Universität Bremen

Masterarbeit

Titel der Arbeit

Erweiterung und Evaluierung der Nutzbarkeit eines Slicers für die Software-Sicherheit mithilfe von Fallstudien

---

Akademischer Abschlussgrad: Grad, Fachrichtung (Abkürzung)
Master of Science (M. Sc.)

Autorenname, Matrikelnummer
Till Schlechtweg, [redacted]

Email-Adresse
schlecht@uni-bremen.de

Studiengang
Informatik

Fachbereich
Fachbereich 3 - Mathematik/Informatik

Erstprüfer
Dr. Karsten Sohr

Zweitprüferin
Prof. Dr. Ute Bormann

Betreuer
Dr. Karsten Sohr

Abgabedatum
13. Januar 2023

Universität Bremen

Eigenständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Teile meiner Arbeit, die wortwörtlich oder dem Sinn nach anderen Werken entnommen sind, wurden unter Angabe der Quelle kenntlich gemacht. Gleiches gilt auch für Zeichnungen, Skizzen, bildliche Darstellungen sowie für Quellen aus dem Internet. Die Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht. Die elektronische Fassung der Arbeit stimmt mit der gedruckten Version überein. Mir ist bewusst, dass wahrheitswidrige Angaben als Täuschung behandelt werden.

Erklärung zur Veröffentlichung von Bachelor- oder Masterarbeiten

Die Abschlussarbeit wird zwei Jahre nach Studienabschluss dem Archiv der Universität Bremen zur dauerhaften Archivierung angeboten. Archiviert werden:

1. Masterarbeiten mit lokalem oder regionalem Bezug sowie pro Studienfach und Studienjahr 10
2. Bachelorarbeiten des jeweils ersten und letzten Bachelorabschlusses pro Studienfach u. Jahr.

Ich bin damit einverstanden, dass meine Abschlussarbeit im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

Einverständniserklärung über die Bereitstellung und Nutzung der Masterarbeit in elektronischer Form zur Überprüfung durch Plagiatssoftware

Eingereichte Arbeiten können mit der Software Plagscan auf einen hauseigenen Server auf Übereinstimmung mit externen Quellen und der institutionseigenen Datenbank untersucht werden. Zum Zweck des Abgleichs mit zukünftig zu überprüfenden Studien- und Prüfungsarbeiten kann die Arbeit dauerhaft in der institutionseigenen Datenbank der Universität Bremen gespeichert werden.
Ich bin damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum Zweck der Überprüfung auf Plagiate auf den Plagscan-Server der Universität Bremen hochgeladen wird.

Mit meiner Unterschrift versichere ich, dass ich die oben stehenden Erklärungen gelesen und verstanden habe. Mit meiner Unterschrift bestätige ich die Richtigkeit der oben gemachten Angaben.

13. Januar 2023,

Datum, Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

1 Einleitung & Motivation

Log4Shell wurde die Lücke genannt, die es einem Angreifer erlaubt, seinen eigenen Programmcode auf dem System auszuführen, welches die Bibliothek Log4j verwendet [Zha]. Sie zeigt, wie wichtig die Vermeidung von Sicherheitslücken und die richtige sowie gewollte Verwendung von Bibliotheken sind. Nach der Veröffentlichung der Sicherheitslücke hat das Bundesamt für Sicherheit in der Informationstechnik (*BSI*) die höchste Cybersicherheitswarnstufe ausgerufen.

Die Verbreitung informationstechnischer Systeme in allen erdenklichen Lebensbereichen macht die Informationssicherheit der in diesen Systemen eingesetzten Software wichtiger denn je. Bei der Verbreitung spielt auch das Internet of Things (*IoT*) eine große Rolle. Im Jahr 2019 beherbergte das IoT etwa 26 Milliarden Geräte, das sind dreimal mehr Geräte als Menschen auf der Erde zum gleichen Zeitpunkt [Tra, Uni]. Zusätzlich werden diese Systeme in immer kritischeren Bereichen eingesetzt und gleichzeitig steigt ihre Komplexität durch Vernetzung sowie steigender Aufgabenanzahl.

Als Beispiel hierfür soll der Linux-Kernel dienen. Über 80% aller Webserver, ein Großteil der eigentlichen Netzinfrastrukturgeräte und Geräte mit dem Android-Betriebssystem verwenden als Grundlage den Linux-Kernel [W3T22]. Mit der Verbreitung steigen auch die Aufgaben, daher ist es keine Überraschung, dass aus den ca. 73.000 Zeilen Quelltext der Version 1.0 aus 1994 des Linux-Kernels nun, in der Version 6.1 aus 2022, ca. 16.000.000 Zeilen Quelltext geworden sind[1].

Die Sicherheit der angesprochenen Systeme ist auch von öffentlichem Interesse. Aus diesem Grund hat das BSI 2011 das nationale Cyber-Abwehrzentrum (*CyberAZ*) aufgebaut. Das CyberAZ soll die Bundesrepublik, ihre Wirtschaft sowie ihre inneren Angelegenheiten vor elektronischen Angriffen schützen. Das BSI übernimmt Aufgaben beim Schutz der Bürgerinnen und Bürger, der Aufrechterhaltung kritischer Infrastruktur und verbreitet den aktuellen Kenntnisstand von Sicherheitslücken [BSI].

Um die Sicherheit der angesprochenen Systeme zu gewährleisten, scheinen viele Maßnahmen möglich und nötig. Zum einen werden von der Politik und Öffentlichkeit Stellen geschaffen, um Betroffene zu unterstützen, sowie die Nutzerinnen und Nutzer aufzuklären. Zum anderen muss auch bei der Quelle der Programme angesetzt werden. Sicherheitslücken präventiv direkt ab Zeitpunkt der Entwicklung zu verhindern soll die vorliegende Arbeit unterstützen. Dafür wird in dieser Arbeit ein Analysewerkzeug um seine Anwendbarkeit im Hinblick auf die Software-Sicherheit evaluiert und durch neue Analysemöglichkeiten erweitert werden.

[1] Ergebnisse aus eigener Zählung

Das Analysewerkzeug hilft dem Entwickler, ein komplexes System auf eine zugeschnittene Ansicht zu reduzieren und somit Schwachstellen in verschiedenen Formen leichter zu identifizieren. Dabei wird die Slicing-Technik verwendet. Diese Technik ermöglicht ein Zuschneiden des Quelltextes anhand verschiedener Parameter und Kriterien. Das Analysewerkzeug wurde bereits in vorherigen Arbeiten an der Universität Bremen um bestimmte Funktionalität erweitert, wie zum Beispiel der Möglichkeit die Systemservices des Android-Betriebssystems zu zerschneiden oder bessere Rekonstruktionen von relevanten Programmteilen aus einem Analyseergebnis zu ermöglichen. Zusätzlich beinhaltet das Werkzeug Möglichkeiten zum Untersuchen von Java-Programmen. Android und Java wurden aufgrund ihrer Verbreitung und weitreichenden Nutzung verwendet. Android hatte 2020 einen Marktanteil von 84,1%, iOS 15,9% und andere Konkurrenten weniger als 1% [IDC21]. Java ist im TIOBE-Index, der die Beliebtheit von Programmiersprachen anhand festgelegter Auswertungsmerkmale bestimmt, auf dem vierten Platz [TIO].

