügen, wurde dieses Vorgehen überarbeitet. In der Abbildung 6.30 ist der erste Teil und in Abbildung 6.31 der zweite Teil der Ansicht zu sehen.
Über das Textfeld "*Add Exclusion*" können einzelne Exklusionen hinzugefügt werden. Das Hinzufügen geschieht über den *Add*-Button. Darunter befindet sich eine Auflistung aller Exklusionen mit jeweils einem *Delete*-Button. Über diesen können die einzelnen Einträge gelöscht werden. Erst durch das Speichern über den *Save*-Button werden alle Änderungen übernommen. Falls also ein Fehler beim Löschen gemacht worden ist, reicht es aus, auf den *Cancel*-Button zu klicken.

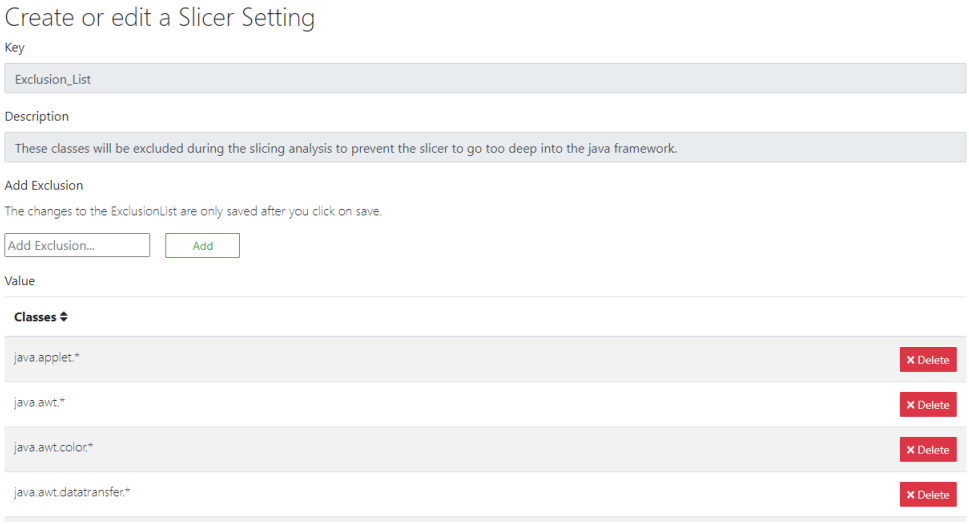


Abbildung 6.30: Exclusion_List - Bearbeiten - Teil 1 (Quelle: Eigene Darstellung)

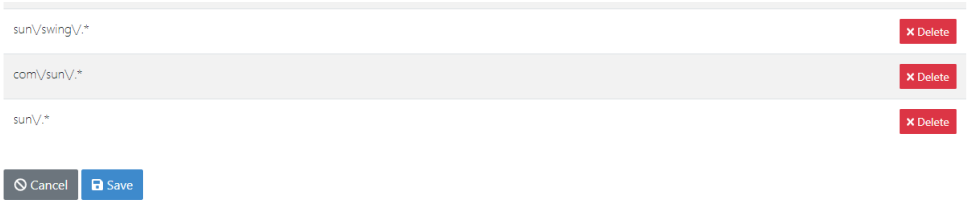


Abbildung 6.31: Exclusion_List - Bearbeiten - Teil 2 (Quelle: Eigene Darstellung)

6.5 Sonstige Änderungen

6.5.1 Slicer Options

Nachdem die *Slicer Options* Seite so korrigiert worden ist, dass diese wieder aufrufbar wurde, hat sich auch gezeigt, dass es nicht mehr möglich war, zwischen Seite 1 und Seite 2 der Optionen zu wechseln. Deswegen wurde weitestgehend der Stand von Cyl wieder hergestellt. An der Benutzeroberfläche und an den einzelnen Einstellungen hat sich dadurch nichts geändert.

6.5.2 Java Parser

Kerimov, Cyl und Möhlmann sind bei ihren Arbeiten nach der Evaluation noch Fehler beim Rekonstruieren der Java-Dateien aufgefallen. Diese waren syntaktisch nicht korrekt oder wichtige Sachen fehlten. Dazu zählt das Fehlen von geschweiften-Klammern bei Catch-Blöcken und import-Anweisungen. Beim eigenen Testen sind noch zusätzlich

folgende Fehler identifiziert worden, welche für die Anforderung AF32 sowie N3 behoben werden mussten:

- Es werden else-Blöcke angegeben, und zwar selbst dann, wenn dort keine gefundene Zeile ist. Außerdem erscheinen Zeilen im Ergebnis, welche dort nicht sein dürften.
- *finally* fehlt bei Try-Catch-Blöcken.
- Statische initial-Blöcke werden komplett ignoriert.
- Zeilenumbrüche bei der Klassendeklaration werden nicht erkannt.

Im Folgenden werden die einzelne Probleme an einem Beispiel verdeutlicht und dann jeweils das Ergebnis nach der Veränderung präsentiert. Das Rekonstruieren passiert in der *MethodVisitorSourceLines*-Klasse, welche vorher *MethodVisitor* hieß. Der ganze Quellcode befindet sich im Anhang A.4. Alle folgenden Beispiele wurden mit den folgenden Einstellungen generiert:

Parameter	Wert
Java Jar Name	LocalParameter
Entrypoint Class	convention/NameDerKlasse
Seed Statements	print
CFA Options	ZERO_ONE_CFA
Data Dependence Options	FULL
Control Dependence Options	NO_EXCEPTIONAL_EDGES

TryCatchConvention

In der Abbildung 6.32 ist ein Beispiel für das Rekonstruieren mit dem alten Algorithmus zu erkennen. Es ist zu erkennen, dass die geschweiften Klammern beim *try*-Block sowie das *finally* im *finally*-Block fehlten. Diese fehlten nach der Änderung nicht mehr, siehe Abbildung 6.33. Dafür werden jede Anfangs- und jede Endzeile jedes *TryStmt*, *TryBlock*, *CatchClause*, *CatchBody* und *FinallyBlock* ins Ergebnis aufgenommen. Diese sind für die Syntax unerlässlich. Für jeden *CatchClause* werden deren Kinderknoten nur dann fokussiert, wenn *isLineInNode true* zurückgibt. Das Finden des Schlüsselworts *finally* ist deshalb schwieriger, weil dieser keine Zeilennummer hat. Daher muss die letzte Zeile vor dem *finally* ermittelt werden. Dieser ist entweder die letzte Zeile des *TryBlock* oder die letzte Zeile des letzten *CatchClause*. Dann werden alle Zeilen von der ermittelten Zeile bis zum Anfang des *FinallyBlock* ins Ergebnis aufgenommen.

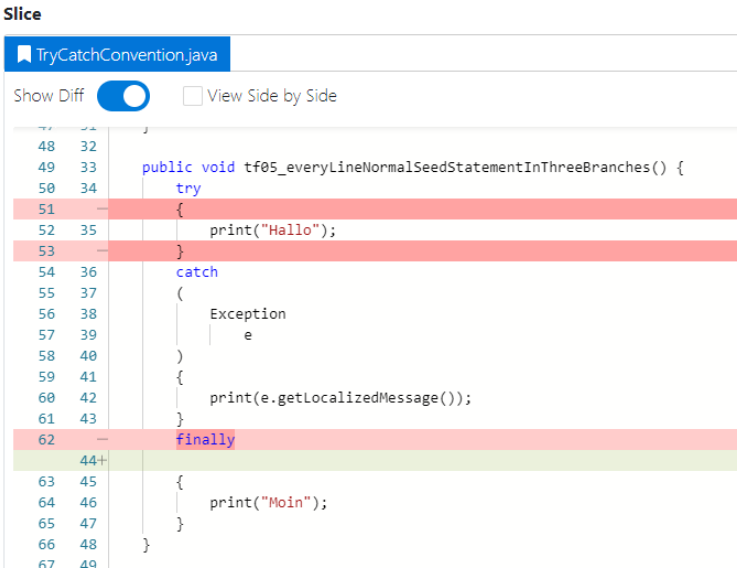


Abbildung 6.32: Slice-Ergebnis - Try-Catch-Blöcke - alt (Quelle: Eigene Darstellung)

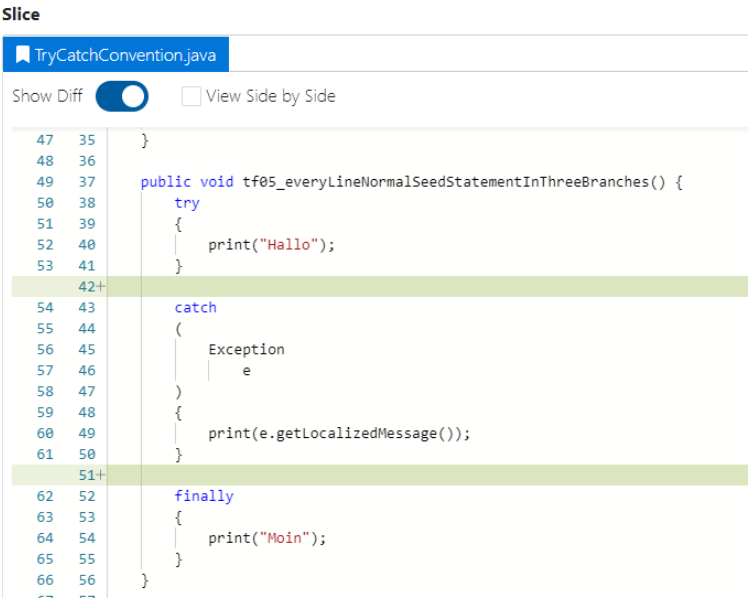


Abbildung 6.33: Slice-Ergebnis - Try-Catch-Blöcke - alt (Quelle: Eigene Darstellung)

Import- & Package- & ClassConvention

Die Methode *setClassBody* wurde komplett überarbeitet. Diese ist für das Rekonstruieren der *ClassOrInterfaceDeclaration* zuständig. Die Import-Anweisungen wurden vorher komplett ignoriert und das Hinzufügen der Packages befand sich vorher in der *SliceMapper*-Klasse. Beides wird nun durch die *setClassBody*-Methode realisiert. Um die Zeilen für diese zu erhalten, muss der Vaterknoten des *ClassOrInterfaceDeclaration*-Knoten eine *CompilationUnit* sein, denn dann kann über die Imports iteriert, die Packages gefunden und deren Zeilen hinzugefügt werden.

Schwieriger ist es bei der Klassendeklaration an sich. Weil dort nicht alle Schlüsselwörter, wie *extends*, eine angegebene Zeile haben, wird über die einzelnen *Token* iteriert, bis das erste "{" Symbol gefunden wird. Ein Token ist ein Symbol im Quellcode. Wenn dieses Symbol gefunden worden ist, wird davon ausgegangen, dass dies die letzte Zeile der Klassendeklaration ist und der Body mit Attributen usw. beginnt. Deswegen werden alle Zeilen vom Anfang der Klassendeklaration bis zur ermittelten Zeile des Symbols ins Ergebnis aufgenommen. Falls der Vaterknoten auch ein *ClassOrInterfaceDeclaration*-Knoten sein sollte, dann wird die Methode *setClassBody* mit den neuen Knoten aufgerufen. Dies kommt dann vor, wenn der betrachtete Knoten eine Innere-Klasse ist.
In der Abbildung 6.34 ist zu erkennen, dass zum einen die Imports richtig erkannt werden, auch wenn diese in mehreren Zeilen umgebrochen worden sind, und zum anderen, dass das Package vorliegt.

Abbildung 6.34: Slice-Ergebnis - Import-Anweisungen - neu (Quelle: Eigene Darstellung)

In den Abbildungen 6.35 und 6.36 ist das Ergebnis für eine Klassendeklaration einmal mit dem alten Algorithmus und einmal mit dem neuen Algorithmus zu erkennen. Jeder Zeilenumbruch wird nun richtig erkannt.

Abbildung 6.35: Slice-Ergebnis - Klassendeklaration - alt (Quelle: Eigene Darstellung)

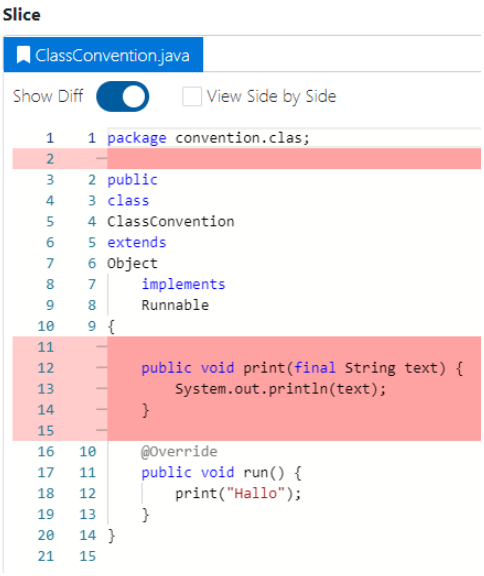


Abbildung 6.36: Slice-Ergebnis - Klassendeklaration - neu (Quelle: Eigene Darstellung)

ifConvention

Die If-Else-Blöcke wurden so angepasst, dass zum einen der Else-Block nicht mehr im Slice-Ergebnis ist, wenn in diesem keine Zeile enthalten ist, welche durch das Slicen gefunden worden ist. Weil ein leerer Else-Block keinen Mehrwert bietet und dieser syntaktisch auch weggelassen werden kann, wird dieser auch weggelassen.

In Abbildung 6.37 ist zu erkennen, dass nicht nur der leere Else-Block im Ergebnis ist, sondern auch Zeilen 122 und 130, welche beide mit dem *println*-Methode, aber nicht dem Seed-Statement *print* zu tun haben. Dies lag darin begründet, dass bei der alten

Berechnung der Zeilen ein Rechenfehler vorlag. In der Abbildung 6.38 ist zu erkennen, dass nach den Änderungen der Else-Block und die beiden Zeilen 122, 130 nicht mehr im Ergebnis vorhanden sind.

Abbildung 6.37: Slice-Ergebnis - If-Else-Block - alt (Quelle: Eigene Darstellung)

Abbildung 6.38: Slice-Ergebnis - If-Else-Block - neu (Quelle: Eigene Darstellung)

staticConvention

In Java gibt es die "statischen initial Blöcke", welche vermutlich nicht häufig benutzt werden. Dies wurde auch nur zufällig beim Testen an Android bemerkt. Der Abbildung

6.39 ist ein Beispiel mit zwei statischen initial Blöcken zu entnehmen. Es wurden nur die beiden *print*-Aufrufe und sonst nichts anderes erkannt. Dies zeigt, dass beim Durchlaufen des Quellcodes diese statischen Blöcke nicht berücksichtigt werden. Dafür wurde die *visit*-Methode mit dem *InitializerDeclaration* hinzugefügt. Da diese vom Prinzip her wie die anderen *visit*-Methoden funktioniert, wird darauf nicht weiter eingegangen.
In Abbildung 6.40 werden die statischen Blöcke mit der Hilfe der *visit*-Methode erkannt.

Abbildung 6.39: Slice-Ergebnis - StaticBlock - alt (Quelle: Eigene Darstellung)

Abbildung 6.40: Slice-Ergebnis - StaticBlock - neu (Quelle: Eigene Darstellung)

6.6 Zusammenfassung des Kapitels

In diesem Kapitel wurden die Veränderungen an der Benutzeroberfläche sowie die dafür verantwortlichen Algorithmen veranschaulicht. Dafür wurde zuerst die "Problematik der Ungleichheit" (Abschnitt 6.2.1) herausgearbeitet und wie verwirrend diese sein kann. Daher wurden die einzelnen Seiten und die Einstellungsmöglichkeiten für einen Slice vereinheitlicht. Mithilfe der unterschiedlichen Tracking-Algorithmen ist es nun möglich, noch präziser zu tracken, insbesondere wurden Attribute, lokale Variablen und Zuweisungen gefunden, welche vorher im Slice-Ergebnis fehlten. Als Letztes wurden aktuelle Fehler beim Rekonstruieren der Java-Dateien gezeigt und es wurde erläutert, wie diese behoben worden sind.

Kapitel 7

Evaluation

Unterteilt ist dieses Kapitel in drei Abschnitte. Im Abschnitt 7.1 werden die alten manuellen Beispieltests der Vorarbeiten in JUnit-Tests umgewandelt. Im zweiten Abschnitt 7.2 werden zusätzliche JUnit-Tests vorgestellt, welche die Veränderungen durch die Implementierung abdecken sollen. Außerdem wird eine Reihe von Tests für *Cipher* und *Signature* vorgestellt (Abschnitt 7.3), diese zeigen, ob und wie *Cipher*, *Signature* usw. „gesliced“ werden können. Zum Schluss wird in Abschnitt 7.4 der Slicer an der öffentlichen Bibliothek „OSCI“ evaluiert.

7.1 Eingliederung alter Beispiele

Es hatte sich recht schnell vor der Implementierung gezeigt, dass Funktionalitäten fehlten oder der Slicer nicht das gewünschte Ergebnis erzielt hatte. Bei der Evaluation ist dann die Tatsache aufgefallen, dass es keine (JUnit)-Tests gibt, welche z.B. verhindern könnten, dass der Slicer mit bestimmten Einstellungen ein falsches Ergebnis liefert. Durch die JUnit-Tests ist es möglich, dass Teile eines Programms auf Richtigkeit getestet werden. Es kann zwar nie das ganze Programm abgedeckt werden, aber dies hat den Vorteil, dass Veränderungen im Programmcode schnell durch Tests erkannt werden können, wenn diese dann fehlschlagen. Ein zusätzlicher Vorteil besteht darin, dass die Tests in großer Anzahl immer und immer wieder ausgeführt werden können und somit Zeit sparen im Vergleich zu manuellen Tests.
Daher wurden die alten Beispiele von Cyl [Cyl, 2019], Nguyen [Nguyen, 2018] und Kerimov [Kerimov, 2019] als JUnit-Tests gebaut. Dies bedeutet, die verwendeten Dateien und Einstellungen für den Slicer wurden dafür übernommen. Ein paar wenige Beispiele wurden aber nicht umgesetzt, denn diese Tests würden in keiner sinnvollen Zeit terminieren und würden daher auch keinen Vorteil generieren. Als Beispiel lief der Test mit der *Google Tink* Bibliothek von Kerimov über 60 Minuten.
In den folgenden Abschnitten werden nicht alle Tests und deren Ergebnisse ausführlich erklärt. Dafür sind es zu viele. Auf Grund dieser Tatsache werden nur exemplarisch einige ausgewählt und das Verhalten des Slicers an ihnen veranschaulicht.

Der Aufbau eines Tests ist in Abbildung 7.1 visualisiert. Der Methodenname ist grob folgendermaßen aufgebaut:
"tfNR_testName_methodName_seedStatement_dataDependence_controllDependence"

```
@Test
public void tf01_arithmeticTest_main_println_full_noExceptionalEdges() throws Exception {
    CompletableFuture<Slice> completableFuture = sliceService.process(createSlice( entryPointClass: "ArithmeticTest",  entryPointMethod: null,  methodClass: null,  methodNames: "main",
        seedStatement: "println",  seedStatementClass: null, CFAType.ZERO_ONE_CFA,  cfaLevel: 0,
        Slicer.DataDependenceOptions.FULL, Slicer.ControlDependenceOptions.NO_EXCEPTIONAL_EDGES,  objectTracking: false,  trackingToSlicingCriterion: false,  javaJarName: "ArithmeticTest"));
    checkSlicedClasses(completableFuture.get().getSlicedClasses(),  javaClass: "ArithmeticTest.java", Arrays.asList(1, 3, 6, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23));
}
```

Abbildung 7.1: Beispiel einer Testmethode zum "Testfall TF1" von Cyl (Quelle: Eigene Darstellung)

Der Aufbau kann sich aber von Testklasse zu Testklasse leicht unterscheiden, wenn andere Parameter für dem Slicer genommen worden sind. Bei der Nummer für den Testfall wurde darauf geachtet, dass dieser zu der Nummer in der Arbeit der Vorarbeiten passt, falls welche verwendet worden sind.
Ein Test läuft im Folgenden so ab:

- Es wird ein Slice mit den bestimmten Parameter erstellt.
- Über die *SliceService*-Klasse wird der Slice ausgeführt.
- Es wird eine Überprüfung der Zeilennummern für die einzelnen analysierten Klassen durchgeführt. Stimmt eine Zeilennummer nicht überein, gilt der Test als fehlgeschlagen.

Alle Tests für alle drei Vorarbeiten sind in den Klassen *ExamplesCylIT*, *ExamplesNguyenIT* und *ExamplesKerimovIT* zu finden, welche sich im Package *org.unibremen.mcyl.androidslicer.wala.VorarbeiterNachname* befinden. In Abbildung 7.2 ist die Package-Struktur für die Tests zu erkennen.

Abbildung 7.2: Packageaufbau der Testklassen (Quelle: Eigene Darstellung)

Zusätzlich wurden alle Tests von den drei Vorarbeiten mit aktivierter lokaler Variablen und Attribut Tracking durchgeführt. Diese befinden sich in den Klassen *ExamplesCylLocalVariableIT*, *ExamplesNguyenLocalVariableIT* und *ExamplesKerimovLocalVariableIT*.

7.1.1 Arbeit von Nguyen

Bei der Evaluation der Beispiele von Nguyen hatte sich erst richtig gezeigt, welches Ausmaß das Fehlen der Funktionalitäten erreichte. Einige Tests konnten zuerst deshalb nicht umgesetzt werden, weil z.B.: *seedStatementClasses* fehlte oder es nicht möglich war, *null* als *entryPointClass* anzugeben. Auch die Idee vom Unterteilen in *entryPointClass*, *entryPointMethods*, *methodClass* und *methodNames* stammte aus dieser Evaluation.
Bei der Evaluation der Beispiele von Nguyen ist sonst nichts Besonderes aufgefallen. Die Veränderungen am Slicer haben die Ergebnisse nicht negativ beeinflusst.
Im Folgenden wird ein Beispiel mit den folgenden Einstellungen präsentiert:

Testfall: SimpleTest-tf05 mit Attribut-Tracking

Parameter	Wert
Java Jar Name	SimpleTest
Entrypoint Class	SimpleTest
Method Name	entry_method
Seed Statements	important_method
CFA Options	ZERO_ONE_CFA
Data Dependence Options	FULL
Control Dependence Options	NO_EXCEPTIONAL_EDGES
Object Tracking	true
Tracking to Slicing Criterion	true
Local Variable & Attribute Tracking	true

In der Abbildung 7.3 ist das alte Slice-Ergebnis aus der Arbeit von Nguyen konkretisiert. In der Abbildung 7.4 ist das neue Slice-Ergebnis mit den gezeigten Einstellungen veranschaulicht. Erst einmal ist kein Unterschied zwischen dem alten und neuen Slice-Ergebnis für die Klasse *SimpleTestObjekt* festzustellen. Daher wird diese Klasse nicht weiter betrachtet. Bei der *SimpleTest*-Klasse sind hingegen zwei Dinge aufgefallen.