Im nachfolgenden Text wird auf die Ziele dieser Arbeit eingegangen. Dabei wird skizziert, wie die Weiterentwicklung des Analysewerkzeuges wesentlichen Einfluss auf die Schwerpunkte der Arbeit genommen hat und welche Forschungsfragen im Rahmen der Evaluation zu klären sind.

Einerseits bietet die vorliegende Arbeit eine Grundlage für weitere Entwicklungen am Analysewerkzeug. Dafür wird der Stand der Forschung sowie die vorherigen Arbeiten analysiert, das Analysewerkzeug in einen wartbaren Zustand gebracht, die aus den vorherigen Schritten resultierenden möglichen Weiterentwicklungen und Forschungsrichtungen dokumentiert sowie erläutert und das Analysewerkzeug bereitgestellt.

Andererseits gibt diese Arbeit weitere Aufschlüsse über die Anwendbarkeit der Slicing-Technik im Kontext der Informationssicherheit. Dafür werden neue Funktionen und Techniken implementiert, deren Entwicklungsprozess dokumentiert und zusätzlich deren Anwendbarkeit im Kontext der Informationssicherheit anhand von Fallbeispielen untersucht.

Das vorliegende Kapitel hat die Arbeit auf wissenschaftlicher, gesellschaftlicher und politischer Basis motiviert und dabei speziell die Rolle der Informationssicherheit im Bereich kritischer Systeme dargestellt. Zusätzlich wurde ein Überblick über die in der Arbeit erarbeiteten Ergebnisse und die dazugehörige Forschung gegeben.

Im folgenden Kapitel 2 wird auf die für das Verständnis der Evaluierungen und Diskussionsergebnisse wichtigen Grundlagen eingegangen.

2 Grundlagen

Um die vorliegende Arbeit mit den verwendeten Begrifflichkeiten und die gewonnenen Ergebnisse beurteilen zu können, werden in diesem Kapitel die benötigten Grundlagen behandelt.

Für die Einordnung der Slicing-Technik wird, in Kapitel 2.1, zuerst die Programmanalyse vorgestellt. Anschließend konzentriert sich Kapitel 2.2 auf das Verständnis der Technik sowie der Nachvollziehbarkeit von Ergebnissen. Darauf aufbauend wird die WALA-Bibliothek vorgestellt und relevante Interna erklärt. Schlussendlich werden in Kapitel 2.4 Begriffe und Techniken der Informationssicherheit vermittelt, damit die später behandelten Fallstudien nachvollziehbar und im Kontext der Informationssicherheit eingeordnet werden können.

Begonnen wird im folgenden Kapitel mit den Grundlagen der Programmanalyse.

2.1 Programmanalyse

Das folgende Kapitel vermittelt einen groben Überblick der Programmanalyse. Dabei geht es auf die wichtigen Begriffe und Techniken ein, es konzentriert sich auf die statischen Analysetechniken und spezieller das Slicing.

Programmanalyse ist die automatisierte Analyse von Programmen mit dem Ziel, das Programmverständnis zu verbessern und möglicherweise Fehler aufzudecken. Zusätzlich kann die Programmanalyse unterstützen das Verhalten des Programms zu verändern oder die Qualität des Quelltextes zu verbessern. Hierbei entstehen oft Daten sowie Informationen über das Programm, wie z. B. der System Dependency Graph (SDG) oder die Anzahl der Quellzeilen, welche für weitere Analyse oder automatisierte Veränderungen des Quelltextes, wie Optimierungen, verwendet werden.

[NNH99, NHR05] nennen vier große Bereiche der Programmanalyse. Dazu gehören die Datenflussanalyse, die beschränkungsbasierte Analyse (constraint-based analysis), die abstrakte Interpretation und die Typ- und Effektsysteme.

Die Techniken der Datenflussanalyse haben das Ziel zu analysieren, welche möglichen Werte eine bestimmte Variable annehmen kann. Das Ergebnis der Analyse kann dann verwendet werden, um zum Beispiel das Problem der Reaching-Definitions zu lösen oder Set-Use-Ketten zu erzeugen. Diese Set-Use-Ketten können anschließend als Testabdeckungsmetrik herangezogen werden.

Die Techniken der beschränkungsbasierten Analysen basieren auf der Idee, dass durch Einschränkungen in der Genauigkeit des Analyseergebnisses eine akzeptable Berechnungs-

zeit benötigt wird. Die Kontrollflussanalyse ist eine solcher Techniken, welche in verschiedenen beschränkten Formen erforscht wurde.

Im Themenfeld der abstrakten Interpretation geht es darum, die eigentliche Transformation von konkreten Werten in eine Transformation von Eigenschaften zu übersetzen, damit wird oft eine Reduzierung der für eine Analyse nötigen Berechnungen erreicht, ohne dabei den Raum der möglichen Analyseergebnisse zu verändern.

Die Typ- und Effektsysteme machen sich die Syntax von typisierten Sprachen zunutze und nehmen Effekte wie Exceptions, Seiteneffekte sowie Kommunikation als Möglichkeiten der Analyse auf. Die Kontrollflussanalyse ist auch unter dieser Art von Programmanalysen zu finden.

Allgemein ist die Slicing-Technik nicht in eine dieser Kategorien einzuordnen, sondern sie verwendet einen Zusammenschluss aus einzelnen Analysen bestimmter Kategorien, um schlussendlich ein Gesamtergebnis zu erzielen.

Die unter die Programmanalyse fallenden Techniken können mithilfe der verschiedenen Herangehensweisen unterschieden werden, welche für bestimmte Anwendungen auch kombiniert werden. Zum einen kann eine Analyse statisch oder dynamisch ablaufen. Eine statische Analyse konzentriert sich dabei auf den Quelltext, also den Teil des Programms der unveränderlich bleibt. Eine dynamische Analyse dagegen untersucht das Programm während der Ausführung. Bei der statischen Analyse ist, bei bestimmten Analysen, eine Approximation des Verhaltens zur Laufzeit notwendig. Slicing kann dynamisch oder statisch ausgeführt werden. In dieser Arbeit findet ausschließlich statisches Slicing Anwendung.

Bestimmte Analysen werden kontextsensitiv angewandt, dass bedeutet ein gespeicherter Zustand wird für die Analyse angelegt und beeinflusst das Ergebnis. Anderseits können Analysen auch nicht kontextsensitiv, also gespeicherten Zustand agieren. Für beide Arten von Analysen existieren Vor- und Nachteile.

Bei der Analyse gibt es zwei unterschiedliche Bereichseinschränkungen. Einmal die intraprozedurale Analyse. Dabei findet die Analyse nur in einzelnen Prozeduren statt. Zum anderen die interprozedurale Analyse. Hierbei findet die Analyse auch prozedurübergreifend statt. Das Slicing ist in beiden Formen vorhanden und in den folgenden Kapiteln wird auf beide Formen eingegangen.

Das vorliegende Kapitel gab einen groben Überblick über die Programmanalyse zur Einordnung der Slicing-Technik. Das nachfolgende Kapitel beschäftigt sich mit der Slicing-Technik im Detail, dabei wird auf die nötigen Datenstrukturen sowie der anschließenden Anwendung eingegangen.