Zum einen ist die Zeile 16 "*value++*" im Slice-Ergebnis vorhanden. Selbst dann, wenn kein Tracking aktiviert ist, ist diese Zeile im Ergebnis vorhanden. Daher konnte an dieser Stelle nicht geklärt werden, warum diese Zeile jetzt im Ergebnis erkannt wird, aber früher nicht.
Und zum anderen ist zu erkennen, dass die lokale Variable "*Integer value = 4*" selbst mit eingeschaltetem Attribut-Tracking nicht gefunden wird. Die Vermutung liegt hier nahe, dass beim Compilieren oder durch WALA dieses komplett wegoptimiert wurde, denn beim Debuggen ist Folgendes aufgefallen:
Der Aufruf der WALA-Methode *getLocalNames* auf der IR gibt nichts zurück. Als SSA-Instruktion wird die *SSABinaryOpInstruction* verwendet. Diese Instruktion beinhaltet nur die Informationen über die Valuenumbers sowie die verwendete Operation, wie z.B. "*add*" oder "*div*". Das bedeutet, dass über diese Instruktion keine neuen Informationen gewonnen werden können. Das größere Problem besteht darin, dass "*Integer value = 4*" als Konstante definiert worden ist. Ob dies durch WALA oder durch einen Compiler passiert ist, konnte nicht eindeutig geklärt werden. Es liegt aber nahe, dass dies durch WALA geschehen ist.
Durch die Definition als Konstante, erhält der *int*-Wert *4* eine Valuenumber. Die Valuenumbers von Konstanten sind daher die einzigen Valuenumbers, die in jeder Methode innerhalb der Klasse verwendet werden können und sich nicht ändern. Aber10 der *int*-Wert *4* ist nur mit der Valuenumber verknüpft und hat keinen Bezug mehr zum ursprünglichen Attribut.
An dieser Stelle hat sich gezeigt, dass das "Lokale Variablen und Attribut Tracking" auch seine Grenzen besitzt.

```
1  public class SimpleTestObject {
2    public static Integer calc_param(){
3      return 5;
4    }
5  }
6
7  public class SimpleTest {
8    public static String entry_method(Integer anothervalue) {
9      SimpleTestObject obj = new SimpleTestObject();
10     obj.do_stuff();
11     Integer addvalue = SimpleTestObject.calc_param();
12     result = obj.important_method(addvalue, anothervalue);
13   }
14   public static void main(String[] argsv){
15     String result = entry_method(value);
16   }
17 }
```

Abbildung 7.3: Slice-Ergebnis - SimpleTest-tf05 - alt [Nguyen, 2018, Seite 103]

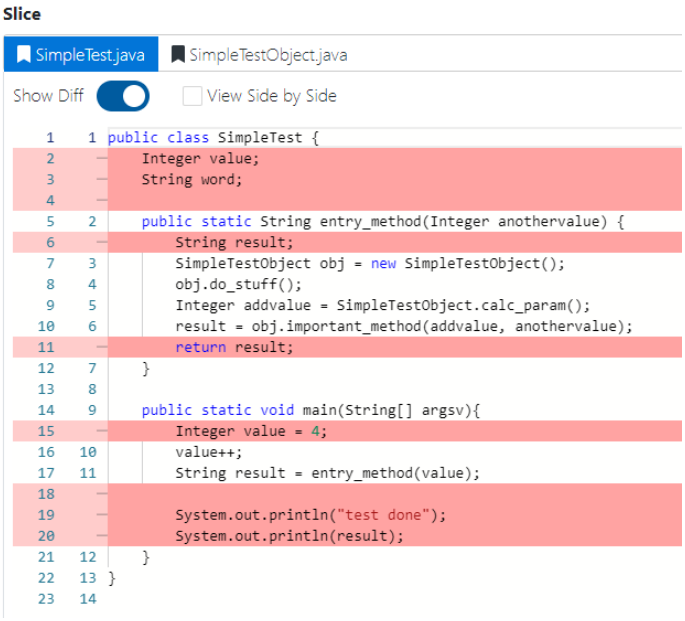


Abbildung 7.4: Slice-Ergebnis - SimpleTest-tf05 - neu (Quelle: Eigene Darstellung)

7.1.2 Arbeit von Cyl

Im Vergleich zu Nguyen und Kerimov hat Cyl sich auf Android konzentriert. Seine Beispiele in JUnit-Tests umzubauen, stellte deshalb überhaupt kein Problem dar, weil auf der Android-Seite des Slicers fast nichts geändert worden ist. Die Android Versionen für die Tests liegen im *android-resources* Ordner im Projekt-Ordner. Für die Tests wurde deshalb aber nur die Android-Version 28 verwendet, weil kein Unterschied zwischen den Versionen 28 und 29 beim Testen aufgefallen ist.
Auch bei den normalen Tests von Cyl hat sich nichts Besonderes gezeigt, daher sind keine Abweichungen zu seinen Test-Ergebnissen festzustellen.
Beim Aktivieren des Attribut-Trackings hingegen kam es vor, das Konstruktoren gefunden worden sind. Dies wird am folgenden Slice-Ergebnis, siehe Abbildung 7.5, mit den folgenden Einstellungen verdeutlicht:

Testfall: PackageManagerService-tf10 mit Attribut-Tracking

Parameter	Wert
Java Jar Name	SimpleTest
Entrypoint Class	com/android/server/pm/PackageManagerService
Entrypoint Methods	reboot
Method Class	com/android/server/pm/PackageManagerService
Seed Statements	enforceCallingOrSelfPermission
CFA Options	ZERO_ONE_CFA
Data Dependence Options	NO_BASE_NO_HEAP_NO_EXCEPTIONS
Control Dependence Options	NO_EXCEPTIONAL_EDGES
Object Tracking	true
Tracking to Slicing Criterion	true
Local Variable & Attribute Tracking	true

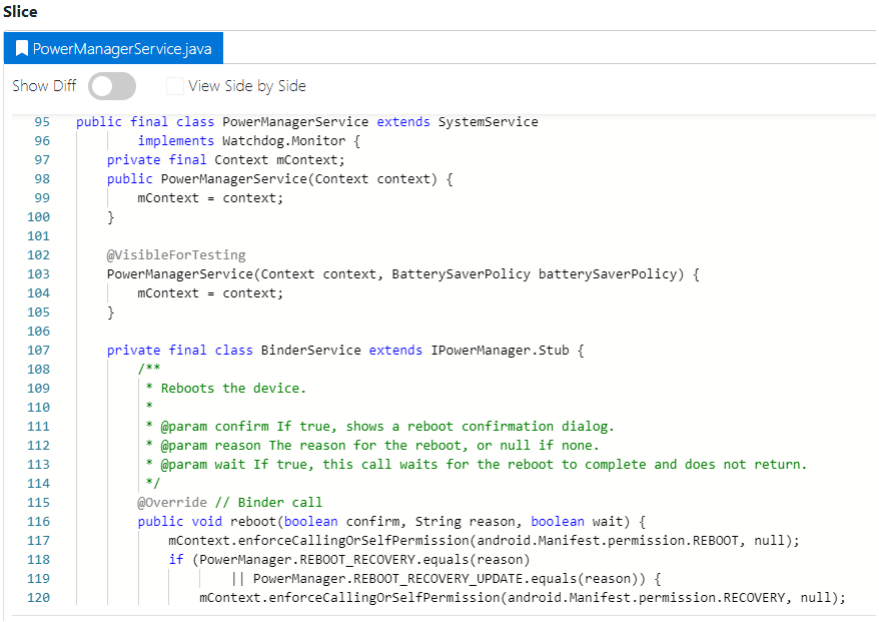


Abbildung 7.5: Slice-Ergebnis - PackageManagerService-tf10 - neu (Quelle: Eigene Darstellung)

Es ist zu erkennen, dass zwei Konstruktoren sowie das Attribut *mContext* erfolgreich gefunden worden sind. In den Konstruktoren ist auch die Zuweisung von *mContext* zu erkennen. Andere Attribute und Variablen wurden in diesem Beispiel nicht gefunden, weil auch keine weiteren benutzt worden sind.
An allen Beispielen von Cyl wurde auch das Objekt-Tracking getestet, weil es zu der Zeit von Cyl noch kein Objekt-Tracking gegeben hat. Das Tracking hat aber keinen Mehrwert generiert, weil immer das Objekt *context* verfolgt wird und dieses wurde schon immer richtig erkannt.

7.1.3 Arbeit von Kerimov

Kerimov testete zum einen den Slicer an selbstgebauten Beispielen mithilfe der *Java Cryptography Architecture* (JCA). Dafür verwendete er die digitalen Signaturen und die "*Message Authentication Codes*". Beide Arten werden für bestimmte Verschlüsselungstechniken benutzt. Und zum anderen testete er den Slicer an den beiden Frameworks *Apache Shiro* und *Google Tink*. *Apache Shiro* ist ein Java-basiertes Security-Framework und hilft den Entwicklern bei der Implementierung von "Sicherheitsmechanismen". *Google Tink* ist auch ein Framework für kryptographische Anwendungen [Kerimov, 2019].
Alle Beispiele außer denen von *Google Tink* wurden als JUnit-Tests gebaut. Der Test von *Google Tink* lief über 60 Minuten und wurde daher nicht mit übernommen. Die Dateien für die Tests befinden sich im Ordner *examples_kerimov* innerhalb des *test* Ordners.
Im Folgenden wird auf ein Testbeispiel mit diesen Einstellungen eingegangen:

Testfall: Shiro-tf03 mit Attribut-Tracking

Parameter	Wert
Java Jar Name	testapp_with_shiro
Method Names	crypt
Seed Statements	doFinal
CFA Options	ZERO_ONE_CFA
Data Dependence Options	NO_HEAP
Control Dependence Options	FULL
Local Variable & Attribute Tracking	true

Für diesen Test wird nur die *JcaCipherService*-Klasse fokussiert und von dieser nur bestimmte Abschnitte. In der Abbildung 7.6 sind die Attribute und der Konstruktor der Klasse zu sehen. Diese wurden auch richtig erkannt und keiner davon wurde False-Positiv erkannt. Aber wie bei dem Beispiel von Nguyen wurden dieses Mal keine Attribute gefunden. In der Abbildung 7.7 ist die Methode *generateInitializationVector* abgebildet. In dieser wird das Attribut *BITS_PER_BYTE* verwendet. Bei diesem Attribut handelt es sich auch um eine Konstante, weshalb kein Name gefunden werden kann.

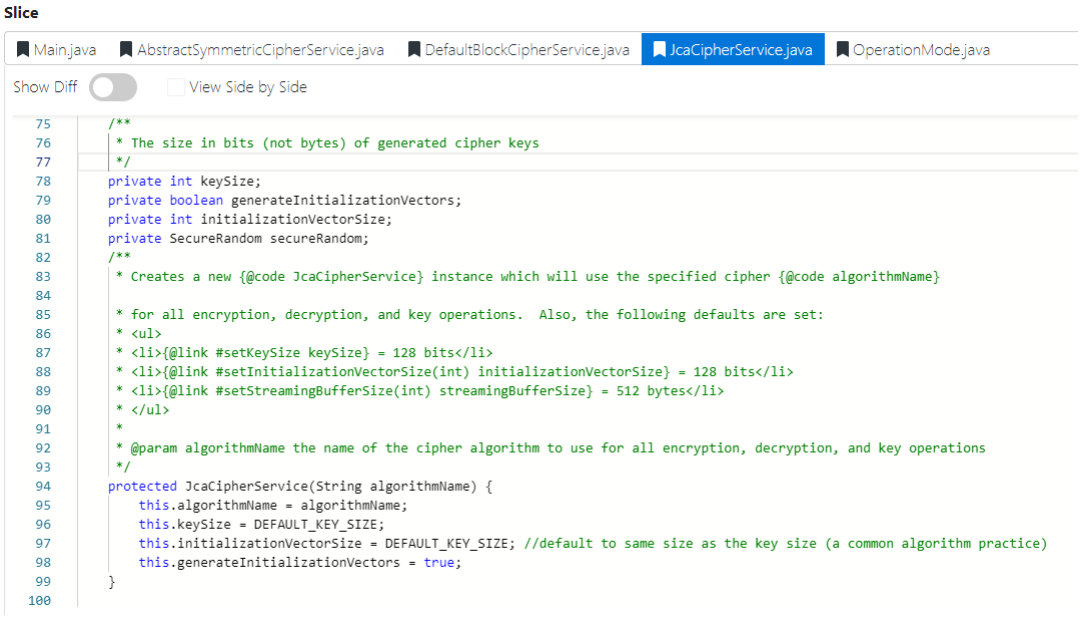


Abbildung 7.6: Slice-Ergebnis - JcaCipherService - neu - Teil 1.1 (Quelle: Eigene Darstellung)

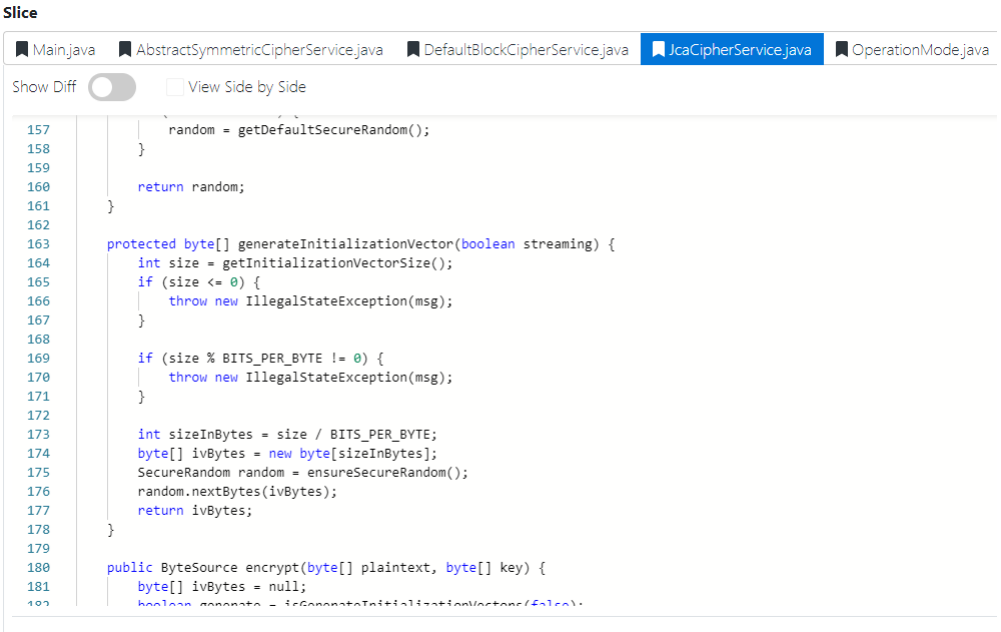


Abbildung 7.7: Slice-Ergebnis - JcaCipherService - neu - Teil 1.2 (Quelle: Eigene Darstellung)

Der Abbildung 7.8 ist die *encrypt*-Methode zu entnehmen. Abgesehen davon, dass die lokale Variable *ivBytes* richtig erkannt worden ist, sind die Zeile 186 und 187 im Slice-Ergebnis enthalten. Diese waren bei dem Test von Kerimov nicht vorhanden. Dies liegt daran, dass Cyl das Rekonstruieren der Java-Dateien verbessert hatte. Unter anderem werden die *ThrowStmts* richtig erkannt und mit ins Slice-Ergebnis übernommen. Dadurch ist das Slice-Ergebnis besser zu lesen, weil, wie z.B. in diesen Beispiel, die *IF*-Blöcke dann nicht mehr leer sind.

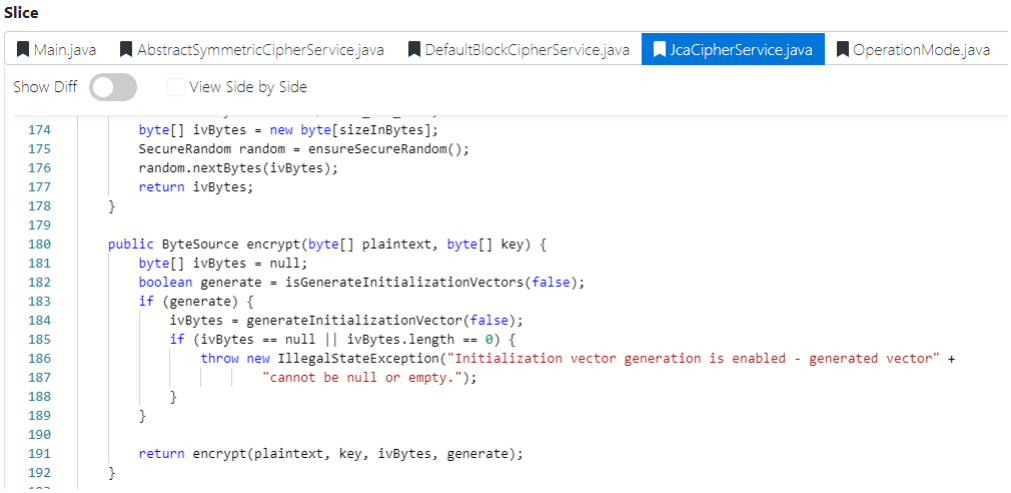


Abbildung 7.8: Slice-Ergebnis - JcaCipherService - neu - Teil 2 (Quelle: Eigene Darstellung)

Der größte Unterschied zwischen diesem Slice-Ergebnis und dem Slice-Ergebnis von Kerimov besteht darin, dass die Methode *init* im Slice-Ergebnis vorhanden ist. Die *init*-Methode ist in Abbildung 7.9 zu erkennen. Diese wird durch die *initNewCipher*-Methode aufgerufen. Bevor die *crypt*-Methode mit dem Seed-Statement aufgerufen wird, wird die *initNewCipher*-Methode verwendet. An dieser Stelle hat sich gezeigt, dass das interprozedurale-Tracking, das gewünschte Ergebnis geliefert hat. Denn das *Cipher*-Objekt wird in der *init*-Methode initialisiert, daher muss diese auch im Slice-Ergebnis mit aktiviertem Objekt-Tracking vorhanden sein.

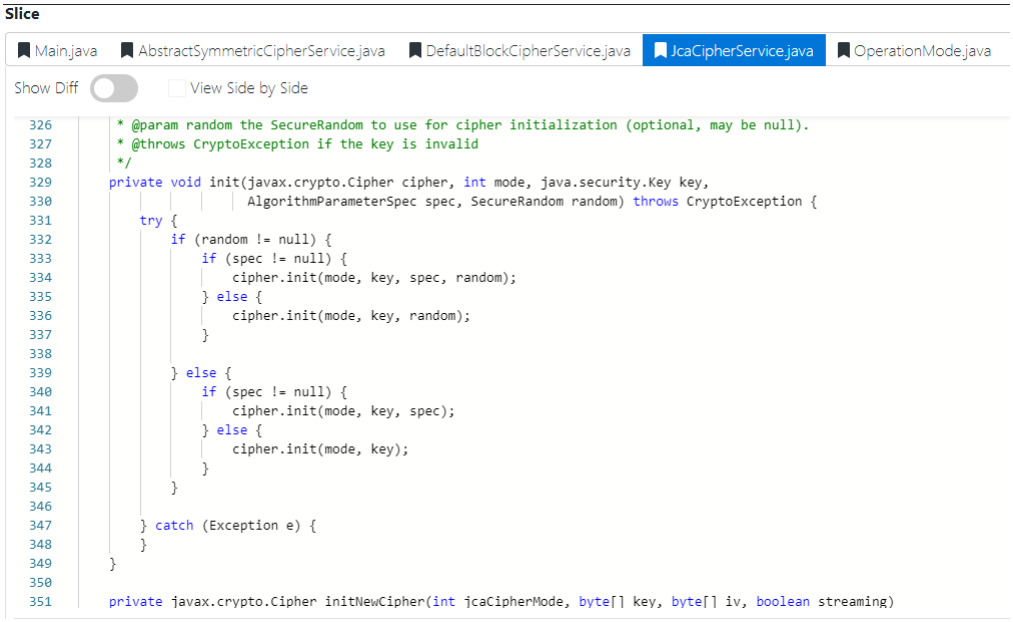


Abbildung 7.9: Slice-Ergebnis - JcaCipherService - neu - Teil 3 (Quelle: Eigene Darstellung)

7.2 Zusätzliche JUnit-Tests

In diesem Abschnitt wird auf die einzelnen neuen Test-Klassen eingegangen, welche nicht durch das Übernehmen der Beispielen der Vorarbeiten entstanden sind. An dieser Stelle wurden nur solche Tests erstellt, welche auch die Veränderungen am Slicer so gut wie möglich abdecken sollen. Dazu zählen Tests für die Code-Konvention, die Ermittlung der Einstiegspunkte und die Ermittlung der "Methoden für das Seed-Statement". Dazu kommen Tests für das Objekt-, Parameter- und Attribut-Tracking sowie Tests für das "Tracking to Slice Criterion". Für die Tests wird ein Beispiel gezeigt, falls dies nicht bereits in einem anderen Abschnitt passiert ist. Ausgenommen sind die Tests für das "Tracking to Slice Criterion". Der Algorithmus ist so einfach, dass die Tests dafür in der Arbeit nicht extra erklärt werden müssen.

7.2.1 Code-Konvention Tests

Insgesamt gibt es sieben Test-Klassen innerhalb des *convention*-Ordners. Diese sind: *ClasConventionIT*, *CommentConventionIT*, *IfConventionIT*, *ImportsConventionIT*, *PakacgeConventionIT*, *StaticConventionIT* und *TryCatchConventionIT*. Die größten und relevantesten Tests wurden bereits im Kapitel 6 "Implementierung" im Abschnitt 6.5.2 "Java Parser" präsentiert, um zum einen die vorherigen Fehler und zum anderen die Richtigkeit der Rekonstruktion der Java-Dateien zu zeigen. Bei diesen Tests sowie beim manuellen Testen sind keine weiteren Fehler bei der Code-Konvention festgestellt worden. Weil die

Tests bereits im "Java Parser" Abschnitt gezeigt worden sind, wird an dieser Stelle darauf nicht weiter eingegangen.