2.2 Slicing

Für das Verständnis der Fallbeispiele, welche im Kapitel 5 behandelt werden, und Ergebnisse des Analysewerkzeuges ist es essenziell, ihre unterliegende Technik zu verstehen. Aus diesem Grund wird in dem vorliegenden Kapitel kurz auf die Entstehung der Slicing-Technik eingegangen und anschließend ihre Funktionsweise erklärt.

Die Idee, der Begriff und ein erster Algorithmus für das Slicing werden erstmals in [Wei84] vorgestellt. [Wei84] bezeichnet das Slicing wie folgt: „Program slicing is a method for automatically decomposing programs by analyzing their data flow and control flow“. Frei übersetzt ist das Slicing eine Methode, um Programme mithilfe von Datenfluss- und Kontrollflussanalysen automatisiert zu zerlegen.

[Wei84] erforschte dabei das Slicing mithilfe sogenannter *Hammock-Graphen* und setzte damit den ersten Grundstein für die weitere Forschung im Slicing-Bereich. Dabei zeigte [Wei84], dass es durch das Halteproblem nicht möglich ist anweisungsminimale Ergebnisse zu berechnen. Anweisungsminimale Ergebnisse sind Slices für die gilt, dass es keinen anderen vollständigen Slice gibt, welcher weniger Anweisungen enthält. Ein Slice stellt immer eine Approximation des anweisungsminimalen Slices dar, damit ist ein gültiger Slice auch immer das gesamte untersuchte Programm.

[OO84] entwickelte eine intraprozedurale Slicing-Technik auf Basis des Programmabhängigkeitsgraphen, engl. Program Dependency Graph (PDG). Für interprozedurales Slicing wurde in [HRB88, HRB90] der Systemabhängigkeitsgraph, engl. System Dependency Graph (SDG), vorgestellt. Dieser ermöglicht die Repräsentation von Programmen, welche mehrere Prozeduren beinhalten und geht dabei über Prozedurgrenzen hinaus. Die in [HRB88, HRB90] vorgestellte Slicing-Technik wird auch *HRB-Slicing* genannt.

In den folgenden Kapiteln wird erst auf die intraprozedurale Slicing-Technik eingegangen, dabei werden benötigte Datenstrukturen, wie z. B. der PDG, und Definitionen, wie z. B. die Kontrollabhängigkeit, erläutert. Darauf aufbauend wird, in Kapitel 2.2.2, die interprozedurale Slicing-Technik erklärt.

2.2.1 Intraprozedurales Slicing

In diesem Kapitel wird das intraprozedurale Slicing nach [OO84] und die dafür notwendigen Datenstrukturen und Definitionen erklärt. Für das intraprozedurale Slicing wird ein PDG errechnet, um mithilfe der Kanten des Graphen das Slicing durchzuführen. Die folgenden Kapitel zeigen nötige Schritte für die Errechnung des PDG und gehen dann auf das eigentliche Slicing ein. Als Erstes wird im folgenden Kapitel der Kontrollflussgraph definiert.

2.2.1.1 Kontrollflussgraph Der Kontrollflussgraph, engl. Control Flow Graph (CFG), greift die potenzielle Ausführungsreihenfolge von Instruktionen eines Programms auf. Der CFG ist ein gerichteter Graph G, wobei die Knoten V Anweisungen und die Kanten E den Kontrollfluss zwischen Anweisungen repräsentieren.

Der Graph enthält für jede Anweisung einen Knoten $n \in V$ und der Kontrollfluss wird durch die Kanten E repräsentiert. Zwischen zwei Knoten $n, m \in V$ gibt es eine Kante $e = (n, m) \in E$, sobald die Anweisung des Knoten m direkt nach der Anweisung von n ausgeführt werden könnte. Das bedeutet es gibt keine weitere Anweisung, die zwischen den Anweisungen von n oder m ausgeführt wird.

Die Kanten des Graphen werden markiert mit *true*, $false$, *exception*. Die Markierung *true* gibt den weiteren Kontrollfluss an, falls die Bedingung in der Anweisung des Quellknotens wahr ist. Die Markierung $false$ ist analog zur Markierung *true*. Zusätzlich werden Kanten mit *exception* markiert, falls diese den Kontrollfluss beim Auslösen einer Exception angeben.

Zusätzlich zu den Knoten, welche die Anweisungen des Programms repräsentieren, gibt es zwei weitere spezielle Knoten. Der Knoten $start \in V$ hat keine Vorgänger und von ihm aus ist jeder andere Knoten im Graphen erreichbar, dieser signalisiert den Startpunkt des Kontrollflusses. Der Knoten $end \in V$ hat keine Nachfolger und wird von jedem anderen Knoten im Graphen erreicht, dieser signalisiert den Endpunkt des Kontrollflusses [Gra16, Kri04].

Beispiel 1 *Aus dem Codeauszug 1 entsteht der in Abbildung 1 angegebene Graph. Im Graphen sind bestimmte Schlüsselwörter für den Kontrollfluss redundant, denn die Kanten des CFG geben den Kontrollfluss vollständig wieder. Deshalb sind in den Knoten des Graphen die Schlüsselwörter* `while` *und* `if` *ausgelassen worden.*

Zur Veranschaulichung werden zwei Kanten als Beispiel herangezogen. Die rote Kante im Graphen aus Abbildung 1 zeigt, dass die Bedingung der `if`*-Anweisung, aus Zeile 3 des Codeauszug 1, wahr ergibt und die Bedingung nicht erneut abgefragt wird. Dies spiegelt den Kontrollfluss beim Durchlaufen des* `else`*-Zweiges einer* `if`*-Anweisung wider.*

Die blaue Kante zeigt, dass die Bedingung des `while`*-Statements aus Zeile 7 wahr ergibt und die Schleife möglicherweise erneut durchgeführt wird. Diese Kante spiegelt den Kontrollfluss beim Durchlaufen des Schleifenkörpers einer* `while`*-Anweisung wider.*

Um das Beispiel einfach zu halten, wurde auf Kanten mit der exception-Markierung verzichtet.

```
1  x := 1
```

```
2
3   if(x >= 1) {
4     x := 10
5   }
6
7   while(x > 0) {
8     x := x - 1
9     y := x * x
10    z := z + y
11  }
```

Codeauszug 1: Beispielprogramm

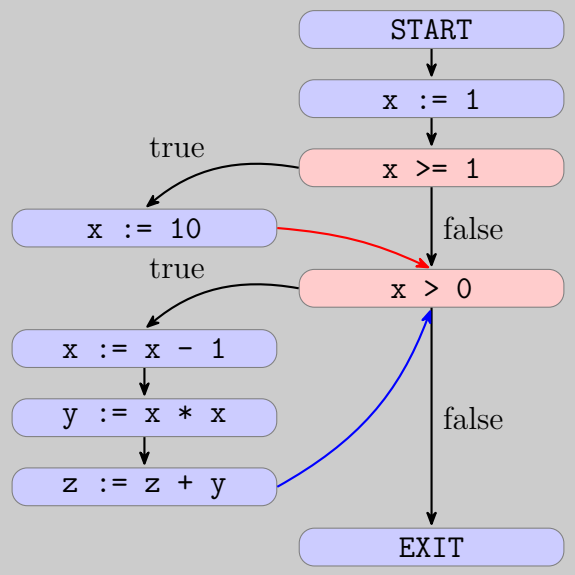


Abbildung 1: CFG des Beispielprogramms aus Codeauszug 1

Im folgenden Kapitel wird der Kontrollabhängigkeitsgraph und deren Berechnung auf Grundlage des CFG erklärt.