7.2.2 Einstiegspunkte-Tests

Der Test der Einstiegspunkte befindet sich in der *EntryPointIT*-Testklasse. In dieser Testklasse wird getestet, ob und wie weit sich die Suche nach den Einstiegspunkten eingeschränkt bzw. erweitert werden kann. Durch die beiden Einstellungsmöglichkeiten "*Entrypoint Class*" und "*Entrypoint Methods*" lassen sich erst einmal vier mögliche Kombinationen erstellen:

- Beide sind *null*.
- Keiner ist *null*.
- Nur *Entrypoint Class* ist *null*.
- Nur *Entrypoint Methods* ist *null*.

Das Testen der beiden Fälle "Beide sind *null*" und "Nur *Entrypoint Class* ist *null*" konnte bei dem Slicemode *Java* nicht umgesetzt werden, denn in den beiden Fällen würden alle *main*-Methoden aller Klassen von der Jar-Datei als Ergebnis geliefert werden. Weil aber diese Anzahl durch das Hinzufügen oder Löschen von Klassen schwanken kann, wurde auf diese Tests verzichtet. Eine Ausnahme bildet der Test, in dem *Entrypoint Class* zwar *null* ist, aber bei der *Entrypoint Methods* eine Methode angegeben worden ist, welche nicht *main* ist. Denn dann wird deshalb kein Einstiegspunkt gefunden, weil eine *WalaException* geworfen wird. Getestet wird allgemein das Ergebnis der *getEntrypoints*-Methode aus der *EntryPointAlgorithms*-Klasse. In der *EntryPointIT*-Testklasse befindet sich die *checkEntryPoints*-Methode, welche das Ergebnis aus der *getEntrypoints*-Methode überprüft. Im Folgenden wird ein Beispiel mit diesen Einstellungen gezeigt.

Testfall: entryPointAlgorithms_java_oneEntryPointMethods

Parameter	Wert
Java Jar Name	LocalParameter
Entrypoint Class	entryPointMethods/EntryPoint
EntryPoint Methods	navigateToPrint
Method Names	doSomething
Seed Statements	navigateToPrint
CFA Options	ZERO_ONE_CFA
Data Dependence Options	FULL
Control Dependence Options	NO_EXCEPTIONAL_EDGES

An dieser Stelle ist die Angabe der *Method Names* und des *Seed-Statements* deshalb irrelevant, weil sich nur auf das Ergebnis der Ermittlung der Einstiegspunkte konzentriert

wird. Auch wird hier nicht der eigene JUnit-Test gezeigt, weil das Ergebnis über die GUI des Slicers anschaulicher ist. In Abbildung 7.10 ist das Log zu diesem Test zu sehen. Es ist zu erkennen, dass der Algorithmus von "*(Gulmann/Gerken) (Detmers)*" verwendet worden ist. Außerdem wurde ein Einstiegspunkt gefunden, und zwar genau die angegebene Methode *navigateToPrint* in der Klasse *entryPointMethods/EntryPoint*. Die ganze Klasse ist im Programmcode 7.1 abgebildet. Der Algorithmus hat daher richtig gearbeitet. Abgesehen von der Einschränkung, dass nicht alle Tests umgesetzt werden konnten, wurde nur bestätigt, dass der Algorithmus wie erwünscht funktioniert. Bei allen Tests konnte kein abweichendes Verhalten festgestellt werden.

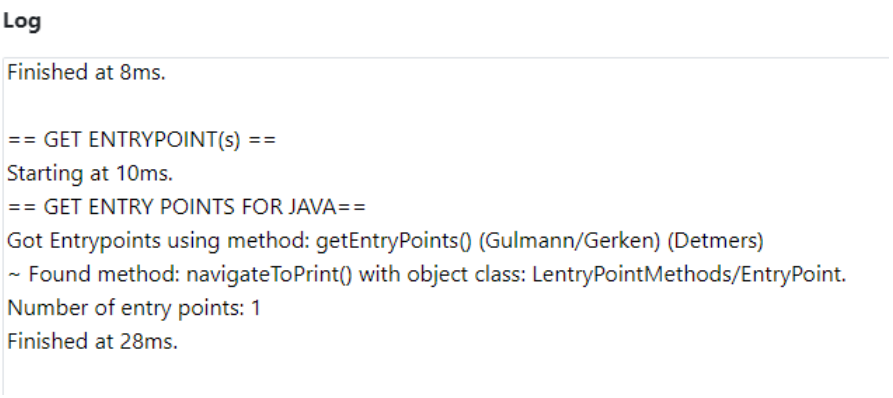


Abbildung 7.10: Log vom Slice-Ergebnis zum Einstiegspunkt-Test (Quelle: Eigene Darstellung)

```
1  package entryPointMethods;
2
3  public class EntryPoint {
4
5      public static void main(String[] args) {
6          String hallo = "hallo";
7          print(hallo);
8      }
9
10     public void bla(){
11         print("Hi");
12     }
13
14     public void fuu(){
15         RandomClass randomClass = new RandomClass();
16         randomClass.doSomething();
17     }
18
19     /**
20      * Bla bla
21      * @param pHallo hello
22      * @throws RuntimeException throw things
23      */
24     private static void print(String pHallo) {
25         System.out.println(pHallo);
26     }
27
28     public static void navigateToPrint(String pNani){
29         print(pNani);
30     }
31
32 }
```

Programmcode 7.1: EntryPoint - Klasse (Quelle: Eigene Darstellung)

7.2.3 "Methoden für das Seed-Statement"-Tests

Der Test für die "Methoden für das Seed-Statement" befindet sich in der *MethodNodesIT*-Testklasse. Analog zu den Einstiegspunkt-Tests können durch die Kombination aus "*Method Class*" und "*Method Names*" vier Kombinationen erstellt werden. Zusätzlich wurde auch getestet, dass die Einstiegspunkte und die "Methoden für das Seed-Statement" in unterschiedlichen Klassen und in unterschiedlichen Methoden liegen können. Dieses war vor der Implementierung nicht möglich. Genau dieses Beispiel wird an den folgenden Einstellungen verdeutlicht:

Testfall: methodNodeAlgorithms_java_differentClass

Parameter	Wert
Java Jar Name	LocalParameter
Entrypoint Class	methodNodes/EntryClass
EntryPoint Methods	entry
Method Class	methodNodes/MethodClass
Method Names	entryMethod
Seed Statements	print
CFA Options	ZERO_ONE_CFA
Data Dependence Options	FULL
Control Dependence Options	NO_EXCEPTIONAL_EDGES

Auch an dieser Stelle ist das Seed-Statement irrelevant, denn nur das Ergebnis der *findMethodNodes*-Methode aus der *MethodNodesAlgorithms*-Klasse wird getestet. Dieses passiert mit der Hilfe der *checkMethodNodes*-Methode, welche sich in der *MethodNodesIT*-Testklasse befindet. Zur besseren Darstellung wird auch hier das Log gezeigt, welches sich in Abbildung 7.11 befindet.
Es ist zu erkennen, dass die Einstiegspunkt-Klasse "LmethodNodes/EntryClass" und die *Method Class* "LmethodNodes/MethodClass" richtig erkannt wurden. Somit sind die beiden Klassen und die ausgewählten Methoden *entry* bzw. *entryMethod* unterschiedlich. Dieser Test hat damit nachgewiesen, dass beide Algorithmen unabhängig voneinander arbeiten, weil die Einstellungsmöglichkeiten für den Slicer aufgeteilt worden sind. Zusätzlich haben die Tests belegt, dass die *findMethodNodes*-Methode genau so funktioniert, wie es in der Implementation beschrieben worden ist. Bei allen Tests konnte kein abweichendes Verhalten konstatiert werden.

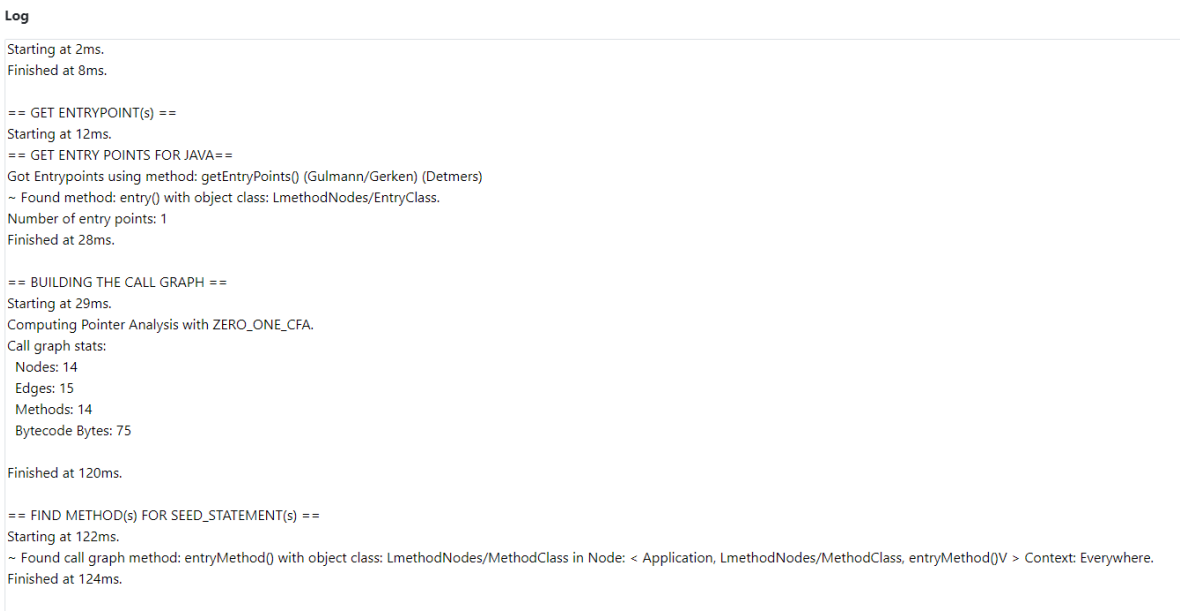


Abbildung 7.11: Log vom Slice-Ergebnis zum MethodNodes-Test (Quelle: Eigene Darstellung)

7.2.4 Tracking Tests

Zu den Testklassen zum Objekt-, Parameter- und Attribut-Tracking gibt es jeweils noch eine *NoHeap* Testklasse. *NoHeap* soll dann als Datenabhängigkeitsoption ausgewählt werden, wenn das Slicen sehr lange dauert. Das Slicing wird dadurch um einiges beschleunigt, aber die Datenabhängigkeitskanten, die sich auf dem Heap zeigen, werden dabei ignoriert. Deswegen wurden die Testklassen einmal mit der Datenabhängigkeitsoption *FULL* und einmal mit *NO_HEAP* ausgeführt, um zu schauen, wie weit die Tracking-Algorithmen den Verlust dieser Datenabhängigkeitskanten kompensieren.
Bei allen Tracking-Tests wurde darauf geachtet, dass so viel wie möglich unterschiedliche Aspekte getestet werden. An dieser Stelle können natürlich nicht alle Aspekte aufgezählt werden, aber einige werden bei den Tracking-Tests in den nächsten Abschnitten benannt oder gezeigt. Außerdem ist es unmöglich, alle Aspekte zu testen bzw. zu wissen, auf was geachtet werden muss, dafür sind Programmiersprachen, wie Java, zu komplex.

7.2.4.1 Objekt-Tracking

Die Tests für das Objekt-Tracking befinden sich in der *ObjektTrackingIT*-Klasse. Insgesamt gibt es 46 Tests. Durch das interprozedurale- und intraprozedurale-Tracking wurde besonders darauf geachtet, dass bei dem Erstellen der Tests die Objekte in unterschiedlichen Methoden in unterschiedlichen Klassen verwendet werden, dabei können diese öfters zugewiesen werden oder als (statische-)Attribute vorliegen. Die Möglichkeiten, wie und wo

ein Objekt benutzt werden kann und in welcher Kombination, scheinen daher grenzenlos zu sein. Aus diesem Grunde wird an dieser Stelle ein Beispiel mit den folgenden Einstellungen präsentiert, welches die Grenzen des Objekt-Trackings verdeutlicht:

Testfall: tf36_objectTracking_assignments_putInObject_moreObjects_static

Parameter	Wert
Java Jar Name	LocalParameter
Entrypoint Class	objectTracking/ObjectTrackingPutMoreStaticObject
EntryPoint Methods	main
Seed Statements	fooMain
CFA Options	ZERO_ONE_CFA
Data Dependence Options	FULL
Control Dependence Options	NO_EXCEPTIONAL_EDGES
Object Tracking	true

Im Programmcode 7.2 ist die ganze Klasse *ObjectTrackingPutMoreStaticObject* abgebildet. In dieser Klasse gibt es die Innere-Klasse *Foo* mit der Methode *fooMain*. In der *main*-Methode der *ObjectTrackingPutMoreStaticObject*-Klasse werden insgesamt drei *Foo*-Objekte (Zeile 15, 20 und 23) erzeugt. Eines davon wird über die *getInstance*-Methode (Zeile 23) erzeugt. Ein viertes Objekt (Zeile 12) wird als Attribut initialisiert. Durch die beiden Seed-Statement-Aufrufe in den Zeilen 18 und 33 dürften nur zwei Objekte und alle ihre dazu zugehörige Zeilen im Slice-Ergebnis erscheinen.

```
1  package objectTracking;
2
3  public class ObjectTrackingPutMoreStaticObject {
4
5      /**
6       * Darf niicht
7       */
8      private static String ne;
9
10     private static Foo fooWow;
11
12     private static Foo fui = new Foo();
13
14     public void main(String[] args) {
15         Foo fooTwo = new Foo();
16         fooTwo.fooNot();
17         fooTwo.init();
18         fooTwo.fooMain();
19
20         fooWow = new Foo();
21
22
23         Foo fooThree = Foo.getInstance(); // new auf Attribute
24         fooThree.fooNot();
25         fooThree.init();
26         fooWow = fooThree;
27         fooWow.initTwo();
28         fooWow = fooTwo;
29         ne = "ne";
30         fooWow.init();
31         fooWow = fui;
32         fooWow.init();
33         fooThree.fooMain();
```

```
34      }
35
36      private static class Foo {
37          private String bla;
38
39          /**
40           * Bla bla
41           * @param pHallo hello
42           * @throws RuntimeException throw things
43           */
44          private void print(String pHallo) {
45              System.out.println(pHallo);
46          }
47
48          public void fooMain() {
49              String hallo = "hallo";
50              print(hallo + bla);
51          }
52
53          public void fooNot() {
54              String moin = "moin";
55              System.out.println(moin);
56          }
57
58          public void init() {
59              bla = "Blaa";
60          }
61
62          public void initTwo() {
63              bla = "New";
64          }
65
66          public static Foo getInstance(){
67              return new Foo();
68          }
69      }
70  }
```

Programmcode 7.2: ObjectTrackingPutMoreStaticObject - Klasse (Quelle: Eigene Darstellung)

Aber in der Abbildung 7.12 sind drei Zeilen im Slice-Ergebnis zu erkennen, welche keine Verbindung zu dem Seed-Statement haben. Diese sind die Zeilen 3, 16 und 17 in der Abbildung 7.12. In Zeile 16 wird *fui* zum ersten Mal in der *main*-Methode verwendet und wird *fooWow* zugewiesen. Zu keinem Zeitpunkt wird auf diesem Objekt das Seed-Statement aufgerufen.

Durch weitere Tests hat sich gezeigt, dass die *interprocedural*-Methode den Fehler verursacht. Weil das Ergebnis aus der *interprocedural*-Methode an die *intraprocedural*-Methode weitergeben wird, werden durch die *intraprocedural*-Methode die Zeilen 3 und 16 erkannt. Dabei handelt es sich um Folgefehler. Die Zeilen 3, 16 und 17 sind somit ein *false positive*.

Wichtig ist es, an dieser Stelle anzumerken, dass nur noch die Zeilen als *false positiv* zu charakterisieren sind, welche in keinem Zusammenhang zu den Objekten des Seed-Statements stehen. Wird daher ein Objekt vom Seed-Statement Zeilen später als die eigentliche Seed-Statement-Zeile weiterhin verwendet, dann wird dies durch das Objekt- oder Parameter-Tracking erkannt und ist ein erwünschter *false positive* und wird daher nicht mehr als *false positive* behandelt.

Warum *interprocedural*-Methode die eine Zeile als "richtig" erkennt, konnte nicht vollständig geklärt werden. Aber es scheint, als würde bei der Pointer-Analyse ein Fehler passieren. Selbst dann wenn andere Einstellungen für das Erzeugen der Pointer-Analyse verwendet worden sind, hat sich das Ergebnis nicht geändert. Ein Blick auf dem Heapgraphen, welcher die Ergebnisse der Pointer-Analyse enthält, verdeutlicht das Problem.

Slice

ObjectTrackingPutMoreStaticObject.java

Show Diff View Side by Side

```
1  package objectTracking;
2  public class ObjectTrackingPutMoreStaticObject {
3      private static Foo fui = new Foo();
4      public void main(String[] args) {
5          Foo fooTwo = new Foo();
6          fooTwo.fooNot();
7          fooTwo.init();
8          fooTwo.fooMain();
9          Foo fooThree = Foo.getInstance(); // new auf Attribute
10         fooThree.fooNot();
11         fooThree.init();
12         fooWow = fooThree;
13         fooWow.initTwo();
14         fooWow = fooTwo;
15         fooWow.init();
16         fooWow = fui;
17         fooWow.init();
18         fooThree.fooMain();
19     }
20
21     private static class Foo {
22         public static Foo getInstance(){
23             return new Foo();
24         }
25     }
26 }
```

Abbildung 7.12: Slice-Ergebnis von ObjectTrackingPutMoreStaticObject-Klasse (Quelle: Eigene Darstellung)

In Abbildung 7.13 ist eine Teilansicht des Heapgraphen zu erkennen. Das Bild wurde beim Debugging gemacht und zeigt die einzelnen Knoten des Graphen. Außerdem würde diese Ansicht auch die Kanten darstellen, indem die Zahlen der Knoten durch einen Pfeil, wie z.B. "$37 \rightarrow 114$" verbunden werden. Dies ist aber in der Abbildung nicht dargestellt. Durch die einzelnen Valuenumbers, welche durch die Instruktionen bezogen worden sind, konnten die Knoten im Graphen identifiziert werden.

```
37  [Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject, main([Ljava/lang/String;)V > Context: Everywhere, v4]
38  [Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject, main([Ljava/lang/String;)V > Context: Everywhere, v5]
39  [Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject, main([Ljava/lang/String;)V > Context: Everywhere, v6]
40  [Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject$Foo, fooNot()V > Context: Everywhere, v1]
41  [Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject, main([Ljava/lang/String;)V > Context: Everywhere, v7]
42  [Exc-Ret-V:Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject$Foo, fooNot()V > Context: Everywhere]
43  [Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject$Foo, init()V > Context: Everywhere, v1]
44  [Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject, main([Ljava/lang/String;)V > Context: Everywhere, v8]
45  [Exc-Ret-V:Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject$Foo, init()V > Context: Everywhere]
46  [Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject$Foo, fooMain()V > Context: Everywhere, v1]
47  [Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject, main([Ljava/lang/String;)V > Context: Everywhere, v9]
48  [Exc-Ret-V:Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject$Foo, fooMain()V > Context: Everywhere]
49  [Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject, main([Ljava/lang/String;)V > Context: Everywhere, v10]
50  [Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject, main([Ljava/lang/String;)V > Context: Everywhere, v11]
51  [< Application, LobjectTracking/ObjectTrackingPutMoreStaticObject, fooWow, <Application,
 LobjectTracking/ObjectTrackingPutMoreStaticObject$Foo> >]
52  [Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject, main([Ljava/lang/String;)V > Context: Everywhere, v13]
53  [Ret-V:Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject$Foo, getInstance()
 LobjectTracking/ObjectTrackingPutMoreStaticObject$Foo; > Context: Everywhere]
54  [Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject, main([Ljava/lang/String;)V > Context: Everywhere, v12]
55  [Exc-Ret-V:Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject$Foo, getInstance()
 LobjectTracking/ObjectTrackingPutMoreStaticObject$Foo; > Context: Everywhere]
56  [Node: < Application, LobjectTracking/ObjectTrackingPutMoreStaticObject, main([Ljava/lang/String;)V > Context: Everywhere, v16]
```

Abbildung 7.13: Debug Teilansicht des Heapgraphen (Quelle: Eigene Darstellung)

In der Abbildung 7.14 wurde die Ansicht des Heapgraphen auf das Wichtigste reduziert. Die Zahlen mit einem *v* am Anfang sind die Valuenumbers. Rechts daneben steht der Name des Objektes. Der Name in den Klammern rechts davon ist das eigentliche Objekt, welches vorher *fooWow* zugewiesen worden ist. Direkt darunter sind die Knotennummer und die Kante zu den anderen Knotennummern abgebildet. z.B. zeigt der Knoten 37 auf den Knoten 144, welcher zum Objekt *fooTwo* mit der Valuenumber *v4* gehört.

Direkt zu erkennen ist, dass *fooWow* auf vier Zahlen zeigt. Weiter unten sind die Zahlen und die dazugehörigen Knoten abgebildet. Es ist zu erkennen, dass es sich um die Initialisierungen der Objekte handelt. Für die Valuenumbers *v4*, *v10*, *v13* und *v21* ist die Zuordnung zu der Initialisierung eindeutig. So ist zu erkennen, dass das Objekt *fooThree* durch die *getInstance()*-Methode erzeugt wird. Die eigentlichen Problemen sind *v16*, *v19* und *v22*. Bei denen ist eine Zuordnung nicht eindeutig möglich. Daher ist nicht klar, zu welchen Objekt diese gehören. Daher wird auch die Zeile 17 durch die *interprocedural*-Methode erkannt.

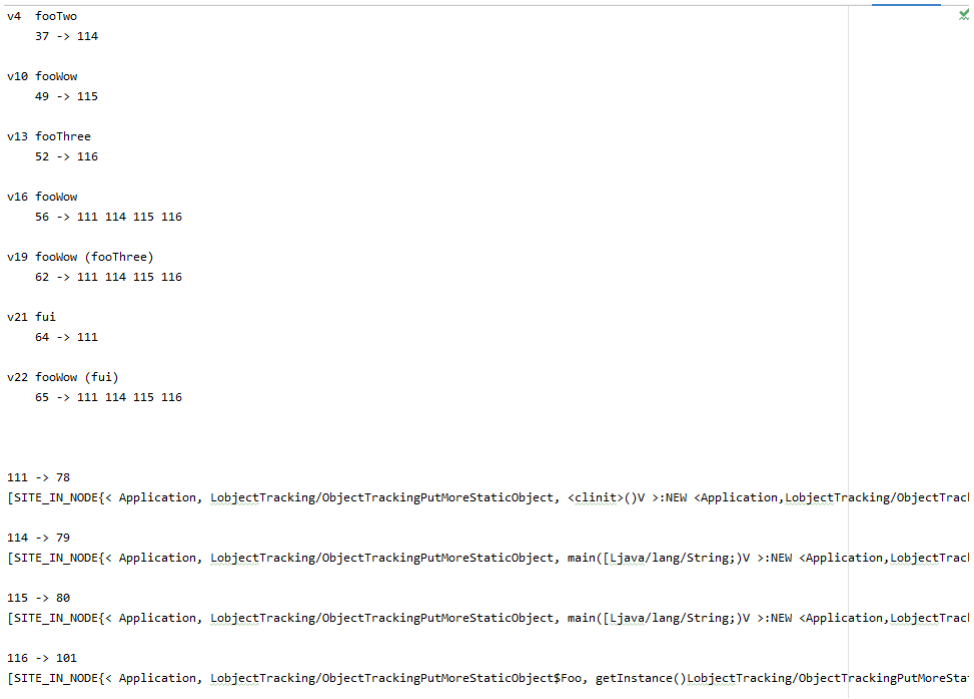


Abbildung 7.14: Vereinfachte und verkleinerte Teilansicht des Heapgraphen (Quelle: Eigene Darstellung)

Es hat sich an dieser Stelle gezeigt, dass das Objekt-Tracking seine Grenzen hat. Das Gute daran ist, dass nur *false positives* erzeugt wurden und sich keine *false negatives* ergeben haben. Auch durch die restlichen Tests für das Objekt-Tracking wurden keine *false negatives* festgestellt.

7.2.4.2 Parameter-Tracking

Die Tests für das Parameter-Tracking befinden sich in der *ParameterTrackingIT*-Klasse. Insgesamt gibt es zwar nur 17 Tests, aber weil der einzige Unterschied zwischen dem Parameter- und dem Objekt-Tracking in der Beschaffung der Valunumbers besteht, decken die Tests des Objekt-Trackings auch gleichzeitig das Paramater-Tracking ab. Deswegen ist auch beim Parameter-Tracking das gleiche Verhalten wie beim Objekt-Tracking zu konstatieren. Es können *false positives* auftreten. Daher wird an dieser Stelle kein weiteres Beispiel für dieses Tracking präsentiert.

7.2.4.3 Lokale Variablen und Attribut-Tracking

Die Tests für das Parameter-Tracking befinden sich in der *ParameterTrackingIT*-Klasse. Wie bereits im Kapitel 6 "Implementierung" im Abschnitt 6.2.8 erwähnt, konnte zwar das Problem, dass einige Attribute oder lokale Variablen im Slice-Ergebnis nicht vorhanden waren, gelöst werden, aber nur dann, wenn WALA auch die Informationen über die lokalen

Variablen oder Attribute gespeichert hat. Auch durch das Tracking können aktuell nicht alle lokalen Variablen oder Attribute erkannt werden, denn das grundlegende Problem, dass einfache Zuweisungen entweder durch Compiler oder durch WALA wegoptimiert werden können, kann durch zusätzliche Algorithmen nicht gelöst werden. Durch die Evaluation hat sich gezeigt, das WALA diese Optimierungen als Konstanten abspeichert (siehe Abschnitt 7.1.1), welche keinen Bezug mehr zu irgendwelchen Objekten haben. Zusätzlich bietet WALA keine *SSAAssignmentInstruction an*. Abgesehen von dieser Einschränkung, findet das Attribut-Tracking dann die lokale Variablen und Attribute, wenn Informationen über diese vorliegen. Dieses wird an dieser Stelle nicht explizit an einem Beispiel gezeigt, weil das Attribut-Tracking auch bei anderen Tests in den nächsten Abschnitten immer einmal wieder genutzt wird oder schon bei anderen Tests gezeigt worden ist.

7.3 Benchmark Tests für Cipher und Signature

Der Slicer soll später Experten bei der Untersuchung von Java-Programmen, welche Kryptografie verwenden, unterstützen. Daher wird eine Reihe von *Cipher*- und *Signature*-Tests angelegt. *Cipher* und *Signature* stammen dabei, wie dies bei den Vorarbeiten der Fall war, aus der *Java Cryptography Architecture*. Dabei verschlüsselt *Cipher* Klartexte, damit diese nicht von jedem gelesen werden können. *Signature* erstellt hingegen eine digitale Signatur, mit welcher eine Nachricht einer bestimmten Person zugeordnet werden kann.
Die Tests für Cipher befinden sich im *cipher* Ordner innerhalb des *security* Ordners. Analog dazu liegen die Tests für *Signature* im *signature* Ordner. Ziel der Tests ist es, herauszufinden, inwieweit die Tracking-Algorithmen das Slicing dann unterstützen, wenn Kryptografie-Klassen, wie *Cipher* oder *Signature* als Objekte für Seed-Statements vorkommen. Das Problem besteht darin, dass beim Slicing von genau denen wichtige Informationen im Slice-Ergebnis nicht vorhanden sein können. Dieses Faktum wird am folgenden Beispiel mit diesen Einstellungen verdeutlicht:

Testfall: tf02_cipher_noTracking

Parameter	Wert
Java Jar Name	LocalParameter
Entrypoint Class	objectTracking/cipher/Simple
EntryPoint Methods	tf02_main
Seed Statements	fooMain
CFA Options	ZERO_ONE_CFA
Data Dependence Options	FULL
Control Dependence Options	NO_EXCEPTIONAL_EDGES

In Abbildung 7.15 ist das Slice-Ergebnis einer sehr einfachen Cipher-Benutzung zu sehen. Über *getInstance()* wird das Objekt erzeugt. An dieser Stelle wird das Verschlüsselungsverfahren übergeben. Als Zweites wird auf das Cipher-Objekt die *init*-Methode aufgerufen. Über die Methode wird festgelegt, ob verschlüsselt oder entschlüsselt werden soll. Über die *update*-Methode wird der Text, welcher verschlüsselt werden soll, als *byte-Array* übergeben. Diese Methode kann öfters oder gar nicht benutzt werden. Zum Schluss wird *doFinal* auf dem Objekt aufgerufen. Auch über diese Methode kann der Text als *byte-Array* übergeben werden. Die *doFinal* wird immer deshalb zum Schluss aufgerufen, weil diese Methode den verschlüsselten Text als *byte-Array* zurückgibt. Obwohl in Abbildung 7.15 die Benutzung von Cipher so einfach wie möglich gehalten worden ist, fehlen die wichtigen Zeilen mit den Methoden *init* und *update*. Genau an dieser Stelle setzen die Tracking-Verfahren an.

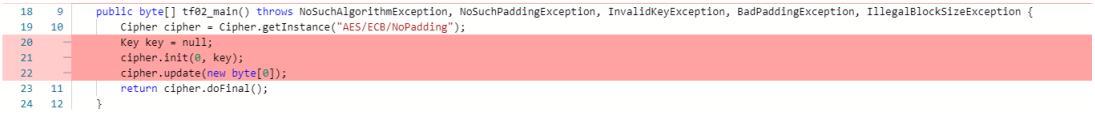


Abbildung 7.15: Slice-Ergebnis - Diff-Ansicht der Klasse Simple - tf02_main (Quelle: Eigene Darstellung)

Für Signature funktioniert der Ablauf der Methoden analog zu Cipher. Dies gilt auch für die meisten restlichen Klassen, welche im Kontext der Kryptographie aus der JCA stammen. Daher werden im Folgenden nur noch Beispiele von Cipher angeschaut. Die Tests für Signature sind analog aufgebaut und sind größtenteils durch das gleiche Verhalten gekennzeichnet.

Abgesehen von den Tests für die einfache Verwendung von Cipher, welche sich in der *CipherSimpleIT*-Testklasse befinden, gibt es noch die *CipherIT*-Testklasse. In dieser gibt es Tests zu Cipher, wobei Cipher komplexer verwendet wird. An dieser Stelle werden die einzelnen Testfälle nicht im Detail erklärt, sondern nur der grobe Aufbau, wie Cipher benutzt worden ist:

- Entschlüsseln/Verschlüsseln der Texte, Speichern/Laden des *SecretKey* uvm. befinden sich in unterschiedlichen Methoden in einer Klasse.
- Entschlüsseln sowie Verschlüsseln befinden sich jeweils nur in einer Methode. (intraprozedural)
- Aufrufe von Cipher sind interprozedural.
- Aufrufe von Cipher sind interprozedural. Die Objekte von Cipher sind Attribute und werden als Attribute initialisiert.
- Aufrufe von Cipher sind interprozedural. Die Objekte von Cipher sind Attribute und werden in den Methoden initialisiert.

Alle Tests von Cipher wurden erst einmal ohne einen Tracking-Algorithmus ausgeführt, dann jeweils mit nur einem eingeschalteten Tracking-Algorithmus und zum Schluss wurde alle eingeschaltet. Insgesamt gibt es daher acht Testklassen:

- CipherSimpleIT
- CipherIT
- CipherObjektTrackingIT
- CipherObjektTrackingTrackingToSliceIT
- CipherParameterTrackingIT
- CipherParameterTrackingTrackingToSliceIT
- CipherLocalVariableAttributeTRackingIT
- CipherAllTRackingIT

Die eben genannten Testklassen werden mit der Datenabhängigkeitsoption *FULL* ausgeführt. Analog dazu gibt die gleichen Testklassen nochmals, nur werden dort die Datenabhängigkeitsoption *NO_HEAP* verwendet. Als Beispieltest wurden folgende Einstellung ausgewählt:

Testfall: CipherIT - tf02_cipher_allInOneClass_null_encryptMessage_doFinal

Parameter	Wert
Java Jar Name	LocalParameter
Entrypoint Class	objectTracking/cipher/AllInOneClass
Method Names	encryptMessage
Seed Statements	doFinal
CFA Options	ZERO_ONE_CFA
Data Dependence Options	FULL
Control Dependence Options	NO_EXCEPTIONAL_EDGES

Die vollständige Klasse ist im Anhang A.5 zu finden. Der Abbildung 7.16 ist das Slice-Ergebnis zu entnehmen. Im Vergleich zur ganzen Klasse ist im Slice-Ergebnis nicht mehr viel vorhanden. Die Attribute und der Konstruktor fehlen komplett. Es ist nicht zu erkennen, mit welchen Einstellungen die Cipher instanziiert und initialisiert wurde.

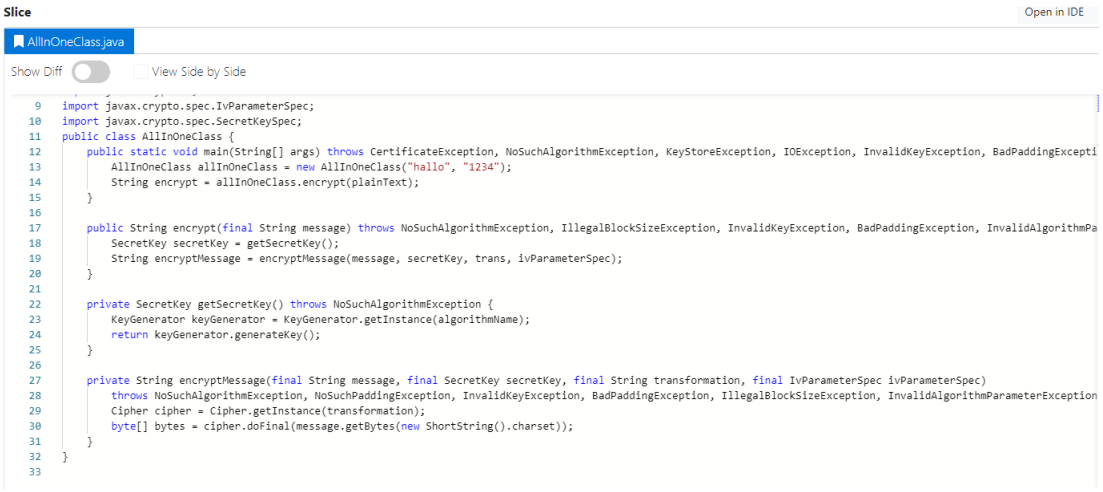


Abbildung 7.16: Slice-Ergebnis - AllInOneClass - kein Tracking (Quelle: Eigene Darstellung)

In den Abbildungen 7.17 und 7.18 sind die Slice-Ergebnisse mit allen aktivierten Tracking-Algorithmen zu erkennen. Es wurden sonst die gleichen Einstellungen wie die ebenen beschriebenen verwendet, und zwar mit der einzigen Ausnahme, dass die *CFA Option* auf *N_CFA* sowie das CFA-Level auf *1* gestellt worden ist. Wie bei den Tests beim Objekt-Tracking beschrieben, hat die *interprocedural*-Methode manchmal Probleme damit, Objekte unterscheiden zu können.

Dieses Faktum wurde auf die Pointer-Analyse zurückgeführt. Anders als bei dem Beispiel vom Objekt-Tracking Test, hat bei diesem Test die Veränderung der Einstellung zur Erzeugung der Pointer-Analyse zu einem besseren Ergebnis geführt. Mit der alten Einstellung wären auch die Methoden *decryptMessage*, *encryptSecretKey* und *decryptSecretKey* im Slice-Ergebnis vorhanden gewesen. Alle drei Methoden sind dabei irrelevant für die *encryptMessage*-Methode. Daher hat sich bei diesen und auch anderen Tests zu Cipher gezeigt, dass das Problem mit der Pointer-Analyse zumindest bei Cipher gelöst werden kann. Interessanterweise gab es bei der Signatur dieses Problem nicht.

Slice

Open in IDE

AllInOneClass.java

Show Diff View Side by Side

```
11  public class AllInOneClass {
12      private String trans = "AES/CBC/PKCS5Padding";
13      private IvParameterSpec ivParameterSpec;
14      private class ShortString {
15          public Charset charset = StandardCharsets.UTF_8;
16      }
17
18      public static void main(String[] args) throws CertificateException, NoSuchAlgorithmException, KeyStoreException, IOException, InvalidKeyException, BadPaddingExcepti
19          AllInOneClass allInOneClass = new AllInOneClass("hallo", "1234");
20          String plainText = "Hallo";
21          String encrypt = allInOneClass.encrypt(plainText);
22      }
23
24      public AllInOneClass(final String secretKeyAlias, final String secretKeyPassword) throws CertificateException, NoSuchAlgorithmException, KeyStoreException, IOExcept
25          ivParameterSpec  = getIvParameterSpec();
26      }
27
28      public String encrypt(final String message) throws NoSuchAlgorithmException, IllegalBlockSizeException, InvalidKeyException, BadPaddingException, InvalidAlgorithmPa
29          SecretKey secretKey = getSecretKey();
30          String encryptMessage = encryptMessage(message, secretKey, trans, ivParameterSpec);
31      }
32
33      private SecretKey getSecretKey() throws NoSuchAlgorithmException {
34          String algorithmName = "AES";
35          KeyGenerator keyGenerator = KeyGenerator.getInstance(algorithmName);
36          return keyGenerator.generateKey();
```

Abbildung 7.17: Slice-Ergebnis - AllInOneClass - mit Tracking - Teil 1 (Quelle: Eigene Darstellung)

Slice

Open in IDE

AllInOneClass.java

Show Diff View Side by Side

```
29          SecretKey secretKey = getSecretKey();
30          String encryptMessage = encryptMessage(message, secretKey, trans, ivParameterSpec);
31      }
32
33      private SecretKey getSecretKey() throws NoSuchAlgorithmException {
34          String algorithmName = "AES";
35          KeyGenerator keyGenerator = KeyGenerator.getInstance(algorithmName);
36          return keyGenerator.generateKey();
37      }
38
39      private IvParameterSpec getIvParameterSpec() throws NoSuchAlgorithmException {
40          SecureRandom randomSecureRandom = SecureRandom.getInstance("SHA1PRNG");
41          byte[] iv = new byte[16];
42          randomSecureRandom.nextBytes(iv);
43          return new IvParameterSpec(iv);
44      }
45
46      private String encryptMessage(final String message, final SecretKey secretKey, final String transformation, final IvParameterSpec ivParameterSpec)
47          throws NoSuchAlgorithmException, NoSuchPaddingException, InvalidKeyException, BadPaddingException, IllegalBlockSizeException, InvalidAlgorithmParameterExceptio
48          Cipher cipher = Cipher.getInstance(transformation);
49          cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivParameterSpec);
50          byte[] bytes = cipher.doFinal(message.getBytes(new ShortString().charset));
51      }
52  }
53
```

Abbildung 7.18: Slice-Ergebnis - AllInOneClass - mit Tracking - Teil 2 (Quelle: Eigene Darstellung)

Zusätzlich ist noch interessant zu beobachten, dass das Slice-Ergebnis viel mehr Informationen als ohne Tracking beinhaltet. Zum einen sind zwei Attribute (*trans*, *ivParameterSpec*) und eine Innere-Klasse *ShortString* und deren Attribut *charset* zu erkennen. Das Attribut *ivParameterSpec* wird dabei im Konstruktor über die Methode *getIvParameterSpec()* instanziiert. Der *plaintText* "Hallo" ist in der *main*-Methode zu sehen. Alle wichtigen Informationen, wie das Cipher-Objekt instanziiert und initialisiert wurde, sind ersichtlich. Das Einzige was fehlt, ist die Initialisierung vom *keyGenerator*. Aber im Großen und Ganzen hat sich gezeigt, dass die Tracking-Algorithmen beim Slicen sehr hilfreich sind, besonders dann, wenn die Datenabhängigkeitsoption *NO_HEAP* ausgewählt worden ist.

7.4 OSCI Tests

OSCI ist ein Java-Protokollstandard für die sichere, vertrauliche und rechtsverbindliche Übertragung elektronischer Daten im E-Government in Deutschland. Es wird im Auftrag der Koordinierungsstelle für IT-Standards (KoSIT) betrieben und wird als Kommunikation zwischen den Behörden benutzt. [KoSIT, 2021]
Die OSCI-Bibliothek beinhaltet Beispiele, in denen gezeigt wird, wie diese Bibliothek genutzt werden kann. Anhand zwei dieser Beispiele wurde der Slicer getestet. Als Beispiele wurden die Klassen *OneWayMessage_PassiveRecipient* und *OneWayMessage_ActiveRecipient* ausgewählt, welche auch jeweils als Einstiegspunkte für den Slicer genommen wurden. Das erste Beispiel ist eine Demo für synchrone Kommunikation mit einer Nachricht, die an einen passiven Empfänger gerichtet ist. Das zweite Beispiel zeigt eine asynchrone Kommunikation, in der die Nachricht an einem aktiven Empfänger gesendet wird. Genauer wird auf OSCI-Bibliothek und die beiden Beispiele nicht eingegangen, weil nur die Ergebnisse des Slicers weiter betrachtet werden. Insbesondere wird darauf geachtet, ob und welche Parameter und Methoden-Aufrufe für die Cipher- und Signature-Objekte erkannt werden. Im ersten Beispiel wird das Cipher-Objekt untersucht. Dafür wird ein Slice mit diesen Einstellungen erzeugt:

Testfall: OSCI - OneWayMessage_PassiveRecipient

Parameter	Wert
Java Jar Name	osci
Entrypoint Class	de/osci/osci12/samples/OneWayMessage_PassiveRecipient
EntryPoint Methods	main
Method Class	de/osci/helper/SymCipherOutputStream
Method Names	close
Seed Statements	doFinal
CFA Options	N_CFA
CFA Level	1
Data Dependence Options	NO_HEAP
Control Dependence Options	NO_EXCEPTIONAL_EDGES
Object Tracking	false/true
Parameter Tracking	false/true
Tracking to Slicing Criterion	false/true
Local Variable & Attribute Tracking	false/true

Für ein besseres Verständnis, in welcher Klasse das Cipher-Objekt benutzt wird, ist in Abbildung 7.19 der Datenfluss von der *OneWayMessage_PassiveRecipient*-Klasse zur *doFinal*-Methode in der *SymCipherOutputStream*-Klasse zu erkennen. Das Cipher-Objekt

wird daher in den Klassen *SymCipherOutputStream* und *Crypto* verwendet. Die Klasse *Crypto* wird aber nicht weiter betrachtet, weil es in der Klasse keinen Unterschied zwischen den Slice-Ergebnissen mit und ohne Tracking-Algorithmen gibt.