2.2.1.2 Kontrollabhängigkeitsgraph Der Kontrollabhängigkeitsgraph, engl. Control Dependency Graph (CDG), wird mithilfe des CFG erzeugt. Der CDG bildet sogenannte indirekte Abhängigkeiten und Kontrollabhängigkeiten zwischen Anweisungen ab [Gra16].

Für die Erzeugung des CDG werden die Knoten des CFG übernommen und anschließend die Kanten mithilfe der Definition von Kontrollabhängigkeit erzeugt. Für die Definition der Kontrollabhängigkeit definiert [FOW87] zuerst die Post-Dominanz.

Ein Knoten n post-dominiert einen Knoten m nur, wenn alle Pfade von m zum Knoten *end* den Knoten n enthalten. Intuitiv bedeutet das, ein Knoten von einem anderen post-dominiert wird, wenn dieser in jedem Pfad zum *end*-Knoten enthalten ist. Sobald also der dominierte Knoten ausgeführt wird, wird auch zwingend der post-dominierende Knoten ausgeführt. Mithilfe dieser Definition können wir die Kontrollabhängigkeit definieren.

Ein Knoten m ist kontrollabhängig von einem Knoten n, wenn es einen Pfad von n zu m gibt, in dem alle Knoten von m post-dominiert werden und m post-dominiert n nicht. Das bedeutet es gibt einen Pfad von n zum *end*, ohne das m vorkommt. Die Kontrollabhängigkeit gibt also intuitiv an, dass der Knoten n entscheidet, ob der Knoten m ausgeführt wird.

Über die Kontrollabhängigkeiten der Knoten können die Kanten des CDG definiert werden. Eine Kante (n, m) besagt, dass der Knoten m kontrollabhängig vom Knoten n ist. Der Startknoten *start* besitzt Kontrollabhängigkeiten zu allen Knoten, welche selbst nicht von anderen Knoten kontrollabhängig sind.

Beispiel 2 *Das Programmbeispiel aus Codeauszug 1 und der dazugehörige CFG in Abbildung 1 wurden in diesem Beispiel als Grundlage zur Erzeugung des CDG herangezogen. Die Knoten werden vom CFG übernommen.*

Der Knoten mit der Anweisung `x := 10`*, im Weiteren* `def` *genannt, aus dem Graphen in Abbildung 1 ist kontrollabhängig vom Knoten mit der Anweisung* `x >= 1`*, im Weiteren* `if` *genannt. Denn es gibt einen Pfad von* `if` *zu* `def`*, in dem alle Knoten von* `def` *post-dominiert werden, in diesem Fall enthält der Pfad zwischen* `if` *und* `def` *keinen Knoten. Zusätzlich post-dominiert* `def` *den Knoten* `if` *nicht. Das bedeutet es gibt einen Pfad von* `if` *zum* `end`*, ohne das* `def` *enthalten ist. Für diese Kontrollabhängigkeit wird eine Kante* (`if`, `def`) *im CDG eingeführt.*

Die gleiche Abhängigkeit ist vorhanden zwischen dem Knoten mit der Instruktion `x > 0` *und den Knoten mit den Anweisungen* `x := x - 1`*,* `y := x * x` *und* `z := z + y`*. Das bedeutet es existiert jeweils eine Kante von* `x > 0` *zu den aufgezählten Knoten.*

Die restlichen Kanten ergeben sich durch die Definition vom Startknoten. Der entstandene CDG ist in Abbildung 2 zu sehen.

Für die Berechnung wurden verschiedene Verfahren entwickelt, da diese den Rahmen der Arbeit jedoch überschreiten würden, werden sie nicht behandelt.

Mithilfe des CFG kann nicht nur der CDG erzeugt werden, sondern auch der Datenabhängigkeitsgraph, welcher im folgenden Kapitel behandelt wird.

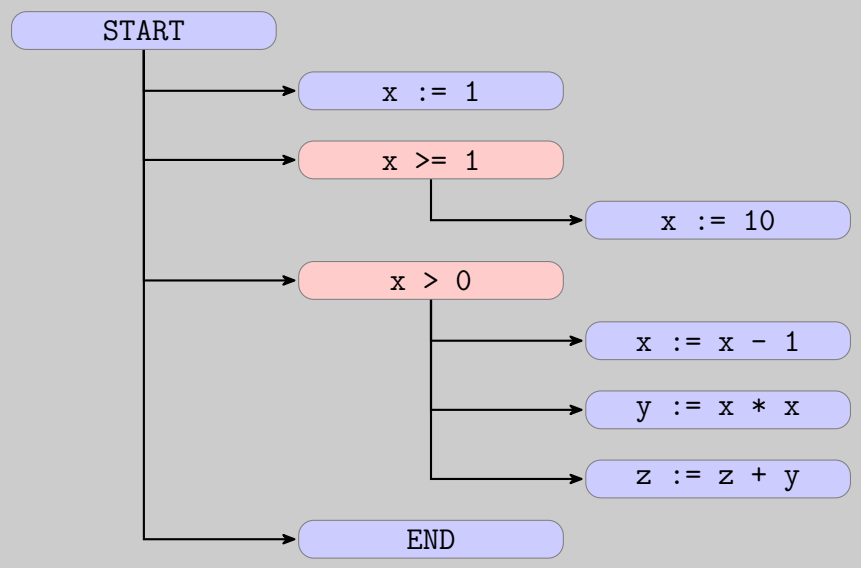


Abbildung 2: CDG des Beispielprogramms aus Codeauszug 1

2.2.1.3 Datenabhängigkeitsgraph Der Datenabhängigkeitgraph, engl Data Dependency Graph (DDG), beschreibt die Abhängigkeiten, welche durch Lese- und Schreibzugriff auf Variablen entstehen. Diese Abhängigkeiten entstehen, wenn in einer Anweisung ein Wert geschrieben wird, welcher in einer anderen Anweisung anschließend gelesen wird, ohne dass die Variable vorher erneut geschrieben wurde.

Um die Datenabhängigkeit zwischen Knoten definieren zu können, braucht es die *Set*- und *Use*-Mengen eines Knoten. Die *Set*-Menge eines Knoten besteht aus allen Variablen, welche in diesem Knoten geschrieben werden, und die *Use*-Menge enthält analog alle Variablen, die in diesem Knoten gelesen werden. Mithilfe dieser Mengen wird die Datenabhängigkeit zwischen Knoten definiert. Dabei gibt $set(n)$ die Set-Menge des Knoten n an respektive für $use(n)$.

Formal ist ein Knoten n datenabhängig von einem Knoten m, wenn ein Pfad $p = m \dots n$ mit $n, m \in V_{CFG}$ vorhanden ist und eine Variable v existiert, wobei $v \in set(m)$ und $v \in use(n)$ gilt und es keinen anderen Knoten $o \in p$ gibt, für den $v \in set(o)$ mit $o \neq m$ gilt. Diese Abhängigkeit ist eine sogenannte *Set-Use*-Beziehung, denn in einem Knoten wird eine Variable geschrieben, die in einem anderen Knoten gelesen wird.

In dieser Arbeit wird auf *Set-Use*-Beziehungen eingegangen. Nachfolgend bezeichnet eine Datenabhängigkeit zwischen zwei Knoten eine solche Beziehung. Diese Datenabhängigkeiten können in einem DDG dargestellt werden.