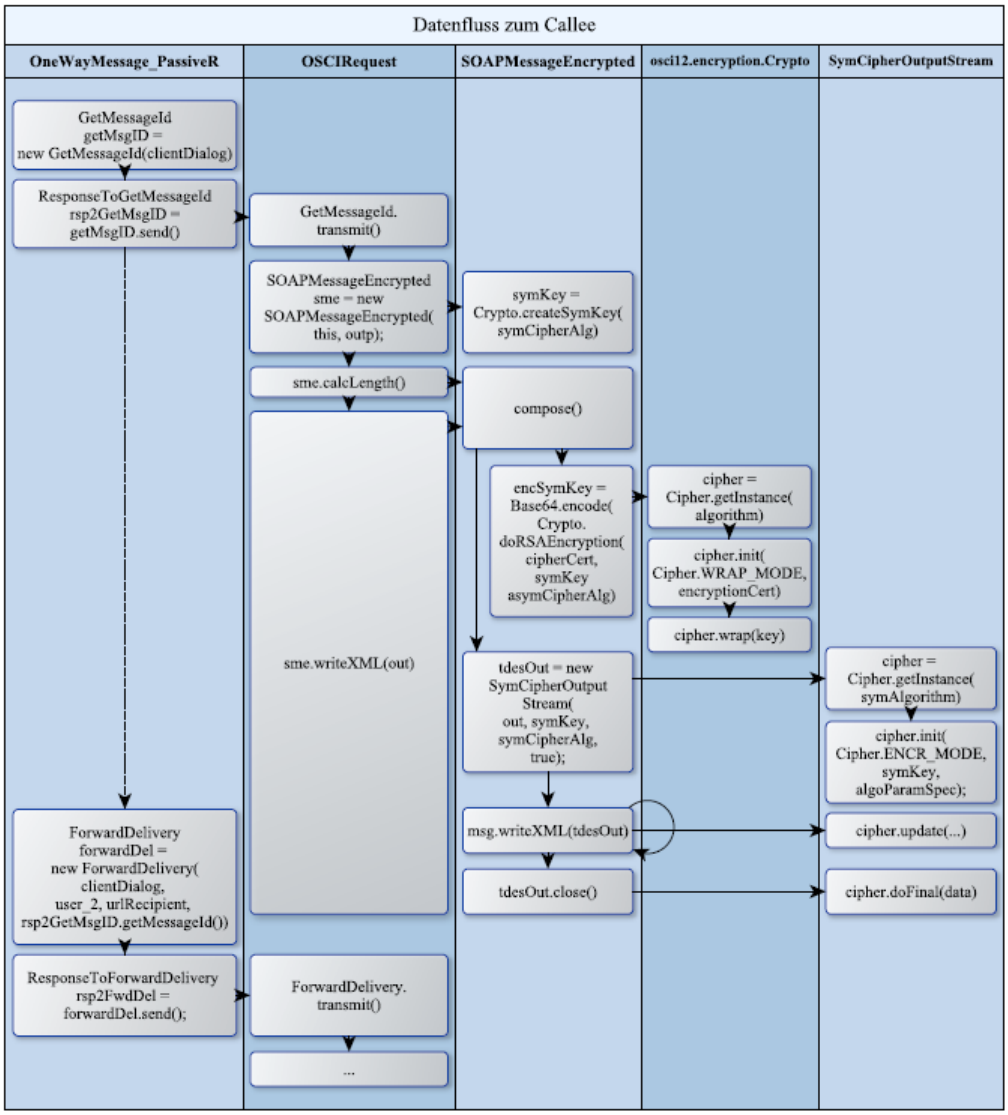


Abbildung 7.19: Datenflüsse zwischen OneWayMessage_PassiveRecipient und der doFinal-Methode [Nguyen, 2018, Seite 130]

In Abbildung 7.20 ist das Slice-Ergebnis ohne ein aktiviertes Tracking-Algorithmus zu erkennen. Im Slice-Ergebnis ist nur der Aufruf von *doFinal* zu sehen. Alles andere wurde nicht erkannt.

Abbildung 7.20: Slice-Ergebnis - SymCipherOutputStream - ohne Tracking (Quelle: Eigene Darstellung)

Im Gegensatz dazu ist im Programmcode 7.3 zu erkennen, dass nicht nur die *doFinal*-Methode erkannt worden ist, sondern auch die Methode *write* sowie der Konstruktor der Klasse *SymCipherOutputStream*. Zur besseren Darstellung ist der ganze Programmcode des Slice-Ergebnisses abgebildet. Zu aller erst ist kein syntaktischer Fehler bei der Rekonstruktion der Java-Datei zu erkennen. Die import-Anweisungen sowie der Package sind vollständig enthalten. Im Konstruktor in den Zeilen 53 sowie 58 und 59 wird das Cipher-Objekt über *getInstance* instantiiert. Als Parameter wird zu einem *Constants.JCA_JCE_MAP.get(symAlgorithm)* und zum anderen *DialogHandler.getSecurityProvider()* übergeben.

```
1  package de.osci.helper;
2  import java.io.FilterOutputStream;
3  import java.io.IOException;
4  import java.io.OutputStream;
5  import java.security.GeneralSecurityException;
6  import java.security.NoSuchAlgorithmException;
7  import javax.crypto.Cipher;
8  import javax.crypto.SecretKey;
9  import javax.crypto.spec.IvParameterSpec;
10 import org.apache.commons.logging.Log;
11 import org.apache.commons.logging.LogFactory;
12 import de.osci.osci12.common.Constants;
13 import de.osci.osci12.common.Constants.LanguageTextEntries;
14 import de.osci.osci12.common.DialogHandler;
15 import de.osci.osci12.encryption.Crypto;
16 /**
17  * Diese Erweiterung der Klasse java.io.FilterOutputStream fhrt eine symmetrische Ver-/Entschlsselung der
18  * geschriebenen Daten durch. Der Initialisierungsvektor wird gem der XML-Encryption-Spezifikation den
19  * Inhaltsdaten hinzugefgt bzw. entnommen.
```

```
20  * <p>
21  * Copyright  2020 Governikus GmbH &amp;amp; Co. KG, Germany
22  * </p>
23  * <p>Erstellt von Governikus GmbH &amp;amp; Co. KG</p>
24  * <p>
25  * Diese Bibliothek kann von jedermann nach Magabe der European Union Public Licence genutzt
26  * werden.
27  * </p>
28  * Die Lizenzbestimmungen knnen unter der URL <a href="https://eupl.eu/">https://eupl.eu/</a> abgerufen werden.
29  * </p>
30  *
31  * @author R. Lindemann, N.Bngener
32  * @version 1.9.0
33  * @see de.osci.helper.SymCipherInputStream
34  */
35 public class SymCipherOutputStream extends FilterOutputStream
36 {
37   private Cipher cipher;
38   private byte[] iv;
39   private SecretKey symKey;
40   SymCipherOutputStream(OutputStream outStream,
41                         SecretKey symKey,
42                         String symAlgorithm,
43                         boolean encrypt,
44                         int ivLength,
45                         byte[] iv)
46     throws IOException
47   {
48     this.symKey = symKey;
49     try
50     {
51       if (DialogHandler.getSecurityProvider() == null)
52       {
53         cipher = Cipher.getInstance(Constants.JCA_JCE_MAP.get(symAlgorithm));
54       }
55
56       else
57       {
58         cipher = Cipher.getInstance(Constants.JCA_JCE_MAP.get(symAlgorithm),
59                                     DialogHandler.getSecurityProvider());
60       }
61
62       if (symAlgorithm.endsWith("tripledes-cbc"))
63       {
64         this.iv = new byte[8];
65       }
66
67       else if (symAlgorithm.endsWith("-gcm"))
68       {
69         if (ivLength == 16 || ivLength == 12)
70         {
71           this.iv = new byte[ivLength];
72         }
73
74         else
75       }
76
77       {
78         this.iv = new byte[16];
79       }
80
81       if (encrypt)
82       {
83         if (iv == null)
84           this.iv = Tools.createRawRandom(this.iv.length);
85         IvParameterSpec algoParamSpec = new javax.crypto.spec.IvParameterSpec(this.iv);
86         cipher.init(Cipher.ENCRYPT_MODE, symKey, algoParamSpec);
87       }
88     }
89
90     catch (GeneralSecurityException ex)
91     {
92     }
93   }
94
95   /**
96    * undocumented
97    *
```

```
98      * @param b undocumented
99      * @param off undocumented
100     * @param len undocumented
101     * @throws IOException undocumented
102     * @throws IndexOutOfBoundsException undocumented
103     */
104   @Override
105   public void write(byte[] b, int off, int len) throws IOException
106   {
107     if ((off | len | (b.length - (len + off)) | (off + len)) < 0)
108       throw new IndexOutOfBoundsException();
109     if ((!encrypt) && (index < iv.length))
110     {
111       if (len < (iv.length - index))
112       {
113         for ( int i = 0 ; i < len ; i++ )
114           iv[index++] = b[off + i];
115       }
116
117       else
118       {
119         int i;
120         for ( i = 0 ; i < (iv.length - index) ; i++ )
121           iv[i + index] = b[off + i];
122         try
123         {
124           IvParameterSpec algoParamSpec = new javax.crypto.spec.IvParameterSpec(iv);
125           cipher.init(Cipher.DECRYPT_MODE, symKey, algoParamSpec);
126           tmp = cipher.update(b, off + i, len - i);
127         }
128
129         catch (Exception ex)
130         {
131         }
132       }
133     }
134
135     {
136       // out.write(b, off, len);
137       if (b.length > 0)
138       {
139         tmp = cipher.update(b, off, len);
140       }
141     }
142   }
143
144   /**
145    * undocumented
146    *
147    * @throws IOException undocumented
148    */
149   @Override
150   public void close() throws IOException
151   {
152     try
153     {
154       out.write(cipher.doFinal());
155     }
156
157     catch (Exception ex)
158     {
159     }
160   }
161 }
```

Programmcode 7.3: Slice-Ergebnis - SymCipherOutputStream - mit Tracking (Quelle: Eigene Darstellung)

In den Zeilen 84-86 wird das Cipher-Objekt initialisiert. Dafür wir ein neues *IvParameterSpec*-Objekt erzeugt. Dieses Objekt sowie die Parameter *Cipher.ENCRYPT_MODE*, *symKey* und *algoParamSpec* werden an dieser Stelle der *init*-Methode übergeben. Des Weiteren ist zu erkennen, wie das *iv*-Array für das *IvParameterSpec*-Objekt entstanden

ist.
Auch in der *write*-Methode wird das Cipher-Objekt initialisiert (Zeile 124-125). Dieses mal aber im *Cipher.DECRYPT_MODE*. In der *write*-Methode wird daher wieder das *IvParameterSpec*-Objekt, sowie das *iv*-Array richtig erkannt. Zum Schluss wird auf das Cipher-Objekt die *update*-Methode aufgerufen. Es hat sich also gezeigt, dass das Cipher-Objekt überall gefunden wurde, wo dieser im Datenfluss auch benutzt wurde.

Die nächste Frage die sich stellt, ist ob auch die verwendeten Parameter, welche aus anderen Klassen kommen, erkannt wurden. Als aller erstes wird *symAlgorithm* betrachtet. *SymCipherOutputStream* wird in der Methode *writeXML* der Klasse *SOAPMessageEncrypted* erzeugt. Übergeben wird dabei das Attribut *symmetricCipherAlgorithm*, welcher im Konstruktor der Klasse *SOAPMessageEncrypted* zugewiesen wird, siehe Abbildung 7.21. Es wird daher die *getSymmetricCipherAlgorithm*-Methode der *DialogHandler*-Klasse aufgerufen.
Die Methode *getSymmetricCipherAlgorithm* ist aber im Slice-Ergebnis der *DialogHandler*-Klasse nicht enthalten. Dies liegt vielleicht daran, dass die Methode eine Konstante zurückgibt.

```
public SOAPMessageEncrypted(OSCIMessage msg, OutputStream storeStream)
  throws NoSuchAlgorithmException
{
  this.msg = msg;
  if (this.msg != null)
  {
    this.symmetricCipherAlgorithm = msg.dialogHandler.getSymmetricCipherAlgorithm();
    symKey = Crypto.createSymKey(symmetricCipherAlgorithm);
  }

  this.storeStream = storeStream;
}
```

Abbildung 7.21: Slice-Ergebnis - SOAPMessageEncrypted - nur der Konstruktor (Quelle: Eigene Darstellung)

DialogHandler.getSecurityProvider() welcher auch für das *getInstance* des Cipher-Objekts verwendet wird, ist vollständig im Slice enthalten. Auch die Initialisierung über ein static-Block wurde richtig erkannt, siehe Abbildung 7.22.

```
46  public class DialogHandler
47  {
48    private static Provider securityProvider;
49    static
50    {
51      try
52      {
53        securityProvider = (Provider)Class.forName("org.bouncycastle.jce.provider.BouncyCastleProvider")
54                                                  .newInstance();
55      }
56
57      catch (Exception ex)
58      {
59      }
60    }
```

Abbildung 7.22: Slice-Ergebnis - DialogHandler - static-Block (Quelle: Eigene Darstellung)

Der letzte interessante Parameter ist *symkey*. Dieser wird auch im Konstruktor der *SOAP-MessageEncrypted*-Klasse durch den Methoden-Aufruf *Crypto.createSymKey* erzeugt, siehe Abbildung 7.21. Die Methode ist aber im Slice in der Klasse *Crypto* nicht enthalten.
Insgesamt hat sich an diesen Beispiel gezeigt, dass das Cipher-Objekt vollständig erkannt wurde. Auch die Herleitung der meisten Parameter ist vorhanden und selbst wenn diese nicht vollständig da sind, ist es über die Diff-Ansicht sehr leicht gewesen, die restlichen Informationen zu erhalten. Es könnten auch über einen zusätzlichen Slice mit anderen Einstellungen die restlichen Informationen gewonnen werden.

Im zweiten Beispiel wird das Signature-Objekt genauer angesehen und ob dort das gleiche Verhalten bezüglich der Zurückverfolgung der Parameter zu sehen ist. Dafür werden zwei Slices mit den folgenden Einstellung erzeugt.

Testfall: OSCI - OneWayMessage_ActiveRecipient

Parameter	Wert
Java Jar Name	osci
Entrypoint Class	de/osci/osci12/samples/OneWayMessage_ActiveRecipient
EntryPoint Methods	main
Method Class	de/osci/osci12/messageparts/ContentContainer
Method Names	checkSignature
Seed Statements	verify
CFA Options	N_CFA
CFA Level	1
Data Dependence Options	NO_HEAP
Control Dependence Options	NO_EXCEPTIONAL_EDGES
Object Tracking	false/true
Parameter Tracking	false/true
Tracking to Slicing Criterion	false/true
Local Variable & Attribute Tracking	false/true

In der *main*-Methode der *OneWayMessage_ActiveRecipient*-Klasse, wird die *checkAllSignatures* aufgerufen, welche wiederum *checkSignature* aufruft und in dieser Methode befindet sich das Seed-Statement *verify*. Aber aus unerklärlichen Gründen ist im Aufrufgraph weder die *checkAllSignatures*- noch die *checkSignature*-Methode zu finden. Deshalb wird auch kein Seed-Statement gefunden und auch kein Slice erstellt. Es wurden die Einstellungen mit unterschiedlichen Angaben wiederholt z.B. wurde die *CFA Option* verändert, aber das Ergebnis hat sich nicht geändert. Bis zum Schluss konnte dieses Phänomen nicht erklärt werden.
Deswegen wurde zum Schluss der Einstiegspunkt geändert:

Parameter	Wert
Entrypoint Class	de/osci/osci12/messageparts/ContentContainer
EntryPoint Methods	checkAllSignatures

Das Slice-Ergebnis ohne Tracking ist im Programmcode 7.4 zu sehen. Zur besseren Darstellung wurde das JavaDoc und die Imports entfernt. Vorneweg kann schon gesagt werden, dass das gleiche Verhalten wie bei Cipher aufgetreten ist. In den Zeilen 83-87 wird das Signature-Objekt erzeugt und auch verwendet. Aber es fehlen die Zeilen in denen die Methoden *initVerify* und *update* benutzt werden. Außerdem fehlt z.B. auch das Objekt *mdg*, welches in der Zeile 50 verwendet wird.

```
1  public class ContentContainer extends MessagePart implements Serializable
2  {
3
4    public boolean checkSignature(Role signatureRole) throws OSCISignatureException, OSCIRoleException
5    {
6      if (check)
7      {
8        try
9        {
```

```
10        OSCISignature[] signatures = findSignatureObjects(signatureRole);
11        for ( int j = 0 ; j < signatures.length ; j++ )
12        {
13          // id
14          for ( int i = 0 ; i < contentList.size() ; i++ )
15          {
16          }
17
18          for ( int i = 0 ; i < encryptedDataList.size() ; i++ )
19          {
20          }
21
22          while (e.hasMoreElements())
23          {
24          }
25
26          if (log.isDebugEnabled())
27          {
28            for ( String refId : sigRefs.keySet() )
29            {
30            }
31          }
32
33          int sigRefCount= sigRefs.size();
34          if (signatures[j].signingTime!=null)
35          {
36            sigRefCount--;
37          }
38
39          if (sigRefCount != newHashes.size())
40          {
41            return false;
42          }
43
44          // sind die Hashwerte gleich?
45          for ( OSCISignatureReference signatureRef : sigRefs.values() )
46          {
47            if (id.equals("#" + signatures[j].signingPropsId))
48            {
49              else
50              newDigest = mdg.digest(signatures[j].signingProperties.getBytes(Constants.CHAR_ENCODING));
51            }
52
53            {
54              newDigest = mp.getDigestValue(signatureRef.getDigestMethodAlgorithm());
55            }
56
57            digest = signatureRef.digestValue;
58            if (newDigest == null)
59            {
60              return false;
61            }
62
63            if (!Tools.compareByteArrays(digest, newDigest))
64            {
65              return false;
66            }
67          }
68
69          for ( Enumeration<String> en = newHashes.keys() ; en.hasMoreElements() ; )
70          {
71            if (!checked.contains(key))
72            {
73              return false;
74            }
75          }
76
77          if ((c.getKeyUsage() != null) && !c.getKeyUsage()[0] && !c.getKeyUsage()[1])
78          {
79            return false;
80          }
81
82          if (DialogHandler.getSecurityProvider() == null)
83            sg = java.security.Signature.getInstance(Constants.JCA_JCE_MAP.get(signatures[j].signatureAlgorithm
                  ));
84          else
85            sg = java.security.Signature.getInstance(Constants.JCA_JCE_MAP.get(signatures[j].signatureAlgorithm
                  ),
```

```
86                                                    DialogHandler.getSecurityProvider());
87            if (!sg.verify(signatures[j].signatureValue))
88            {
89              return false;
90            }
91          }
92        }
93
94        catch (IllegalStateException ex)
95        {
96        }
97
98        catch (Exception ex)
99        {
100       }
101     }
102
103     return true;
104   }
105
106
107   private OSCISignature[] findSignatureObjects(Role roleToCheck) throws OSCIRoleException
108   {
109     return sigs.toArray(new OSCISignature[0]);
110   }
111
112
113   public boolean checkAllSignatures() throws OSCIRoleException, OSCISignatureException
114   {
115     Role[] signer = this.getSigners();
116     for ( int i = 0 ; i < signer.length ; i++ )
117     {
118       if (!checkSignature(signer[i]))
119       {
120         return false;
121       }
122     }
123   }
124
125
126   public Role[] getSigners()
127   {
128     Role[] roles = new Role[signerList.size()];
129     return roles;
130   }
131 }
```

Programmcode 7.4: Slice-Ergebnis - ContentContainer - ohne Tracking (Quelle: Eigene Darstellung)

Im Vergleich dazu ist im Programmcode 7.5 der Slice mit Tracking abgebildet. In den Zeilen 124 und 125 werden die Methoden *initVerify* und *update* verwendet. Es ist auch zu sehen, dass der *PublicKey* (Zeile 124) bzw. das *X509Certificate*-Objekt für das Signature-Objekt in Zweile 110 erzeugt wird. Weil der Einstiegspunkt die Methode *checkAllSignatures* ist, ist nicht zu erkennen wo das *signatureRole* herkommt, weil dieser als Parameter dieser Methode übergeben wird.
Auch das *MessageDigest*-Objekt (*mdg*) wird erkannt. Es wird in den Zeilen 74-80 initialisiert und verwendet. Die *getDigestMethodAlgorithm*-Methode, welche zur Initialisierung genutzt wird, ist in der *OSCISignatureReference*-Klasse im Slice nicht enthalten.

```
1 public class ContentContainer extends MessagePart implements Serializable
2 {
3   private static Log log = LogFactory.getLog(ContentContainer.class);
4   // Vector fr die Signer dieses ContentContainers
5   Vector<OSCISignature> signerList = new Vector<OSCISignature>();
6   // Vector fr alle Roles des ContentContainer. Beinhalten auch die Rollen Objekte eventueller Encrypted Data
7   // Childs
8   Vector<Role> roles = new Vector<Role>();
```

```
9   // Vector fr die Contents
10  private Vector<Content> contentList = new Vector<Content>();
11  // Vector fr die EncryptedData Objekte
12  private Vector<EncryptedDataOSCI> encryptedDataList = new Vector<EncryptedDataOSCI>();
13  /**
14   * Legt ein ContentContainer-Objekt an.
15   */
16  public ContentContainer()
17  {
18    id = typ + (++idNr);
19  }
20
21
22  public boolean checkSignature(Role signatureRole) throws OSCISignatureException, OSCIRoleException
23  {
24    if (check)
25    {
26      try
27      {
28        OSCISignature[] signatures = findSignatureObjects(signatureRole);
29        for ( int j = 0 ; j < signatures.length ; j++ )
30        {
31          Hashtable<String, MessagePart> newHashes = new Hashtable<String, MessagePart>();
32          // id
33          for ( int i = 0 ; i < contentList.size() ; i++ )
34          {
35          }
36
37          for ( int i = 0 ; i < encryptedDataList.size() ; i++ )
38          {
39          }
40
41          Enumeration<Attachment> e = attachments.elements();
42          while (e.hasMoreElements())
43          {
44          }
45
46          Map<String, OSCISignatureReference> sigRefs = signatures[j].getReferences();
47          if (log.isDebugEnabled())
48          {
49            for ( String refId : sigRefs.keySet() )
50            {
51            }
52          }
53
54          Vector<String> checked = new Vector<String>();
55          MessagePart mp;
56          byte[] digest, newDigest;
57          int sigRefCount= sigRefs.size();
58          if (signatures[j].signingTime!=null)
59          {
60            sigRefCount--;
61          }
62
63          if (sigRefCount != newHashes.size())
64          {
65            return false;
66          }
67
68          // sind die Hashwerte gleich?
69          for ( OSCISignatureReference signatureRef : sigRefs.values() )
70          {
71            String id = signatureRef.getRefID();
72            if (id.equals("#" + signatures[j].signingPropsId))
73            {
74              MessageDigest mdg;
75              if (DialogHandler.getSecurityProvider() == null)
76                mdg = MessageDigest.getInstance(Constants.JCA_JCE_MAP.get(signatureRef.getDigestMethodAlgorithm
                      ()));
77              else
78                mdg = MessageDigest.getInstance(Constants.JCA_JCE_MAP.get(signatureRef.getDigestMethodAlgorithm
                      ()),
79                                                DialogHandler.getSecurityProvider());
80              newDigest = mdg.digest(signatures[j].signingProperties.getBytes(Constants.CHAR_ENCODING));
81            }
82
83            {
84              mp = newHashes.get(id);
```

```
85              newDigest = mp.getDigestValue(signatureRef.getDigestMethodAlgorithm());
86            }
87
88            digest = signatureRef.digestValue;
89            if (newDigest == null)
90            {
91              return false;
92            }
93
94            if (!Tools.compareByteArrays(digest, newDigest))
95            {
96              return false;
97            }
98          }
99
100         String key;
101         for ( Enumeration<String> en = newHashes.keys() ; en.hasMoreElements() ; )
102         {
103           key = en.nextElement();
104           if (!checked.contains(key))
105           {
106             return false;
107           }
108         }
109
110         X509Certificate c = signatureRole.getSignatureCertificate();
111         if ((c.getKeyUsage() != null) && !c.getKeyUsage()[0] && !c.getKeyUsage()[1])
112         {
113           return false;
114         }
115
116         // nun noch das Signed Info testen
117         java.security.Signature sg;
118         if (DialogHandler.getSecurityProvider() == null)
119           sg = java.security.Signature.getInstance(Constants.JCA_JCE_MAP.get(signatures[j].signatureAlgorithm
                ));
120         else
121           sg = java.security.Signature.getInstance(Constants.JCA_JCE_MAP.get(signatures[j].signatureAlgorithm
                ),
122                                                    DialogHandler.getSecurityProvider());
123         // sg.initVerify(signatureRole.getSignatureCertificate());
124         sg.initVerify(c.getPublicKey());
125         sg.update(signatures[j].getSignedInfoBytes());
126         if (!sg.verify(signatures[j].signatureValue))
127         {
128           return false;
129         }
130       }
131     }
132
133     catch (IllegalStateException ex)
134     {
135     }
136
137     catch (Exception ex)
138     {
139     }
140   }
141
142   return true;
143 }
144
145
146 private OSCISignature[] findSignatureObjects(Role roleToCheck) throws OSCIRoleException
147 {
148   Enumeration<OSCISignature> e = this.signerList.elements();
149   Vector<OSCISignature> sigs = new Vector<OSCISignature>();
150   return sigs.toArray(new OSCISignature[0]);
151 }
152
153
154 public boolean checkAllSignatures() throws OSCIRoleException, OSCISignatureException
155 {
156   Role[] signer = this.getSigners();
157   for ( int i = 0 ; i < signer.length ; i++ )
158   {
159     if (!checkSignature(signer[i]))
160     {
```

```
161         return false;
162       }
163     }
164   }
165
166
167   public Role[] getSigners()
168   {
169     Role[] roles = new Role[signerList.size()];
170     return roles;
171   }
172 }
```

Programmcode 7.5: Slice-Ergebnis - ContentContainer - mit Tracking (Quelle: Eigene Darstellung)

Im großen und ganzen hat sich beim Testen an der OSCI-Bibliothekt gezeigt, dass sich das Slice-Ergebnis sehr gut verbessert, wenn alle Tracking-Algorithmen eingeschaltet sind. Die Methoden *getInstance*, *init*, *initVerify* uvm. für die Cipher und Signature-Objekte konnten immer vollständig gefunden werden. Auch der Weg und die Verwendung der einzelnen Parameter für diese Methoden konnte meistens verfolgt und erkannt werden. Besonders die Diff-Ansicht hat dabei geholfen, Konstanten oder ein paar Methoden, welche nicht im Slice-Ergebnis waren, anzuzeigen. Trotzdem ist dies im Vergleich zu keinem aktivierten Tracking eine sehr große Verbesserung.