Beispiel 3 *Das Beispielprogramm aus Codeauszug 1 wird für die Erzeugung des DDG herangezogen.*

Für die Knoten des DDG werden die des CFG als Grundlage genommen und der start- und end-Knoten entfernt. Für die Kanten müssen die Datenabhängigkeiten berechnet werden, dies soll beispielhaft geschehen.

Der Knoten mit der Anweisung `x > 0`*, im Weiteren* `while` *genannt, ist datenabhängig vom Knoten mit der Anweisung* `x := 10`*. Denn es gibt einen Pfad von* `x := 10` *zu* `while` *ohne dass ein weiterer Knoten vorkommt, der die Variable* `x` *schreibt. Damit gibt es eine Kante von* `x := 10` *zu* `while`*. Trotzdem ist* `while` *auch datenabhängig vom Knoten* `x := 1`*, denn es gibt einen Pfad, auf dem* `x := 10` *nicht vorkommt.*

Der vollständige DDG mit allen Datenabhängigkeitskanten ist in Abbildung 3 dargestellt.

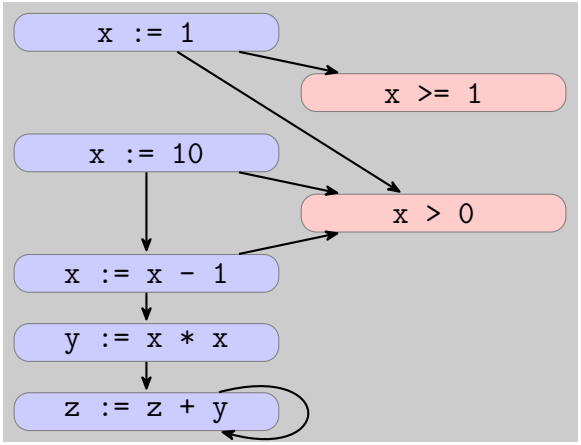


Abbildung 3: DDG des Beispielprogramms aus Codeauszug 1

Im folgenden Kapitel wird der Programmabhängigkeitsgraph und dessen Berechnung aus den vorherigen Graphen vorgestellt.

2.2.1.4 Programmabhängigkeitsgraph Der Programmabhängigkeitsgraph, engl. Program Dependency Graph (PDG), stellt Daten- und Kontrollabhängigkeiten zusammen dar. Dafür werden der CFG, CDG und DDG in einem Graphen zusammengeführt.

Der PDG erbt dabei die Knoten des CFG und übernimmt die Kanten des CDG und DDG. Der PDG besteht also aus Kontrollabhängigkeitskanten und Datenabhängigkeitskanten sowie den Knoten mit den Anweisungen des Programmes. Der entstandene PDG kann anschließend für das intraprozedurale Slicing verwendet werden.

Beispiel 4 *Für die Erzeugung des PDG werden die vorangegangenen Graphen und das Beispielprogramm aus Codeauszug 1 verwendet.*

In Abbildung 4 ist der vollständige PDG des Beispielprogrammes zu sehen. Blaue Kanten repräsentieren die Datenabhängigkeitskanten des DDG und schwarze Kanten die Kontrollabhängigkeiten des CDG.

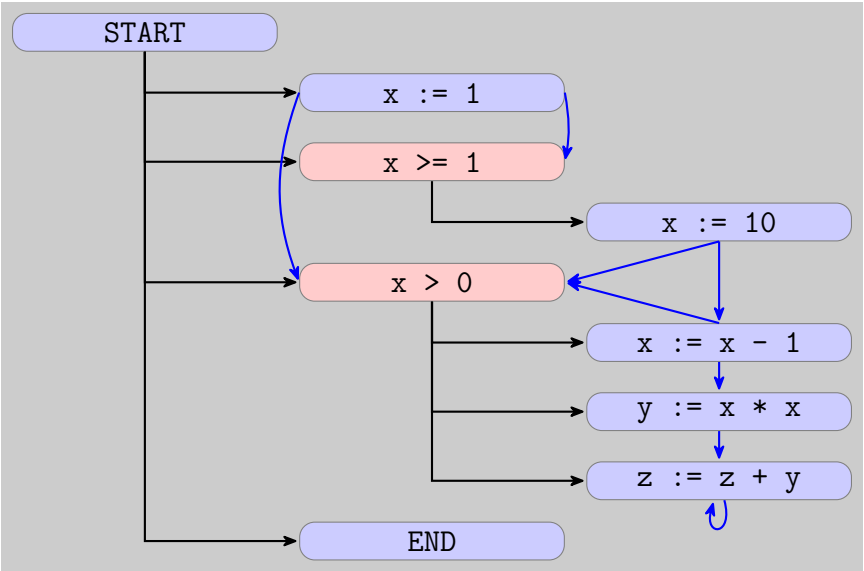


Abbildung 4: PDG des Beispielprogrammes aus Codeauszug 1. Blaue Kanten sind Datenabhängigkeiten und schwarze Kanten sind Kontrollabhängigkeiten.

Mithilfe des PDG kann nun das intraprozedurale Slicing durchgeführt werden. Allerdings wird im folgenden Kapitel erst auf die Single-Static-Assignment Form eingegangen, die den PDG vereinfacht und im WALA-Framework Anwendung findet.

2.2.1.5 Single-Static-Assignment Form Die Single-Static-Assignment Form (SSA-Form) ist eine Art von Zwischendarstellung, engl. Intermediate Representation (IR), und wird verwendet, um die Datenabhängigkeitskanten zu vereinfachen [SVKW07]. Zwischendarstellungen werden benutzt, um Quelltext zu abstrahieren. Das Abstrahieren des Quelltextes bietet die Möglichkeit, Algorithmen auf Basis der Zwischendarstellung zu entwickeln, ohne auf Spezifika der einzelnen Programmiersprachen eingehen zu müssen.

Die SSA-Form lässt keine mehrfachen Zuweisungen einer Variable zu. Falls es eine neue Zuweisung geben sollte, wird dafür ein neuer eindeutiger Variablenname eingeführt. Bei der Verwendung von Variablen gibt es keine Einschränkung.

Beispiel 5 *Im Codeauszug 2 werden drei Variablen erzeugt, dabei wird eine Variable zweimal beschrieben. Im Codeauszug 3 ist das vorherige Programm in SSA-Form übersetzt worden. Jede Zuweisung bekommt hier einen eindeutigen Variablennamen. Dabei wird*

ersichtlich, dass die Zuweisung der Variable **`x1`** *im weiteren Programmverlauf nicht von Relevanz für die Berechnung der anderen Variablen ist.*

```
1  x := 1
2  z := 3
3  x := z
4  y := x
```

Codeauszug 2: Nicht in SSA-Form

```
1  x1 := 1
2  z1 := 3
3  x2 := z1
4  y1 := x2
```

Codeauszug 3: In SSA-Form

Nur die Umbenennung von Variablen alleine reicht nicht aus, denn nicht jede neue Zuweisung von Variablen würde eindeutig auf eine vorher definierte Variable schließen. Der Kontrollfluss kann an einer bestimmten Stelle zwischen zwei weiteren Programmabläufen entscheiden. Beim Zusammentreffen der Programmabläufe kann es passieren, dass eine Variable in beiden Abläufen zugewiesen wurde, somit ist eine spätere Zuweisung, zumindest nur über die Variablenbenennung, nicht mehr eindeutig.