7.5 Zusammenfassung des Kapitels

In diesen Kapitel wurde zuallererst aufgezeigt, wie die alten Beispiele der Vorarbeiten in JUnit-Tests umgebaut wurden, besonders, welche Ergebnisse diese geliefert haben. Durch diese und die zusätzlichen JUnit-Tests konnten die Grenzen des Slicers und der einzelnen Algorithmen konkretisiert werden. Unter anderem hat daraus resultiert, dass nicht alle lokalen Variablen und Attribute gefunden werden können. Außerdem wurde zeigt, dass es Probleme beim Objekt- und Parameter-Tracking gibt, welche auf die Pointer-Analyse zurückzuführen sind. Anhand der Tests von Cipher und Signature wurde dann gezeigt, wie hilfreich die Algorithmen sind, wenn die Datenabhängigkeitsoption *NO_HEAP* verwendet wird. Zum Schluss wurde der Slicer an der OSCI-Bibliothek evaluiert. Durch zwei Beispiele konnte nochmals gezeigt werden, dass die Kombination aller Tracking-Algorithmen das Verständnis für den Slice verbessert.

Kapitel 8

Fazit

Durch diese Abschlussarbeit konnten die Vorarbeiten für diese Arbeit zusammengefasst und in einen Slicer integriert werden. Dafür wurde am Anfang die Grundlage für die Programmanalyse und das Slicing definiert. Anhand der unterschiedlichen Graphen wurden das inter- sowie das intraprozedurale-Slicing präzisiert. Außerdem wurde das WALA-Framework vorstellt.
Darauf folgte die Beschreibung der Vorarbeiten. Insbesondere wurden die aufgetretenen Problematiken, die Ausblicke, die verschwundenen- und ersetzten Funktionalitäten aufgelistet und zusammengefasst.
Im nächsten Schritt wurde die für diese Arbeit verwendete Methodik vorgestellt. Dadurch und durch die zusammengefassten Informationen aus den Vorarbeiten wurden die Anforderungen an den Slicer erstellt, welche bei der Implementierung umgesetzt worden sind.
Bei der Evaluation wurde nachgewiesen, dass der Slicer mit den zusätzlichen Tracking-Algorithmen den Slicer sehr gut unterstützt. Aber es wurde auch gezeigt, dass diese Algorithmen und daher der Slicer seine Grenzen hat. So kann nicht immer jede lokale Variable oder jedes Attribut gefunden werden, und zwar selbst dann nicht, wenn diese benutzt werden. Das Objekt- und Parameter-Tracking erzeugen *false positives*, welche wiederum aber nicht so schlimm sind, weil das Ergebnis dadurch nicht schlechter wird. Zum Schluss hat sich auch gezeigt, dass auch mit der Datenabhängigkeitsoption *NO_HEAP* die Tracking-Algorithmen gute Informationen liefern, welche Experten weiterhelfen werden, die Slice-Ergebnisse zu verstehen, und das in einer kurzen Rechenzeit.
In dieser Arbeit konnten nicht alle aufgetretenen Problematiken und die Ausblicke der Vorarbeiten evaluiert oder als Anforderungen umgesetzt werden. Es bestehen noch viele offene Fragen, dass genau an dieser Stelle die nächsten Arbeiten zu diesem Slicer ansetzen können. Auf eine genaue Auflistung wird an dieser Stelle verzichtet und auf den Abschnitt 3.6 verwiesen. Zusätzlich wäre es interessant, den Slicer in einer Studie zu evaluieren, indem Experten das Tool in ihrem alltäglichen Leben (Job) eine längere Zeit nutzen. Über diese Informationen könnte vielleicht dann der Slicer dahin gehend verbessert werden, dass seine Slice-Ergebnisse noch präziser werden. Es wäre auch möglich, dass die Einstellungsmöglichkeiten für einen Slice dann überarbeitet werden müssten.

Literaturverzeichnis

[iee, 2018] (2018). Iso/iec/ieee international standard - systems and software engineering – life cycle processes – requirements engineering. *ISO/IEC/IEEE 29148:2018(E)*, pages 1–104.

[1000sourcecodes, 2014] 1000sourcecodes (2014). Software engineering-basis path testing. https://www.1000sourcecodes.com/2012/05/software-engineering-basis-path-testing.html. Zuletzt aufgerufen am: 08.03.2021.

[Basil and Turner, 1975] Basil, V. R. and Turner, A. J. (1975). Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering*, SE-1(4):390–396.

[Boehm, 1988] Boehm, B. W. (1988). A spiral model of software development and enhancement. *Computer*, 21(5):61–72.

[Cyl, 2019] Cyl, M. (2019). Sicherheitsanalyse für android-systemdienste auf der basis von programm-slicing - entwicklung eines tools zur statischen codeanalyse von android-systemdiensten auf der grundlage des wala-frameworks. Master's thesis, Universität Bremen.

[Cytron et al., 1991] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. (1991). Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490.

[Detmers, 2016] Detmers, M. (2016). Evaluation des wala-slicers bzgl. der anwendbarkeit auf sicherheitskritische java-programme. Master's thesis, Universität Bremen.

[Ernst, 2003] Ernst, M. D. (2003). Static and dynamic analysis: Synergy and duality. In *WODA 2003: Workshop on Dynamic Analysis*, pages 24–27, Portland, OR, USA.

[Gerken, 2015] Gerken, P. (2015). Statische sicherheitsanalyse von java enterprise-anwendungen mittels program-slicing. Master's thesis, Universität Bremen.

[Graf, 2016] Graf, J. (2016). *Information Flow Control with System Dependence Graphs - Improving Modularity, Scalability and Precision for Object Oriented Languages*. PhD thesis, Karlsruher Institut für Technologie (KIT).

[Gulmann, 2014] Gulmann, M. (2014). Statische sicherheitsanalyse der android system-services. Master's thesis, Universität Bremen.

[Jain, 2014] Jain, A. (2014). Programing slicing and its applications. https://www.slideshare.net/AnkurJain89/programing-slicing-static-slice-dynamic-slice-system-dependency-graph. Zuletzt aufgerufen am: 27.02.2021.

[Kerimov, 2019] Kerimov, A. (2019). Evaluation eines auf slicing basierenden codeanalysewerkzeugs in bezug auf seine praktische anwendbarkeit im kontext der it-sicherheit.

[Koschke, 2009] Koschke, P. D. R. (2009). Vorlesung software-reengineering. http://www.informatik.uni-bremen.de/st/lehre/re09/slicing.pdf. Zuletzt aufgerufen am: 27.02.2021.

[Koschke, 2010] Koschke, P. D. R. (2010). Vorlesung software-reengineering - statische programmanalyse. https://www.informatik.uni-bremen.de/st/lehre/re10/statische_analyse.pdf. Zuletzt aufgerufen am: 08.03.2021.

[Koschke, 2018] Koschke, P. D. R. (2018). Software-reengineering: Grundlagen der softwareanalyse und -transformation. Nicht veröffentlicht.

[KoSIT, 2021] KoSIT (2021). Osci und xta. https://www.xoev.de/detail.php?gsid=bremen83.c.3355.de. Zuletzt aufgerufen am: 01.04.2021.

[Möhlmann, 2020] Möhlmann, J. (2020). Konzeption und weiterentwicklung eines java slicing-tools für sicherheitsanalysen. Master's thesis, Universität Bremen.

[MongoDB, 2020] MongoDB, I. (2020). The database for modern applications. https://www.mongodb.com/. Zuletzt aufgerufen am: 29.10.2020.

[Nguyen, 2018] Nguyen, P. P. D. H. (2018). Statische sicherheitsanalyse mit automatisierten code audits - sicherheitsanalyse von java-applikationen mit erweitertem programm (wala-)slicing. Master's thesis, Universität Bremen.

[Ottenstein and Ottenstein, 1984] Ottenstein, K. J. and Ottenstein, L. M. (1984). The program dependence graph in a software development environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, page 177–184, New York, NY, USA. Association for Computing Machinery.

[Robschink, 2005] Robschink, T. (2005). *Pfadbedingungen in Abhängigkeitsgraphen und ihre Anwendung in der Softwaresicherheitstechnik*. PhD thesis, Universität Passau.

[Royce, 1970] Royce, W. W. (1970). Managing the development of large software systems: concepts and techniques. *Proc. IEEE WESTCON, Los Angeles*, pages 1–9. Reprinted in *Proceedings* of the Ninth International Conference on Software Engineering, March 1987, pp. 328–338.

[Tichy and Padberg, 2007] Tichy, W. and Padberg, F. (2007). Empirische methodik in der softwaretechnik im allgemeinen und bei der software-visualisierung im besonderen. In Böttinger, S., Theuvsen, L., Rank, S., and Morgenstern, M., editors, *Software*

Engineering 2007 – Beiträge zu den Workshops – Fachtagung des GI-Fachbereichs Softwaretechnik, pages 211–221, Bonn. Gesellschaft für Informatik e. V.

[WALA, 2016] WALA (2016). Sourceforge documentation. https://github.com/wala/WALA/wiki/SourceForge-Documentation. Zuletzt aufgerufen am: 17.03.2021.

[WALA, 2020] WALA (2020). Wala releases. https://github.com/wala/WALA/releases. Zuletzt aufgerufen am: 01.02.2021.

[WALA, 2021a] WALA (2021a). Userguide:mappingtosourcecode. http://wala.sourceforge.net/wiki/index.php/UserGuide:MappingToSourceCode #From_Slices_to_source_line_numbers. Zuletzt aufgerufen am: 18.02.2021.

[WALA, 2021b] WALA (2021b). Userguide:slicer. http://wala.sourceforge.net/wiki/index.php/UserGuide:Slicer. Zuletzt aufgerufen am: 12.02.2021.

[WALA, 2021c] WALA (2021c). Wala slicer. https://github.com/wala/WALA/wiki/Slicer. Zuletzt aufgerufen am: 17.03.2021.

[Weiser, 1984] Weiser, M. (1984). Program slicing. *IEEE Trans. Softw. Eng.*, 10(4):352–357.

[Wikipedia, 2021] Wikipedia (2021). Static single assignment form. https://en.wikipedia.org/wiki/Static_single_assignment_form. Zuletzt aufgerufen am: 12.03.2021.

Anhang A

Anhang

A.1 Programmcode

```
1  public static void interprocedural(final PointerAnalysis<InstanceKey> pa, final Set<Statement> seedStatements,
       final Collection<Statement> sliceList,
2                  final boolean objectTracking, final boolean parameterTracking, final SliceLogger logger) {
3
4   logger.log("Interprocedural Starting");
5
6   // this should save all value numbers of variables, that may point to an watched object
7   HashMap<CGNode, Set<Integer>> aliasValueNumbers = new HashMap<>();
8
9   for (Statement s : seedStatements) {
10    if (!(s instanceof StatementWithInstructionIndex)) {
11      continue;
12    }
13
14    SSAInstruction ssaInstruction = ((StatementWithInstructionIndex) s).getInstruction();
15    if (!(ssaInstruction instanceof SSAAbstractInvokeInstruction)) {
16      continue; // Possible improvement: Add support for other SSAInstructions when needed
17    }
18
19    SSAAbstractInvokeInstruction ssaAbstractInvokeInstruction = (SSAAbstractInvokeInstruction) ssaInstruction;
20    Set<Integer> valueNumbers = new HashSet<>(); // valueNumber of value, for which we look up the aliases
21    if (objectTracking) {
22      // static method call have no receiver
23      if (!ssaAbstractInvokeInstruction.isStatic()) {
24        valueNumbers.add(ssaAbstractInvokeInstruction.getReceiver());
25      }
26    }
27
28    if (parameterTracking) {
29      int i = 0;
30      // If the method call is not static, then getUse(i) and i = 0 is always the object on which the method is
            called. But used are the parameters i>=1.
31      if (!ssaAbstractInvokeInstruction.isStatic()) {
32        i = 1;
33      }
34      for (; i < ssaAbstractInvokeInstruction.getNumberOfUses(); i++) {
35        valueNumbers.add(ssaAbstractInvokeInstruction.getUse(i));
36      }
37    }
38
39    if (valueNumbers.isEmpty()) {
40      continue;
41    }
42
43    CGNode cgNode = s.getNode();
44
45    HeapGraph<InstanceKey> heapGraph = pa.getHeapGraph();
46    for (Object heapNode : heapGraph) {
47      if (!(heapNode instanceof LocalPointerKey)) {
48        continue;
49      }
50
51      // Find LocalPointerKey for the founded valueNumbers
```

```
52        LocalPointerKey localPointerKey = (LocalPointerKey) heapNode;
53        if (localPointerKey.getNode() != cgNode || !valueNumbers.contains(localPointerKey.getValueNumber())) {
54          continue;
55        }
56
57        // LocalPointerKey points to our watched object, lets check for aliases
58        // First get all successor nodes from the LocalPointerKey
59        Iterator<Object> instanceIter = heapGraph.getSuccNodes(localPointerKey);
60        while (instanceIter.hasNext()) {
61          Object instanceKey = instanceIter.next();
62          if (!(instanceKey instanceof InstanceKey)) {
63            continue; // should not happen
64          }
65
66          // for each instance localPointerKey may points to
67          // Get all predecessor nodes from the instanceKey (successor node)
68          Iterator<Object> pointerIter = heapGraph.getPredNodes(instanceKey);
69          while (pointerIter.hasNext()) {
70            Object lpkAliasObj = pointerIter.next();
71            // for each variable, that may points on the instance
72            if (!(lpkAliasObj instanceof LocalPointerKey)) {
73              continue; // should not happen
74            }
75
76            LocalPointerKey lpkAlias = (LocalPointerKey) lpkAliasObj;
77            CGNode aliasNode = lpkAlias.getNode();
78            int aliasValue = lpkAlias.getValueNumber();
79
80            if (!(aliasValueNumbers.containsKey(aliasNode))) {
81              aliasValueNumbers.put(aliasNode, new HashSet<>());
82            }
83
84            // Safe node and aliasValue
85            aliasValueNumbers.get(aliasNode).add(aliasValue);
86          }
87        }
88      }
89    }
90
91    if (!aliasValueNumbers.isEmpty()) {
92      if (aliasValueNumbers.size() > WalaSlicer.MAX_DETAILED_LOG) {
93        logger.log("Number of aliases for Seed Statements exceeded maximum Statements to log(" + WalaSlicer.
              MAX_DETAILED_LOG + "). Therefore aliases will not be logged in detail.");
94      } else {
95        logger.log("Found following possible aliases:");
96        aliasValueNumbers.forEach((cgNode, integers) -> {
97          String valueNumbers = integers
98            .stream()
99            .map(Object::toString)
100           .collect(Collectors.joining(", "));
101         logger.log("On Method " + cgNode.getMethod().getName() +
102           " the variables with the value numbers " + valueNumbers);
103       });
104     }
105     addAliasesToSlice(aliasValueNumbers, sliceList, logger);
106     logger.log("");
107   }
108 }
```

Programmcode A.1: *interprocedural*-Methode (Quelle: Eigene Darstellung)

```
1   public static void intraprocedural(final SDG<InstanceKey> sdg, final Set<Statement> seedStatements, final
        Collection<Statement> rawSliceList,
2                   final boolean objectTracking, final boolean parameterTracking, final SliceLogger logger) {
3     logger.log("Intraprocedural Starting");
4     PointerAnalysis<InstanceKey> pa = sdg.getPointerAnalysis();
5     Collection<NormalStatement> sliceList = new HashSet<>();
6
7     // Logging is much time consuming
8
9     // Step 1: Find LocalPointerKey for each SeedStatement
10    Set<LocalPointerKey> localPointerKeysSeedStatement = new HashSet<>();
11    for (Statement seedStatement : seedStatements) {
12      if (seedStatement instanceof NormalStatement) {
13        NormalStatement normalStatement = (NormalStatement) seedStatement;
14        if(normalStatement.getInstructionIndex() == -1){
15          continue;
```

```
16        }
17        if (normalStatement.getInstruction() instanceof SSAAbstractInvokeInstruction) {
18          SSAAbstractInvokeInstruction call = (SSAAbstractInvokeInstruction) normalStatement.getInstruction();
19          if (objectTracking) {
20            if (!call.isStatic()) {
21              LocalPointerKey localPointerKey = (LocalPointerKey) pa.getHeapModel().getPointerKeyForLocal(
                    normalStatement.getNode(), call.getUse(0));
22              localPointerKeysSeedStatement.add(localPointerKey);
23  //                          logger.log("Found ObjectTracking SeedStatement LocalPointerKey: " +
    localPointerKey + " for Statement " + seedStatement);
24            }
25          }
26          if (parameterTracking) {
27            int i = 0;
28            if (!call.isStatic()) {
29              i = 1;
30            }
31            for (; i < call.getNumberOfUses(); i++) {
32              LocalPointerKey localPointerKey = (LocalPointerKey) pa.getHeapModel().getPointerKeyForLocal(
                    normalStatement.getNode(), call.getUse(i));
33              localPointerKeysSeedStatement.add(localPointerKey);
34  //                          logger.log("Found ParameterTracking SeedStatement LocalPointerKey: " +
    localPointerKey + " for Statement " + seedStatement);
35            }
36          }
37        }
38      }
39    }
40
41    if (localPointerKeysSeedStatement.isEmpty()) {
42      return;
43    }
44
45    // Step 2: Find all names with the coherent valueNumbers. Each SSAPutInstruction would be create new name
46    //         Find LocalPointerKey from Ref from the SSAInstruction where LocalPointerKey from Val/Ref equal to
        the LocalPointerKey from Step 1
47    HashMap<CGNode, TrackingInfos> nameWithValues = new HashMap<>();
48    for (CGNode n : sdg.getCallGraph()) {
49      IR ir2 = n.getIR();
50
51      if (ir2 == null) {
52        continue;
53      }
54
55      for (Iterator<SSAInstruction> it = ir2.iterateAllInstructions(); it.hasNext(); ) {
56        SSAInstruction s = it.next();
57
58        if (s instanceof SSAGetInstruction) {
59          SSAGetInstruction get = (SSAGetInstruction) s;
60          Atom atom = get.getDeclaredField().getName();
61          TrackingInfos trackingInfos = nameWithValues.get(n) != null ? nameWithValues.get(n) : new TrackingInfos
                ();
62          TrackingInfos.Info info = trackingInfos.getInfo(atom);
63          trackingInfos.addInfo(info);
64          info.addValueNumberAtLastSet(get.getDef(), get);
65          nameWithValues.put(n, trackingInfos);
66
67          LocalPointerKey pointerKey = (LocalPointerKey) pa.getHeapModel().getPointerKeyForLocal(n, get.getDef())
                ;
68          if (localPointerKeysSeedStatement.contains(pointerKey)) {
69            if (!get.isStatic()) { // ignore static, because getRef is -1
70              // ignore ref == 1, ref 1 always points to an attribute
71              if (get.getRef() != 1) {
72                LocalPointerKey localPointerKey = (LocalPointerKey) pa.getHeapModel().getPointerKeyForLocal(n,
                      get.getRef());
73  //                              if (!localPointerKeysSeedStatement.contains(localPointerKey)) {
74                localPointerKeysSeedStatement.add(localPointerKey);
75  //                                  logger.log("Found SSAGetInstruction with LocalPointerKey: " +
    localPointerKey + " for Instruction " + s);
76  //                              }
77              }
78            }
79          }
80        }
81        if (s instanceof SSAPutInstruction) {
82          SSAPutInstruction put = (SSAPutInstruction) s;
83          Atom atom = put.getDeclaredField().getName();
84          TrackingInfos trackingInfos = nameWithValues.get(n) != null ? nameWithValues.get(n) : new TrackingInfos
```

```
            ();
85          TrackingInfos.Info info = trackingInfos.getInfo(atom);
86          trackingInfos.addInfo(info);
87          info.addValueNumberWithNewSet(put.getVal(), put);
88          nameWithValues.put(n, trackingInfos);
89
90          LocalPointerKey pointerKey = (LocalPointerKey) pa.getHeapModel().getPointerKeyForLocal(n, put.getVal())
                ;
91          if (localPointerKeysSeedStatement.contains(pointerKey)) {
92            if (!put.isStatic()) { // ignore static, because getRef is -1
93              LocalPointerKey localPointerKey = (LocalPointerKey) pa.getHeapModel().getPointerKeyForLocal(n, put.
                    getRef());
94              // ignore ref == 1, ref 1 always points to an attribute
95              if (put.getRef() != 1) {
96  //                              if (!localPointerKeysSeedStatement.contains(localPointerKey)) {
97                localPointerKeysSeedStatement.add(localPointerKey);
98  //                                  logger.log("Found SSAPutInstruction with LocalPointerKey: " +
        localPointerKey + " for Instruction " + s);
99  //                              }
100             }
101           }
102         }
103       }
104     }
105   }
106
107   // Step 3: Only add Names where the valueNumbers are equal to the valueNumber from the localPointerKey
108   for (CGNode node : nameWithValues.keySet()) {
109     TrackingInfos trackingInfos = nameWithValues.get(node);
110     for (TrackingInfos.Info info : trackingInfos.getInfos()) {
111       for (Set<Pair<SSAFieldAccessInstruction, Integer>> set : info.getValueNumbers()) {
112         for (LocalPointerKey localPointerKey : localPointerKeysSeedStatement) {
113           for (Pair<SSAFieldAccessInstruction, Integer> pair : set) {
114             if (localPointerKey.getNode().equals(node) && pair.two == localPointerKey.getValueNumber()) {
115               Set<Pair<SSAFieldAccessInstruction, Integer>> sortedPairs = info.getSortedValueNumbers();
116               sortedPairs.addAll(set);
117               break;
118             }
119           }
120         }
121       }
122     }
123   }
124
125   // Step 4: Add Names where the valueNumbers are equal to the valueNumbers from Step 2
126   for (CGNode node : nameWithValues.keySet()) {
127     if (nameWithValues.get(node) == null) {
128       continue;
129     }
130     TrackingInfos trackingInfos = nameWithValues.get(node);
131     for (TrackingInfos.Info info : trackingInfos.getInfos()) {
132       for (Set<Pair<SSAFieldAccessInstruction, Integer>> set : info.getValueNumbers()) {
133         for (Pair<SSAFieldAccessInstruction, Integer> pair : set) {
134
135           for (TrackingInfos.Info sortedInfo : trackingInfos.getInfos()) {
136             Set<Pair<SSAFieldAccessInstruction, Integer>> sortedSet = sortedInfo.getSortedValueNumbers();
137             for (Pair<SSAFieldAccessInstruction, Integer> sortedPair : new HashSet<>(sortedSet)) {
138
139               if (sortedPair.two.equals(pair.two)) {
140                 sortedSet.addAll(set);
141                 break;
142               }
143             }
144           }
145         }
146       }
147     }
148   }
149
150   // Step 5: Add Statement to sliceList, when LocalPointerKey from the Instruction is equal to the found
        LocalPointerKey or the valueNumbers are equal to the founded valueNumbers
151   for (CGNode node : sdg.getCallGraph()) {
152     IR ir = node.getIR();
153
154     if (ir == null) {
155       continue;
156     }
157
```

```
158     for (Iterator<SSAInstruction> iterator = ir.iterateAllInstructions(); iterator.hasNext(); ) {
159       SSAInstruction s = iterator.next();
160       for (int i = 0; i < s.getNumberOfUses(); i++) {
161         if (s.getUse(i) > 0) {
162           LocalPointerKey pointerKey = (LocalPointerKey) pa.getHeapModel().getPointerKeyForLocal(node, s.getUse
                (i));
163           if (localPointerKeysSeedStatement.contains(pointerKey)) {
164             NormalStatement normalStatement = new NormalStatement(node, s.iIndex());
165             if (!sliceList.contains(normalStatement)) {
166               sliceList.add(normalStatement);
167             }
168           } else {
169             if (nameWithValues.get(node) != null) {
170               TrackingInfos trackingInfos = nameWithValues.get(node);
171               for (Atom atom : trackingInfos.getAtoms()) {
172                 for (Pair<SSAFieldAccessInstruction, Integer> pair : trackingInfos.getInfo(atom).
                      getSortedValueNumbers()) {
173                   if (pair.two == s.getUse(i)) {
174                     NormalStatement normalStatement = new NormalStatement(node, s.iIndex());
175                     if (!sliceList.contains(normalStatement)) {
176                       sliceList.add(normalStatement);
177                     }
178                   }
179                 }
180               }
181             }
182           }
183           if (nameWithValues.get(node) != null) {
184             for (CGNode nn : sdg.getCallGraph()) {
185               IR ir2n = nn.getIR();
186               if (ir2n == null) {
187                 continue;
188               }
189               if (nn.getMethod().isInit() || nn.getMethod().isClinit()) {
190                 TrackingInfos trackingInfos = nameWithValues.get(node);
191                 for (TrackingInfos.Info info : trackingInfos.getInfos()) {
192                   for (Pair<SSAFieldAccessInstruction, Integer> set : info.getSortedValueNumbers()) {
193                     if (nn.getMethod().getDeclaringClass().getReference().equals(set.one.getDeclaredField().
                          getDeclaringClass())) {
194                       for (Iterator<SSAInstruction> itn = ir2n.iterateAllInstructions(); itn.hasNext(); ) {
195                         SSAInstruction sn = itn.next();
196                         if (sn instanceof SSAPutInstruction) {
197
198                           if (((SSAPutInstruction) sn).getDeclaredField().getName().equals(set.one.
                                getDeclaredField().getName())) {
199                             NormalStatement normalStatement = new NormalStatement(nn, sn.iIndex());
200                             if (!sliceList.contains(normalStatement)) {
201                               sliceList.add(normalStatement);
202                             }
203                           }
204                         }
205                       }
206                     }
207                   }
208                 }
209               }
210             }
211           }
212         }
213       }
214       for (int i = 0; i < s.getNumberOfDefs(); i++) {
215         LocalPointerKey pointerKey = (LocalPointerKey) pa.getHeapModel().getPointerKeyForLocal(node, s.getDef(i
              ));
216         if (localPointerKeysSeedStatement.contains(pointerKey)) {
217           NormalStatement normalStatement = new NormalStatement(node, s.iIndex());
218           if (!sliceList.contains(normalStatement)) {
219             sliceList.add(normalStatement);
220           }
221         } else {
222           if (nameWithValues.get(node) != null) {
223             TrackingInfos trackingInfos = nameWithValues.get(node);
224             for (Atom atom : trackingInfos.getAtoms()) {
225               for (Pair<SSAFieldAccessInstruction, Integer> pair : trackingInfos.getInfo(atom).
                    getSortedValueNumbers()) {
226                 if (pair.two == s.getDef(i)) {
227                   NormalStatement normalStatement = new NormalStatement(node, s.iIndex());
228                   if (!sliceList.contains(normalStatement)) {
229                     sliceList.add(normalStatement);
```

```
                }
              }
            }
          }
        }
      }
    }
  }
  if (sliceList.size() > WalaSlicer.MAX_DETAILED_LOG) {
    logger.log("Number of Tracking Statements exceeded maximum Statements to log(" + WalaSlicer.
        MAX_DETAILED_LOG + "). ThereforeTracking Statements will not be logged in detail.");
  } else {
    for (NormalStatement n : sliceList) {
      if (n.getInstructionIndex() == -1) {
        continue;
      }
      SSAInstruction s = n.getInstruction();
      if (s instanceof SSAAbstractInvokeInstruction) {
        logger.log("Found method call: \"" + ((SSAAbstractInvokeInstruction) s).getCallSite().getDeclaredTarget
            ().getName().toString() + "\": " + s);
      } else if (s instanceof SSAFieldAccessInstruction) {
        logger.log("Found field instruction: \"" + ((SSAFieldAccessInstruction) s).getDeclaredField().getName()
            + "\": " + s);
      }
    }
  }
  rawSliceList.addAll(sliceList);
}
```

Programmcode A.2: *intraprocedural*-Methode (Quelle: Eigene Darstellung)

```
public class MethodVisitorParameterNames extends VoidVisitorAdapter<Object> {
    // in
    private Set<Integer> slicedLineNumbers;

    // out
    private Set<Integer> sourceLineNumbers;

    private Set<Pair<String, String>> parameterNames;


    public MethodVisitorParameterNames(final Set<Integer> slicedLineNumbers, final Set<Pair<String, String>>
        parameterNames) {
        super();
        this.slicedLineNumbers = slicedLineNumbers;
        // base the out-list on the in-list so all wala lines are added by default
        this.sourceLineNumbers = new HashSet<>(slicedLineNumbers);
        this.parameterNames = parameterNames;
    }


    public void addStatementBodyForLocalNames(Node node) {
        if (parameterNames == null) {
            return;
        }

        if (node instanceof BlockStmt) {
            BlockStmt blockStmt = (BlockStmt) node;
            for (Statement stmt : blockStmt.getStatements()) {
                addStatementBodyForLocalNames(stmt);
            }
        }

        if (node instanceof ForStmt) {
            ForStmt forStmt = (ForStmt) node;
            if (forStmt.getBody() instanceof BlockStmt) {
                addStatementBodyForLocalNames(forStmt.getBody());
            }
        }

        if (node instanceof ForEachStmt) {
            ForEachStmt forEachStmt = (ForEachStmt) node;
            if (forEachStmt.getBody() instanceof BlockStmt) {
                addStatementBodyForLocalNames(forEachStmt.getBody());
            }
        }
```

```
45
46          if (node instanceof SynchronizedStmt) {
47              SynchronizedStmt synchronizedStmt = (SynchronizedStmt) node;
48              if (synchronizedStmt.getBody() != null) {
49                  addStatementBodyForLocalNames(synchronizedStmt.getBody());
50              }
51          }
52
53          if (node instanceof WhileStmt) {
54              WhileStmt whileStmt = (WhileStmt) node;
55              if (whileStmt.getBody() instanceof BlockStmt) {
56                  addStatementBodyForLocalNames(whileStmt.getBody());
57              }
58          }
59
60          if (node instanceof IfStmt) {
61              IfStmt ifStmt = (IfStmt) node;
62              Statement thenStmt = ifStmt.getThenStmt();
63              if (thenStmt != null) {
64                  addStatementBodyForLocalNames(thenStmt);
65              }
66              if (ifStmt.getElseStmt() != null && ifStmt.getElseStmt().isPresent()) {
67                  Statement elseStmt = ifStmt.getElseStmt().get();
68                  addStatementBodyForLocalNames(elseStmt);
69              }
70          }
71
72          if (node instanceof TryStmt) {
73              TryStmt tryStmt = (TryStmt) node;
74              for (Node child : tryStmt.getChildNodes()) {
75                  addStatementBodyForLocalNames(child);
76              }
77
78              for (CatchClause catchClause : tryStmt.getCatchClauses()) {
79                  for (Node child : catchClause.getChildNodes()) {
80                      addStatementBodyForLocalNames(child);
81                  }
82              }
83
84              Optional<BlockStmt> finallyBlock = tryStmt.getFinallyBlock();
85              if (finallyBlock.isPresent()) {
86                  for (Node child : finallyBlock.get().getChildNodes()) {
87                      addStatementBodyForLocalNames(child);
88                  }
89              }
90          }
91
92          if (node instanceof SwitchStmt) {
93              SwitchStmt switchStmt = (SwitchStmt) node;
94              for (SwitchEntry switchEntry : switchStmt.getEntries()) {
95                  addStatementBodyForLocalNames(switchEntry);
96              }
97          }
98
99          if (node instanceof SwitchEntry) {
100             SwitchEntry switchEntry = (SwitchEntry) node;
101             for (Statement switchEntryStmt : switchEntry.getStatements()) {
102                 addStatementBodyForLocalNames(switchEntryStmt);
103             }
104         }
105
106         if (node instanceof ExpressionStmt) {
107             ExpressionStmt expressionStmt = (ExpressionStmt) node;
108             if (expressionStmt.getExpression() instanceof VariableDeclarationExpr) {
109                 VariableDeclarationExpr var = (VariableDeclarationExpr) expressionStmt.getExpression();
110                 for (VariableDeclarator vde : var.getVariables()) {
111                     addSourceLineNumbersForParameterName(vde.getNameAsString(), node, expressionStmt.getBegin()
                            .get().line, expressionStmt.getEnd().get().line);
112                 }
113             }
114             if (expressionStmt.getExpression() instanceof AssignExpr) {
115                 AssignExpr assignExpr = (AssignExpr) expressionStmt.getExpression();
116
117                 if (assignExpr.getTarget() instanceof NameExpr) {
118                     NameExpr nameExpr = (NameExpr) assignExpr.getTarget();
119                     addSourceLineNumbersForParameterName(nameExpr.getNameAsString(), node, expressionStmt.
                            getBegin().get().line, expressionStmt.getEnd().get().line);
120                 } else if (assignExpr.getTarget() instanceof FieldAccessExpr) {
```

```
121                     FieldAccessExpr fieldAccessExpr = (FieldAccessExpr) assignExpr.getTarget();
122                     addSourceLineNumbersForParameterName(fieldAccessExpr.getNameAsString(), node,
                                expressionStmt.getBegin().get().line, expressionStmt.getEnd().get().line);
123                 }
124             }
125         }
126         if (node instanceof VariableDeclarator) {
127             VariableDeclarator variableDeclarator = (VariableDeclarator) node;
128             addSourceLineNumbersForParameterName(variableDeclarator.getNameAsString(), node, node.getBegin().
                    get().line, node.getEnd().get().line);
129         }
130     }
131
132
133     private void addSourceLineNumbersForParameterName(final String parameterName, final Node node, final int
             beginLine, final int endLine) {
134         String className = getClassNameFromNode(node);
135         for (Pair<String, String> pair : parameterNames) {
136             if (pair.two.equals(parameterName) && (areSlicedLineNumbersInNode(getClassOrInterfaceDeclaration(
                    node)) || pair.one.equals(className))) {
137                 addAllLinesFromBeginToEnd(beginLine, endLine, sourceLineNumbers);
138             }
139         }
140     }
141
142
143     private String getClassNameFromNode(final Node node) {
144         return getClassOrInterfaceDeclaration(node).getNameAsString(); // should never happen
145     }
146
147
148     public ClassOrInterfaceDeclaration getClassOrInterfaceDeclaration(final Node node) {
149         if (node instanceof ClassOrInterfaceDeclaration) {
150             return ((ClassOrInterfaceDeclaration) node);
151         }
152         if (node.getParentNode().isPresent()) {
153             Node parentNode = node.getParentNode().get();
154             while (!(parentNode instanceof ClassOrInterfaceDeclaration)) {
155                 if (parentNode.getParentNode().isPresent()) {
156                     parentNode = parentNode.getParentNode().get();
157                 } else {
158                     parentNode = null;
159                     break;
160                 }
161             }
162             if (parentNode != null) {
163                 return (ClassOrInterfaceDeclaration) parentNode;
164             }
165         }
166         return null; // should never happen
167     }
168
169
170     @Override
171     public void visit(final InitializerDeclaration initializerDeclaration, final Object arg) {
172         for (Node node : initializerDeclaration.getBody().getChildNodes()) {
173             addStatementBodyForLocalNames(node);
174         }
175     }
176
177
178     @Override
179     public void visit(FieldDeclaration fieldDeclaration, Object arg) {
180         super.visit(fieldDeclaration, arg);
181         for (VariableDeclarator vde : fieldDeclaration.getVariables()) {
182             addStatementBodyForLocalNames(vde);
183         }
184     }
185
186
187     @Override
188     public void visit(MethodDeclaration methodDeclaration, Object arg) {
189         if (areSlicedLineNumbersInNode(methodDeclaration)) {
190             if (methodDeclaration.getBody().isPresent()) {
191                 for (Node node : methodDeclaration.getBody().get().getChildNodes()) {
192                     addStatementBodyForLocalNames(node);
193                 }
194             }
```

```
195         }
196     }
197
198
199     @Override
200     public void visit(ConstructorDeclaration constructorDeclaration, Object arg) {
201         for (Node node : constructorDeclaration.getBody().getChildNodes()) {
202             addStatementBodyForLocalNames(node);
203         }
204     }
205
206
207     public Set<Integer> getSlice() {
208         return sourceLineNumbers;
209     }
210
211
212     private boolean isLineInNode(Node node, int line) {
213         return (node.getBegin().get().line <= line && node.getEnd().get().line >= line);
214     }
215
216
217     private boolean areSlicedLineNumbersInNode(Node node) {
218         for (Integer line : slicedLineNumbers) {
219             if (isLineInNode(node, line)) {
220                 return true;
221             }
222         }
223         return false;
224     }
225
226
227     // --------------------- Note -----------------------------------
228     // rangeClosed(int startInclusive, int endInclusive):
229     // Returns a sequential ordered IntStream from startInclusive (inclusive)
230     // to endInclusive (inclusive) by an incremental step of 1.
231     // (see: https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html)
232     private void addAllLinesFromBeginToEnd(int start, int end, Set<Integer> collection) {
233         collection.addAll(IntStream.rangeClosed(start, end).boxed().collect(Collectors.toList()));
234     }
235 }
```