Für das Problem der unterschiedlichen Zuweisungen in auseinandergehenden Kontrollflüssen wird der ϕ-Knoten eingesetzt. An diesem Knoten wird entschieden, welche Zuweisung für eine Variable geschehen muss, um das korrekte Programmverhalten zu repräsentieren.

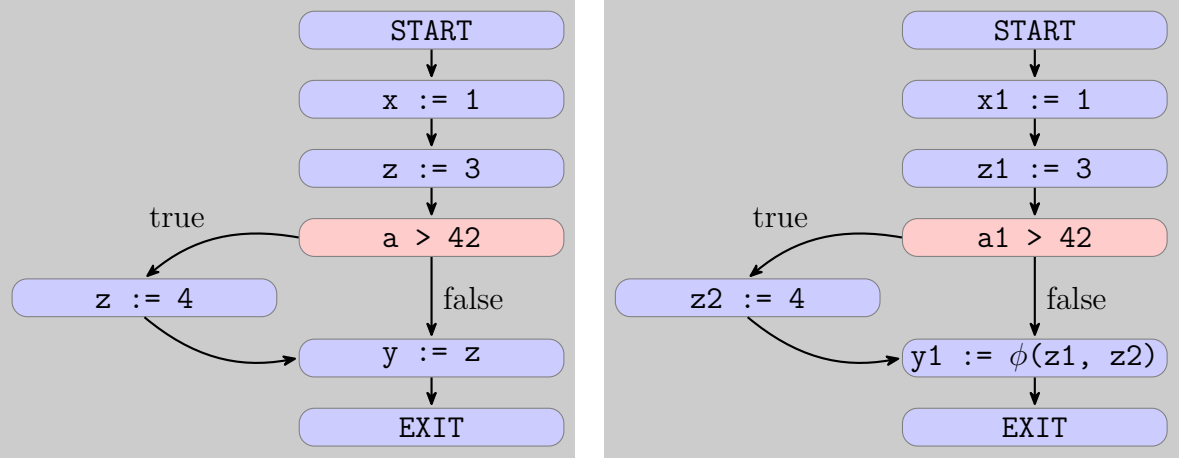
Beispiel 6 *Im Codeauszug 4 ist ein Programm mit abzweigendem Kontrollfluss dargestellt. Wenn dieser Codeauszug nun in SSA-Form übersetzt werden sollte, würde die Zuweisung* **`y := z`** *nicht eindeutig zu einer der vorherigen Definitionen von* **`z`** *führen. Deshalb wird im Codeauszug 5 ein ϕ-Knoten eingeführt, dieser Knoten referenziert beide Definitionen von* **`z`***, also* **`z1`** *und* **`z2`***, und erhält somit die korrekten Datenabhängigkeiten. In Abbildung 5a ist der resultierende CFG des Codeauszug 4 zu sehen, daneben reiht sich in Abbildung 5b der CFG des Programmes in SSA-Form. In der Abbildung 5b ist zu sehen, wie bei der Zuweisung* **`y := z`** *im Graphen mit SSA-Form durch den ϕ-Knoten unterschieden wird, ob* **`z1`** *oder* **`z2`** *den neuen Wert von* **`y1`** *angibt.*

```
1  x := 1
2  z := 3
3
4  if(a > 42) {
5    z := 4
6  }
7
8  y := z
```

Codeauszug 4: Nicht in SSA-Form

```
1  x1 := 1
2  z1 := 3
3
4  if(a > 42) {
5    z2 := 4
6  }
7
8  y1 := φ(z1, z2)
```

Codeauszug 5: In SSA-Form



(a) CFG ohne SSA-Form

(b) CFG mit SSA-Form

Abbildung 5: CFG (a) ohne SSA-Form und (b) mit SSA-Form

Die SSA-Form wird auch vom WALA-Framework als Zwischendarstellung verwendet. Dafür werden die Java-Bytecode Befehle in diese Zwischendarstellung umgewandelt und weiterverarbeitet.

Alle für das intraprozedurale Slicing notwendigen Datenstrukturen und Definitionen wurden in den vorherigen Kapiteln erklärt. Als Letztes folgt im nächsten Kapitel die eigentliche Durchführung des intraprozeduralen Slicing.

2.2.1.6 Durchführung des intraprozeduralen Slicing In den vorherigen Kapiteln wurden die benötigten Daten, welche über das Programm gesammelt werden müssen, vorgestellt und deren Erzeugung erklärt. Der letzte Schritt besteht aus dem Slicing. Es gibt grundsätzlich verschiedene Varianten des Slicing. Im Folgenden werden die Definitionen des Forward- und Backward-Slicing vorgestellt, sowie deren Durchführung erläutert.

Beim Slicing wird ein Programmpunkt ausgewählt. Dieser Programmpunkt wird als

Seed-Statement bezeichnet und legt fest, welche Teile des Programmes entfernt werden. Es werden die Teile des Programmes entfernt, welche das Verhalten des Programmpunktes nicht beeinflussen. Anweisungen beeinflussen einen Programmpunkt, wenn der Programmpunkt kontroll- oder datenabhängig von diesen ist.

Durch die Erzeugung des PDG und der Definition der Beeinflussung wird deutlich, dass ein Slicing mithilfe der Kanten des PDG durchgeführt werden kann. Beim Forward-Slicing werden die Kanten des PDG, ausgehend vom Programmpunkt, in ihrer eigentlichen Kontrollflussrichtung durchlaufen und besuchte Knoten werden im Ergebnis gespeichert. Beim Backward-Slicing werden die Kanten in umgekehrter Richtung durchlaufen.

Intuitiv bedeutet das, dass beim Forward-Slicing Teile des Programmes erhalten werden, auf die der gewählte Programmpunkt Einfluss nimmt und beim Backward-Slicing werden Teile des Programmes erhalten, welche den gewählten Programmpunkt beeinflussen.

Das intraprozedurale Slicing konzentriert sich nur auf einzelne Prozeduren. Durch die steigende Komplexität und die Verbreitung objektorientierter Programmierung wird im folgenden Kapitel das interprozedurale Slicing erklärt, welches über Prozeduren hinweg funktioniert.

2.2.2 Interprozedurales Slicing

In diesem Kapitel werden die notwendigen Datenstrukturen und Algorithmen erläutert, welche notwendig für das interprozedurale Slicing in Form des *HRB-Slicing* sind [HRB90]. Zuerst wird dabei auf den Call Graph (CG) eingegangen, um anschließend die Points-To Analyse zu erklären, welche einen Einfluss auf die Genauigkeit des CG hat. Darauffolgend wird der System Dependency Graph (SDG) und schlussendlich die Durchführung des interprozeduralen Slicing erklärt.

2.2.2.1 Aufrufgraph

In diesem Kapitel wird der allgemeine Aufbau eines Aufrufgraphen, engl. Callgraph (CG), erklärt. Der CG stellt die Aufrufe von Prozeduren im Programm dar, analysiert dabei die Klassen sowie Objekte und hat wesentlichen Einfluss auf den SDG. Die Erzeugung des CG ist der erste von zwei Schritten, um interprozedurales Slicing durchzuführen. Denn dieser bietet die Möglichkeit, Funktionsaufrufe zu unterscheiden und somit prozedurübergreifende Abhängigkeiten darzustellen.