Programmcode A.3: *MethodVisitorParameterNames*-Klasse (Quelle: Eigene Darstellung)

```
1  /**
2   * This is an implementation of the of the MethodVisitor-pattern for the JavaParser described in their ebook.
3   * The code based on the work by Markus Gulman (Masterthesis 2014) and Philip Phu Dang Hoan Nguyen (
        Masterthesis 2018) but has been
4   * heavily altered by Michael Cyl with bug fixed, improvements and refactorings. Most notable changes are the
        update from Version 1.5
5   * to 3.14.9 along with the fix of all breaking changes and the addition of missing statement types like
        SynchronizedStmt, ExpressionStmt,
6   * FinallyBlocks and changes to handling of special statements like return statements.
7   */
8  public class MethodVisitorSourceLines extends VoidVisitorAdapter<Object> {
9      // in
10     private Set<Integer> slicedLineNumbers;
11
12     // out
13     private Set<Integer> sourceLineNumbers;
14
15     // use this to always take the class declaration into the slice
16     private String mainClassName;
17
18
19     public MethodVisitorSourceLines(final Set<Integer> slicedLineNumbers, String mainClassName) {
20         super();
21         this.slicedLineNumbers = slicedLineNumbers;
22         // base the out-list on the in-list so all wala lines are added by default
23         this.sourceLineNumbers = new HashSet<>(slicedLineNumbers);
24         this.mainClassName = mainClassName;
25     }
26
27
28     // add all kinds of ast statements
29     // see:
```

```
30      // https://static.javadoc.io/com.github.javaparser/javaparser-core/3.5.0/com/github/javaparser/ast/stmt/
            Statement.html
31      public void addStatementBodyForLineNumbers(Node node, int line) {
32
33          if (node instanceof BlockStmt) {
34              BlockStmt blockStmt = (BlockStmt) node;
35              for (Statement stmt : blockStmt.getStatements()) {
36                  addStatementBodyForLineNumbers(stmt, line);
37              }
38              return;
39          }
40
41          if (isLineInNode(node, line)) {
42
43              if (node instanceof ForStmt) {
44                  ForStmt forStmt = (ForStmt) node;
45                  // mcyl: fix for multiline heads
46                  addAllLinesFromBeginToEnd(forStmt.getBegin().get().line,
47                      forStmt.getBody().getBegin().get().line,
48                      sourceLineNumbers);
49                  // add closing line
50                  sourceLineNumbers.add(forStmt.getBody().getEnd().get().line);
51
52                  // search inner statements
53                  if (forStmt.getBody() instanceof BlockStmt) {
54                      addStatementBodyForLineNumbers(forStmt.getBody(), line);
55                  }
56              }
57
58              // mcyl: added ForEachStmt
59              if (node instanceof ForEachStmt) {
60                  ForEachStmt forEachStmt = (ForEachStmt) node;
61                  // mcyl: fix for multiline heads
62                  addAllLinesFromBeginToEnd(
63                      forEachStmt.getBegin().get().line,
64                      forEachStmt.getBody().getBegin().get().line,
65                      sourceLineNumbers);
66                  sourceLineNumbers.add(forEachStmt.getBody().getEnd().get().line);
67
68                  if (forEachStmt.getBody() instanceof BlockStmt) {
69                      addStatementBodyForLineNumbers(forEachStmt.getBody(), line);
70                  }
71              }
72
73              // mcyl: added SynchronizedStmt
74              if (node instanceof SynchronizedStmt) {
75                  SynchronizedStmt synchronizedStmt = (SynchronizedStmt) node;
76                  // mcyl: fix for multiline heads
77                  addAllLinesFromBeginToEnd(
78                      synchronizedStmt.getBegin().get().line,
79                      synchronizedStmt.getBody().getBegin().get().line,
80                      sourceLineNumbers);
81                  sourceLineNumbers.add(synchronizedStmt.getBody().getEnd().get().line);
82
83                  if (synchronizedStmt.getBody() != null) {
84                      addStatementBodyForLineNumbers(synchronizedStmt.getBody(), line);
85                  }
86              }
87
88              if (node instanceof WhileStmt) {
89                  WhileStmt whileStmt = (WhileStmt) node;
90                  // mcyl: fix for multiline heads
91                  addAllLinesFromBeginToEnd(
92                      whileStmt.getBegin().get().line,
93                      whileStmt.getBody().getBegin().get().line,
94                      sourceLineNumbers);
95                  sourceLineNumbers.add(whileStmt.getBody().getEnd().get().line);
96
97                  if (whileStmt.getBody() instanceof BlockStmt) {
98                      addStatementBodyForLineNumbers(whileStmt.getBody(), line);
99                  }
100             }
101
102             if (node instanceof IfStmt) {
103                 IfStmt ifStmt = (IfStmt) node;
104                 Statement thenStmt = ifStmt.getThenStmt();
105
106                 int ifBegin = ifStmt.getBegin().get().line;
```

```
107                 int thenBegin = thenStmt.getBegin().get().line;
108                 int thenEnd = thenStmt.getEnd().get().line;
109 
110                 sourceLineNumbers.add(ifBegin);
111                 if (ifBegin != thenBegin - 1) {
112                     addAllLinesFromBeginToEnd(
113                         ifBegin,
114                         thenBegin - 1,
115                         sourceLineNumbers);
116                 }
117 
118                 if (thenStmt.toString().startsWith("{")) {
119                     sourceLineNumbers.add(thenBegin);
120                     sourceLineNumbers.add(thenEnd);
121                 }
122                 addStatementBodyForLineNumbers(thenStmt, line);
123 
124                 if (ifStmt.getElseStmt() != null && ifStmt.getElseStmt().isPresent()) {
125                     Statement elseStmt = ifStmt.getElseStmt().get();
126                     int elseBegin = elseStmt.getBegin().get().line;
127                     int elseEnd = elseStmt.getEnd().get().line;
128                     if (areSlicedLineNumbersInNode(elseStmt) || thenEnd == elseBegin) {
129 
130                         if (elseStmt.toString().startsWith("{")) {
131                             sourceLineNumbers.add(elseBegin);
132                             sourceLineNumbers.add(elseEnd);
133                         }
134 
135                         for (JavaToken javaToken : ifStmt.getTokenRange().get()) {
136                             if (javaToken.getText().equals("else")) {
137                                 sourceLineNumbers.add(javaToken.getRange().get().begin.line);
138                                 break;
139                             }
140                         }
141                         addStatementBodyForLineNumbers(elseStmt, line);
142                     }
143                 }
144             }
145 
146             if (node instanceof TryStmt) {
147                 TryStmt tryStmt = (TryStmt) node;
148                 if (isLineInNode(tryStmt.getTryBlock(), line)) {
149                     addStatementBodyForLineNumbers(tryStmt.getTryBlock(), line);
150                 }
151 
152                 sourceLineNumbers.add(tryStmt.getBegin().get().line);
153                 sourceLineNumbers.add(tryStmt.getEnd().get().line);
154 
155                 sourceLineNumbers.add(tryStmt.getTryBlock().getBegin().get().line);
156                 sourceLineNumbers.add(tryStmt.getTryBlock().getEnd().get().line);
157                 int lastLineForFinally = tryStmt.getTryBlock().getEnd().get().line;
158 
159                 for (CatchClause catchClause : tryStmt.getCatchClauses()) {
160                     sourceLineNumbers.add(catchClause.getBegin().get().line);
161                     sourceLineNumbers.add(catchClause.getEnd().get().line);
162                     lastLineForFinally = catchClause.getEnd().get().line;
163 
164                     sourceLineNumbers.add(catchClause.getBody().getBegin().get().line);
165                     sourceLineNumbers.add(catchClause.getBody().getEnd().get().line);
166 
167                     addAllLinesFromBeginToEnd(
168                         catchClause.getBegin().get().line,
169                         catchClause.getBody().getBegin().get().line,
170                         sourceLineNumbers);
171 
172                     if (isLineInNode(catchClause, line)) {
173                         for (Node child : catchClause.getChildNodes()) {
174                             addStatementBodyForLineNumbers(child, line);
175                         }
176                     }
177                 }
178 
179                 Optional<BlockStmt> finallyBlock = tryStmt.getFinallyBlock();
180                 if (finallyBlock.isPresent()) {
181                     sourceLineNumbers.add(finallyBlock.get().getBegin().get().line);
182                     sourceLineNumbers.add(finallyBlock.get().getEnd().get().line);
183 
184                     addAllLinesFromBeginToEnd(
```

```
185                         lastLineForFinally,
186                         finallyBlock.get().getBegin().get().line,
187                         sourceLineNumbers);
188
189                     if (isLineInNode(finallyBlock.get(), line)) {
190                         addStatementBodyForLineNumbers(finallyBlock.get(), line);
191                     }
192                 }
193             }
194
195             // mcyl: fix for multiline expression statements
196             if (node instanceof ExpressionStmt) {
197                 ExpressionStmt expressionStmt = (ExpressionStmt) node;
198
199                 addAllLinesFromBeginToEnd(
200                     expressionStmt.getBegin().get().line,
201                     expressionStmt.getEnd().get().line,
202                     sourceLineNumbers);
203             }
204
205             // mcyl: added switch statements
206             if (node instanceof SwitchStmt) {
207                 SwitchStmt switchStmt = (SwitchStmt) node;
208                 // mcyl: fix for multiline heads
209                 int entrySize = switchStmt.getEntries().size();
210                 if (entrySize > 0) {
211                     addAllLinesFromBeginToEnd(
212                         switchStmt.getBegin().get().line,
213                         switchStmt.getEntry(0).getBegin().get().line,
214                         sourceLineNumbers);
215                     sourceLineNumbers.add(switchStmt.getEntry(entrySize - 1).getEnd().get().line);
216
217                     for (SwitchEntry switchEntry : switchStmt.getEntries()) {
218                         addStatementBodyForLineNumbers(switchEntry, line);
219                     }
220                 }
221             }
222
223             // mcyl: added switch entry statements
224             if (node instanceof SwitchEntry) {
225                 SwitchEntry switchEntry = (SwitchEntry) node;
226
227                 addAllLinesFromBeginToEnd(
228                     switchEntry.getBegin().get().line,
229                     switchEntry.getEnd().get().line,
230                     sourceLineNumbers);
231
232                 for (Statement switchEntryStmt : switchEntry.getStatements()) {
233                     addStatementBodyForLineNumbers(switchEntryStmt, line);
234                 }
235             }
236         }
237
238         // mcyl: add all return-statements, regardless if sliced line is inside this return-statement node
239         if (node instanceof ReturnStmt) {
240             addAllLinesFromBeginToEnd(
241                 node.getBegin().get().line,
242                 node.getEnd().get().line,
243                 sourceLineNumbers);
244         }
245
246         // mcyl: add all continue- and break-statements
247         if (node instanceof ContinueStmt | node instanceof BreakStmt) {
248             sourceLineNumbers.add(node.getBegin().get().line);
249         }
250
251         // mcyl: fix for multiline throw statements + always add throw statements
252         if (node instanceof ThrowStmt) {
253             ThrowStmt throwStmt = (ThrowStmt) node;
254             addAllLinesFromBeginToEnd(
255                 throwStmt.getBegin().get().line,
256                 throwStmt.getEnd().get().line,
257                 sourceLineNumbers);
258         }
259     }
260
261
262     @Override
```

```
263     public void visit(final InitializerDeclaration initializerDeclaration, final Object arg) {
264         if (areSlicedLineNumbersInNode(initializerDeclaration)) {
265             Node parentNode = initializerDeclaration.getParentNode().get();
266             if (parentNode instanceof ClassOrInterfaceDeclaration) {
267                 setClassBody((ClassOrInterfaceDeclaration) parentNode);
268             }
269         }
270 
271         // search inside of constructor if slice node line is inside or else return
272         if (!areSlicedLineNumbersInNode(initializerDeclaration)) {
273             return;
274         }
275 
276         // Add all lines between method and first brackets
277         // (Fix for multiple line method heads)
278         addAllLinesFromBeginToEnd(
279             initializerDeclaration.getBegin().get().line,
280             initializerDeclaration.getBody().getBegin().get().line,
281             sourceLineNumbers);
282         sourceLineNumbers.add(initializerDeclaration.getBody().getEnd().get().line);
283         sourceLineNumbers.add(initializerDeclaration.getEnd().get().line);
284 
285         for (Node node : initializerDeclaration.getBody().getChildNodes()) {
286             for (int line : slicedLineNumbers) {
287                 if (isLineInNode(node, line)) {
288                     addStatementBodyForLineNumbers(node, line);
289                 }
290             }
291         }
292     }
293 
294 
295     @Override
296     public void visit(VariableDeclarator variableDeclarator, Object arg) {
297         super.visit(variableDeclarator, arg);
298         if (!areSlicedLineNumbersInNode(variableDeclarator)) {
299             return;
300         }
301         sourceLineNumbers.add(variableDeclarator.getBegin().get().line);
302         sourceLineNumbers.add(variableDeclarator.getEnd().get().line);
303         if (variableDeclarator.getParentNode().get() instanceof ClassOrInterfaceDeclaration) {
304             setClassBody((ClassOrInterfaceDeclaration) variableDeclarator.getParentNode().get());
305         }
306     }
307 
308 
309     @Override
310     public void visit(FieldDeclaration fieldDeclaration, Object arg) {
311         super.visit(fieldDeclaration, arg);
312         if (!areSlicedLineNumbersInNode(fieldDeclaration)) {
313             return;
314         }
315         sourceLineNumbers.add(fieldDeclaration.getBegin().get().line);
316         sourceLineNumbers.add(fieldDeclaration.getEnd().get().line);
317         if (fieldDeclaration.getParentNode().get() instanceof ClassOrInterfaceDeclaration) {
318             setClassBody((ClassOrInterfaceDeclaration) fieldDeclaration.getParentNode().get());
319         }
320     }
321 
322 
323     @Override
324     public void visit(MethodDeclaration methodDeclaration, Object arg) {
325         if (!areSlicedLineNumbersInNode(methodDeclaration)) {
326             return;
327         }
328 
329         Node parentNode = methodDeclaration.getParentNode().get();
330         // pnguyen: add all lines between class and first node (fix for brackets in next line)
331         if (parentNode instanceof ClassOrInterfaceDeclaration) {
332             setClassBody((ClassOrInterfaceDeclaration) parentNode);
333         }
334 
335         sourceLineNumbers.add(parentNode.getEnd().get().line);
336 
337         // Add all lines between method and first brackets
338         // (Fix for multiple line method heads)
339         addAllLinesFromBeginToEnd(
340             methodDeclaration.getBegin().get().line,
```

```
341             methodDeclaration.getBody().get().getBegin().get().line,
342             sourceLineNumbers);
343         sourceLineNumbers.add(methodDeclaration.getBody().get().getEnd().get().line);
344         sourceLineNumbers.add(methodDeclaration.getEnd().get().line);
345 
346         for (Node node : methodDeclaration.getBody().get().getChildNodes()) {
347             for (int line : slicedLineNumbers) {
348                 if (isLineInNode(node, line)) {
349                     addStatementBodyForLineNumbers(node, line);
350                 }
351             }
352         }
353     }
354 
355 
356     // pnguyen: ConstructorDeclaration was ignored and led to wrong reconstructed code
357     @Override
358     public void visit(ConstructorDeclaration constructorDeclaration, Object arg) {
359         // mcyl: set class body if slice line is inside note or if its the main class
360         if (areSlicedLineNumbersInNode(constructorDeclaration) ||
361             constructorDeclaration.getNameAsString().equals(this.mainClassName)) {
362 
363             Node parentNode = constructorDeclaration.getParentNode().get();
364             if (parentNode instanceof ClassOrInterfaceDeclaration) {
365                 setClassBody((ClassOrInterfaceDeclaration) parentNode);
366             }
367         }
368 
369         // search inside of constructor if slice node line is inside or else return
370         if (!areSlicedLineNumbersInNode(constructorDeclaration)) {
371             return;
372         }
373 
374         // Add all lines between method and first brackets
375         // (Fix for multiple line method heads)
376         addAllLinesFromBeginToEnd(
377             constructorDeclaration.getBegin().get().line,
378             constructorDeclaration.getBody().getBegin().get().line,
379             sourceLineNumbers);
380         sourceLineNumbers.add(constructorDeclaration.getBody().getEnd().get().line);
381         sourceLineNumbers.add(constructorDeclaration.getEnd().get().line);
382 
383         for (Node node : constructorDeclaration.getBody().getChildNodes()) {
384             for (int line : slicedLineNumbers) {
385                 if (isLineInNode(node, line)) {
386                     addStatementBodyForLineNumbers(node, line);
387                 }
388             }
389         }
390     }
391 
392 
393     public Set<Integer> getSlice() {
394         return sourceLineNumbers;
395     }
396 
397 
398     private boolean isLineInNode(Node node, int line) {
399         return (node.getBegin().get().line <= line && node.getEnd().get().line >= line);
400     }
401 
402 
403     private boolean areSlicedLineNumbersInNode(Node node) {
404         for (Integer line : slicedLineNumbers) {
405             if (isLineInNode(node, line)) {
406                 return true;
407             }
408         }
409         return false;
410     }
411 
412 
413     /**
414      * This method adds the class declartion, any extended ore implemented types and imports
415      */
416     private void setClassBody(ClassOrInterfaceDeclaration classNode) {
417         int endLine = classNode.getEnd().get().line;
418         for (JavaToken javaToken : classNode.getTokenRange().get()) {
```

```
419             if (javaToken.getText().equals("{")) {
420                 endLine = javaToken.getRange().get().end.line;
421                 break;
422             }
423         }
424         addAllLinesFromBeginToEnd(classNode.getBegin().get().line, endLine, sourceLineNumbers);
425         addAllLinesFromBeginToEnd(classNode.getEnd().get().line, classNode.getEnd().get().line,
                sourceLineNumbers);
426
427         if (classNode.getParentNode().get() instanceof CompilationUnit) {
428             CompilationUnit compilationUnit = (CompilationUnit) classNode.getParentNode().get();
429             for (ImportDeclaration importDeclaration : compilationUnit.getImports()) {
430                 addAllLinesFromBeginToEnd(importDeclaration.getRange().get().begin.line, importDeclaration.
                        getRange().get().end.line, sourceLineNumbers);
431             }
432             if (compilationUnit.getPackageDeclaration().isPresent()) {
433                 PackageDeclaration packageDeclaration = compilationUnit.getPackageDeclaration().get();
434                 addAllLinesFromBeginToEnd(packageDeclaration.getRange().get().begin.line, packageDeclaration.
                        getRange().get().end.line, sourceLineNumbers);
435             }
436         }
437
438         if (classNode.getParentNode().get() instanceof ClassOrInterfaceDeclaration) {
439             setClassBody((ClassOrInterfaceDeclaration) classNode.getParentNode().get());
440         }
441     }
442
443
444     // --------------------- Note -----------------------------------
445     // rangeClosed(int startInclusive, int endInclusive):
446     // Returns a sequential ordered IntStream from startInclusive (inclusive)
447     // to endInclusive (inclusive) by an incremental step of 1.
448     // (see: https://docs.oracle.com/javase/8/docs/api/java/util/stream/IntStream.html)
449     private void addAllLinesFromBeginToEnd(int start, int end, Set<Integer> collection) {
450         collection.addAll(IntStream.rangeClosed(start, end).boxed().collect(Collectors.toList()));
451     }
452 }
```

Programmcode A.4: *MethodVisitorSourceLines*-Klasse (Quelle: Eigene Darstellung)

```
1  package objectTracking.cipher;
2
3  import java.io.*;
4  import java.nio.charset.Charset;
5  import java.nio.charset.StandardCharsets;
6  import java.security.*;
7  import java.security.cert.CertificateException;
8  import java.util.Base64;
9  import javax.crypto.*;
10 import javax.crypto.spec.IvParameterSpec;
11 import javax.crypto.spec.SecretKeySpec;
12
13
14 public class AllInOneClass {
15
16     private static String keyStoreName;
17     private String trans = "AES/CBC/PKCS5Padding";
18     private KeyStore keyStore;
19     private KeyPair keyPair;
20     private IvParameterSpec ivParameterSpec;
21     private String secretKeyAlias;
22     private String secretKeyPassword;
23
24     private class ShortString {
25         public Charset charset = StandardCharsets.UTF_8;
26     }
27
28     static {
29         keyStoreName = "keystore.ks";
30     }
31
32     public static void main(String[] args) throws CertificateException, NoSuchAlgorithmException,
           KeyStoreException, IOException, InvalidKeyException, BadPaddingException,
           InvalidAlgorithmParameterException, IllegalBlockSizeException, NoSuchPaddingException,
           UnrecoverableKeyException {
33         AllInOneClass allInOneClass = new AllInOneClass("hallo", "1234");
34         String plainText = "Hallo";
```

```
35          String encrypt = allInOneClass.encrypt(plainText);
36          String decrypt = allInOneClass.decrypt(encrypt);
37          if(plainText.equals(decrypt)) {
38              System.out.println("Equals!!!");
39          }
40      }
41
42      public AllInOneClass(final String secretKeyAlias, final String secretKeyPassword) throws
            CertificateException, NoSuchAlgorithmException, KeyStoreException, IOException {
43          keyStore = getKeyStore();
44          ivParameterSpec  = getIvParameterSpec();
45          this.secretKeyAlias = secretKeyAlias;
46          this.secretKeyPassword = secretKeyPassword;
47      }
48
49      public String encrypt(final String message) throws NoSuchAlgorithmException, IllegalBlockSizeException,
            InvalidKeyException, BadPaddingException, InvalidAlgorithmParameterException, NoSuchPaddingException,
            KeyStoreException {
50          SecretKey secretKey = getSecretKey();
51
52          KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
53          keyPair = keyPairGenerator.generateKeyPair();
54
55          SecretKey encryptSecretKey = encryptSecretKey(secretKey, keyPair.getPublic(), "RSA");
56
57          saveSecretKey(encryptSecretKey, secretKeyAlias, secretKeyPassword);
58
59          String encryptMessage = encryptMessage(message, secretKey, trans, ivParameterSpec);
60
61          String hallo = "hallo";
62          print(hallo);
63
64          return encryptMessage;
65      }
66
67      public String decrypt(final String encryptMessage) throws UnrecoverableKeyException,
            NoSuchAlgorithmException, KeyStoreException, IllegalBlockSizeException, InvalidKeyException,
            BadPaddingException, NoSuchPaddingException, InvalidAlgorithmParameterException {
68          SecretKey encryptSecretKey = loadSecretKey(secretKeyAlias, secretKeyPassword);
69
70          SecretKey secretKey = decryptSecretKey(encryptSecretKey, keyPair.getPrivate(), "RSA");
71
72          return decryptMessage(encryptMessage, secretKey, trans, ivParameterSpec);
73      }
74
75      private void print(String hallo) throws RuntimeException {
76          System.out.println(hallo);
77      }
78
79      private SecretKey getSecretKey() throws NoSuchAlgorithmException {
80          String algorithmName = "AES";
81          KeyGenerator keyGenerator = KeyGenerator.getInstance(algorithmName);
82          SecureRandom secureRandom = new SecureRandom();
83          int keyBitSize = 256;
84          keyGenerator.init(keyBitSize, secureRandom);
85          return keyGenerator.generateKey();
86      }
87
88      private IvParameterSpec getIvParameterSpec() throws NoSuchAlgorithmException {
89          SecureRandom randomSecureRandom = SecureRandom.getInstance("SHA1PRNG");
90          byte[] iv = new byte[16];
91          randomSecureRandom.nextBytes(iv);
92          return new IvParameterSpec(iv);
93      }
94
95      private String encryptMessage(final String message, final SecretKey secretKey, final String transformation,
              final IvParameterSpec ivParameterSpec)
96          throws NoSuchAlgorithmException, NoSuchPaddingException, InvalidKeyException, BadPaddingException,
                IllegalBlockSizeException, InvalidAlgorithmParameterException {
97          Cipher cipher = Cipher.getInstance(transformation);
98          cipher.init(Cipher.ENCRYPT_MODE, secretKey, ivParameterSpec);
99          byte[] bytes = cipher.doFinal(message.getBytes(new ShortString().charset));
100         return Base64.getEncoder().encodeToString(bytes);
101     }
102
103     private String decryptMessage(final String message, final SecretKey secretKey, final String transformation,
              final IvParameterSpec ivParameterSpec)
104         throws NoSuchAlgorithmException, NoSuchPaddingException, InvalidKeyException, BadPaddingException,
```

```
            IllegalBlockSizeException, InvalidAlgorithmParameterException {
105         Cipher cipher = Cipher.getInstance(transformation);
106         cipher.init(Cipher.DECRYPT_MODE, secretKey, ivParameterSpec);
107         byte[] bytes = cipher.doFinal(Base64.getDecoder().decode(message));
108         return new String(bytes);
109     }
110
111     private KeyStore getKeyStore() throws KeyStoreException, IOException, CertificateException,
              NoSuchAlgorithmException {
112         KeyStore keyStore = KeyStore.getInstance("PKCS12");
113         char[] keyStorePassword = "123abc".toCharArray();
114
115         if (!new File(keyStoreName).exists()) {
116             try (FileOutputStream fileOutputStream = new FileOutputStream(keyStoreName)) {
117                 keyStore.load(null, keyStorePassword);
118                 keyStore.store(fileOutputStream, keyStorePassword);
119             }
120         } else {
121             try (InputStream inputStream = new FileInputStream(keyStoreName)) {
122                 keyStore.load(inputStream, keyStorePassword);
123             }
124         }
125         return keyStore;
126     }
127
128     private SecretKey encryptSecretKey(final SecretKey secretKey, final PublicKey publicKey, final String
              transformation)
129         throws NoSuchAlgorithmException, NoSuchPaddingException, InvalidKeyException, BadPaddingException,
                IllegalBlockSizeException {
130         Cipher cipher = Cipher.getInstance(transformation);
131         cipher.init(Cipher.ENCRYPT_MODE, publicKey);
132         byte[] secretKeyAsBytes = cipher.doFinal(secretKey.getEncoded());
133         return new SecretKeySpec(secretKeyAsBytes, 0, secretKeyAsBytes.length, "AES");
134     }
135
136     private SecretKey decryptSecretKey(final SecretKey secretKey, final PrivateKey privateKey, final String
              transformation)
137         throws NoSuchAlgorithmException, NoSuchPaddingException, InvalidKeyException, BadPaddingException,
                IllegalBlockSizeException {
138         Cipher cipher = Cipher.getInstance(transformation);
139         cipher.init(Cipher.DECRYPT_MODE, privateKey);
140         byte[] secretKeyAsBytes = cipher.doFinal(secretKey.getEncoded());
141         return new SecretKeySpec(secretKeyAsBytes, 0, secretKeyAsBytes.length, "AES");
142     }
143
144     private void saveSecretKey(final SecretKey secretKey, final String alias, final String password) throws
              KeyStoreException {
145         KeyStore.SecretKeyEntry secretKeyEntry = new KeyStore.SecretKeyEntry(secretKey);
146         KeyStore.ProtectionParameter protectionParameter = new KeyStore.PasswordProtection(password.toCharArray
                ());
147         keyStore.setEntry(alias, secretKeyEntry, protectionParameter);
148     }
149
150     private SecretKey loadSecretKey(final String alias, final String password) throws NoSuchAlgorithmException,
               KeyStoreException, UnrecoverableKeyException {
151         return (SecretKey) keyStore.getKey(alias, password.toCharArray());
152     }
153 }
```

Programmcode A.5: *Cipher AllInOneClass*-Klasse (Quelle: Eigene Darstellung)