Die Knoten des CG bestehen aus zwei Knotenarten, den Prozeduren und Aufrufanweisungen. Kanten haben die Form (m_1, c, m_2), wobei m_1 die Prozedur repräsentiert in der eine Aufrufanweisung c vorhanden ist, die die Prozedur m_2 aufruft.

Im Codeauszug 6 ist ein Beispielprogramm gegeben und in Abbildung 6 ist der dazugehörige CG abgebildet.

```
1  class A {
2    int foo() {
3      return 42;
4    }
5
6    void print() {
7      println(foo())
8    }
9  }
10
11 class B extends A {
12   int foo() {
13     return 23;
14   }
15
16   void main(String argv[]) {
17     A a = new A();
18     A b = new B();
19     a.print();
20     b.print();
21     if(argv[1].equals("B")) {
22       a = b;
23     }
24     a.print();
25   }
26 }
```

Codeauszug 6: Beispielprogramm [Gra16]

Der CG aus Abbildung 6 ist, bei genauerer Betrachtung, nur eine Approximation der möglichen Aufrufe des Beispielprogrammes aus Codeauszug 6. Im Knoten 6 des Graphen werden immer die `foo`-Methoden beider Klassen `A` und `B` in Betracht gezogen. Dies entspricht, z. B. bei einem Aufruf der `print`-Methode auf einem Objekt der Klasse `A`, nicht immer der Realität. [Gra16] beschreibt eine Methode zum Vergrößern der Genauigkeit des erzeugten CG, dabei werden Teile des CG, je nach Aufrufkontext, geklont. Diese und andere Optimierungen würden den Rahmen der Arbeit überschreiten. Aus diesem Grund

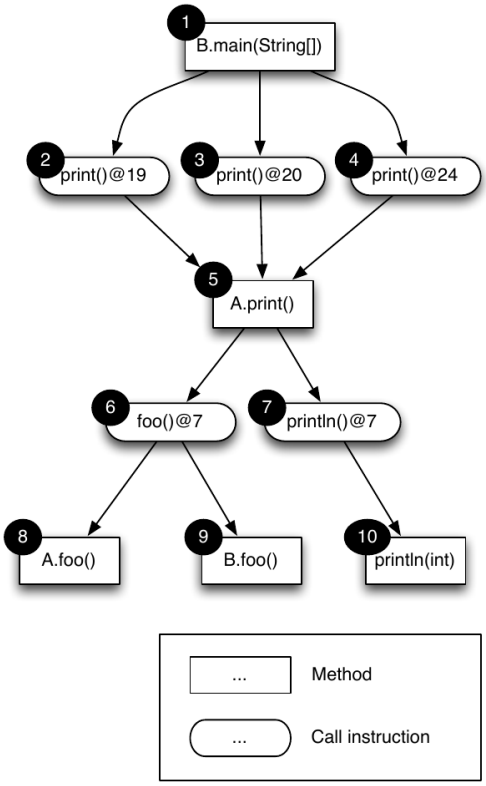


Abbildung 6: CG des Beispielprogramms aus Codeauszug 6 [Gra16]

wird auf die Literatur in [Gra16, Kri04, ALSU06] verwiesen.

In diesem Kapitel wurde der CG sowie die darunterliegende Approximation behandelt. Dabei wurde nicht auf die Erzeugung eingegangen. Um zu bestimmen, welches Objekt einer bestimmten Klasse hinter einem Aufruf steckt, wird im folgenden Kapitel die sogenannte Points-To Analyse erläutert, welche diese Informationen aus dem Programmcode extrahiert und damit die Erzeugung des CG ermöglicht.

2.2.2.2 Points-To Analyse Bei einer statischen Analyse müssen alle Effekte die zur Laufzeit passieren können approximiert werden. Die Points-To Analyse untersucht, welche Speicheradressen zur Laufzeit möglicherweise gelesen oder modifiziert werden. In objektorientierten Sprachen werden sogenannte Zeiger, engl. Pointer, eingesetzt, um auf Speicheradressen zu zeigen, welche bestimmte Objektinstanzen enthalten. Java verwendet solche Zeiger im Hintergrund. Diese werden Referenzen genannt und bieten keine Zeiger-Arithmetik für den Entwickler.

Durch sogenannte Aliasse wird die Analyse noch erschwert. Aliasse sind verschiedene Variablen, welche eine Referenz auf dieselbe Objektinstanz sind. Das würde bedeuten

eine Änderung einer dieser Aliasse ändert auch die Auflösung anderer Aliasse derselben Objektinstanz.

Die Points-To Analyse sammelt Informationen über die Variablen bei der interprozeduralen Programmanalyse. Einerseits berechnet sie die konkreten Objektinstanzen, auf welche eine Variable oder ein Objektattribut, während der Programmlaufzeit, zeigen kann. Damit ermöglicht die Analyse dynamische Bindungen für die Konstruktion des Aufrufgraphen aufzulösen. Andererseits werden Seiteneffekte, welche durch Methodenaufrufe auftreten, bestimmt. Diese fließen anschließend in die Konstruktion des CG mit ein.

Die Analyse kann in verschiedenen Genauigkeiten ausgeführt werden, um die Genauigkeit des Aufrufgraphen und sowie Größe und Berechnungszeit zu beeinflussen. Ausführliche Erklärungen zu den einzelnen Möglichkeiten zur Beschränkung oder Erweiterung der Genauigkeit sind in [Gra16, Cyl19] zu finden.

Im vorherigen Kapitel wurde der CG vorgestellt, um in diesem Kapitel die Points-To Analyse zu erklären. Die Analyse nimmt direkten Einfluss auf die Berechnung des CG und damit auch auf die Genauigkeit des interprozeduralen Slicing. Im folgenden Kapitel wird der SDG erklärt, welcher als Grundlage für das interprozedurale Slicing verwendet wird.

2.2.2.3 Systemabhängigkeitsgraph Um interprozedurales Slicing durchzuführen, wird ein Graph benötigt, der anschließend in einer bestimmten Weise traversiert wird. Beim HRB-Slicing aus [HRB90] ist dies der SDG. Der SDG verwendet die PDG aller Methoden und verknüpft diese über ihre Abhängigkeiten.

Ein SDG besteht aus den PDG aller Methoden, den synthetischen Parameter-Knoten sowie dazugehörigen Parameter-In- und Parameter-Out-Kanten, den Call-Kanten, welche die PDG verbinden und den Summary-Edges. Die PDG wurden in Kapitel 2.2.1.4 erläutert.

Die PDG referenzieren sich durch Aufrufe anderer Methoden gegenseitig und müssen miteinander verbunden werden. Die Call-Site-Knoten bilden das Aufrufen einer Methode ab und werden mithilfe einer Call-Kante mit den Eintrittspunkten der PDG aufgerufener Methoden verbunden.

Alle Werte und Speicheradressen, welche die aufgerufene Methode referenziert oder modifiziert, werden durch synthetische Parameter-Knoten abgebildet. Referenzierte Werte oder Speicheradressen, werden durch einen Actual-In-Knoten beim Caller, welcher die aufrufende Methode darstellt, und einem zugehörigen Parameter-In-Knoten beim Callee, welcher die aufgerufene Methode darstellt, repräsentiert. Die Knoten sind über eine Parameter-In-Kante verbunden, welche den Informationsfluss in den Callee abbildet. Modifizierte Werte oder Speicheradressen, werden über Formal-Out-Knoten beim Callee und

einem zugehörigen Actual-Out-Knoten beim Caller sowie der verbindenden Parameter-Out-Kante repräsentiert.

Die Formal-In- und Formal-Out-Knoten sind dabei kontrollabhängig vom Call-Site-Knoten und Parameter auf der Seite des Caller sind datenabhängig von Knoten in denen die zugehörigen Variablen gesetzt werden. Diese Parameter-Knoten und -Kanten bilden die Abhängigkeiten der eingegebenen Werte und Speicheradressen mit der aufgerufenen Methode ab. Dabei werden auch Seiteneffekte über Methodenaufrufe bei objektorientierten Sprachen über synthetische Parameter-Knoten und -Kanten abgebildet. Diese Seiteneffekte werden über die Points-To Analyse approximiert.

Summary-Edges werden eingesetzt, um die kontext-sensitive Traversierung des SDG zu vereinfachen. Sie geben Aufschluss über den Caller von dem ein bestimmter PDG, während der Traversierung, betreten wurde. Summary-Edges verbinden die Actual-In-Knoten eines Callers mit den Actual-Out-Knoten desselben Callers.

2.2.2.4 Durchführung des interprozeduralen Slicing In diesem Kapitel soll die kontextsensitive Traversierung des SDG für das aus [HRB90] stammende HRB-Slicing erläutert werden. Dafür wird die Motivation für den zweiphasigen Algorithmus skizziert und anschließend auf die Funktionweise eingegangen.

Die Traversierung des Graphen beginnt bei dem sogenannten Seed-Statement, also einem bestimmten Knoten im SDG.

Ein einfacher rückwärts-traversierender Algorithmus, welcher ähnlich dem intraprozeduralen Slicing vorgeht, würde nicht zwischen unterschiedlichen Aufrufen der gleichen Methode unterscheiden und somit das Ergebnis möglicherweise ungenauer und damit die enthaltene Anzahl von Anweisungen erhöhen. Um dem entgegenzuwirken, wird beim HRB-Slicing der Caller zwischengespeichert, durch welchen ein bestimmter Callee aufgerufen wird. Dafür werden die Summary-Edges verwendet, die den Aufwand im Vergleich zu einer einfachen Liste reduzieren.

Das eigentliche Slicing wird in zwei Phasen durchgeführt. In der ersten Phase werden alle Datenabhängigkeits-, Kontrollabhängigkeits-, Call-, Summary- und Parameter-In-Kanten rückwärts durchlaufen, besuchte Knoten werden markiert. Parameter-Out-Kanten werden explizit ignoriert und die Formal-Out-Knoten besuchter Actual-Out-Knoten werden zwischengespeichert für die nächste Phase.

In der zweiten Phase werden, ausgehend von den in der ersten Phase gespeicherten Formal-Out-Knoten, wie in der ersten Phase alle Kanten rückwärts durchlaufen. Diesmal werden statt Parameter-In-Kanten allerdings die Parameter-Out-Kanten mit einbezogen.

Vereinfacht gesagt wird der SDG in der ersten Phase nur „nach oben“ durchlaufen, um

aufrufende Methoden mit einzubeziehen. Anschließend wird in der zweiten Phase „nach unten“ durchlaufen, um aufgerufene Methoden und deren Abhängigkeiten mit einzubeziehen.

In den vorherigen Kapiteln 2.2.1 und 2.2.2 wurden das intraprozedurale Slicing nach [OO84] und das interprozedurale Slicing nach [HRB90] vorgestellt und anhand von Beispielen erläutert. Im folgenden Kapitel werden andere Slicing-Varianten vorgestellt, diese Varianten unterscheiden sich bei der Zielsetzung, Durchführung und Berechnungen.

2.2.3 Thin-Slicing

Thin-Slicing [SFB07] ist eine Variante des Slicing, die sich aus dem Umstand motiviert, dass traditionelles Slicing, z. B. HRB-Slicing, zu viele Anweisungen im Ergebnis enthält, welche für den Entwickler zur Fehlerfindung nicht von Bedeutung sind. Thin-Slicing wurde in dem in dieser Arbeit erweiterten Analysewerkzeug implementiert und für die spätere Evaluierung verwendet. In diesem Kapitel werden wichtige Begriffe für das Verständnis der Technik erklärt. Diese Erklärungen werden mithilfe von Beispielen verdeutlicht.

[SFB07] definiert einen Thin-Slice als Slice, welcher nur sogenannte *producer-statements* für das vorgegebene Seed-Statement enthält. Dabei entsteht, entgegen [Wei84], ein nicht ausführbarer Slice.

Thin-Slicing definiert also die Relevanz von Anweisungen neu. Es sind nur Anweisungen relevant, welche dem Entwickler helfen zu verstehen, weshalb ein bestimmter Wert zustande kommt, ohne dabei z. B. auf interne Implementierungen von Datenstrukturen zu achten, sondern nur auf Anweisungen die den Wert im Seed-Statement direkt beeinflussen. Deshalb unterscheidet [SFB07] zwischen den *producer-statements*, welche eben Teil der Kette von Anweisungen sind, die direkten Einfluss auf den Wert im Seed-Statement nehmen, und den sogenannten *explainer-statments*. Die *explainer-statements* sind, wie der Name ausdrückt, Anweisungen, welche ausschließlich erklären, warum ein *producer-statement* das Seed-Statement bzw. den Wert im Seed-Statement beeinflusst. [SFB07] stellt fest, dass die *explainer-statements* hauptsächlich den Slice vergrößern und nicht zum Programmverständnis beitragen.

Falls Thin-Slicing nicht das gewünschte Ergebnis liefert, stellt [SFB07] die Möglichkeit vor, Thin-Slicing hierarchisch anzuwenden und somit inkrementell den Suchraum mithilfe weiterer *producer-statements* zu vergrößern.

[SFB07] zeigt auch, dass mithilfe dieser Definition von Slicing die Anzahl von Anweisungen in Slices deutlich reduziert werden kann und trotzdem oft die wichtigsten Anweisungen für das Programmverständnis und die Fehlerfindung enthalten sind. Thin-Slicing ist dabei

statisch und dynamisch anwendbar und besitzt eine kontextsensitive und nicht kontextsensitive Variante.

In diesem Kapitel wurde das Thin-Slicing skizziert und erklärt. Im nächsten Kapitel 2.3 wird das WALA Framework, seine Geschichte, die verwendeten internen Techniken und ein Überblick über die Funktionsweise gezeigt.

2.3 WALA Framework

Die *T.J.* ***Wa****tson* ***L****ibraries for* ***A****nalysis* (WALA) sind eine Sammlung von verschiedenen Implementierungen zur statischen Analyse von Java Bytecode sowie darauf basierenden Sprachen und JavaScript. Das WALA-Framework entstand ursprünglich als Teil des DOMO Projektes am IBM Thomas. J. Watson Research Center, wurde 2006 von IBM an die Allgemeinheit übergeben und unter die Eclipse Public License gestellt.

WALA bietet vielseitige Werkzeuge und eine Application Programming Interface (API) zum statischen Analysieren von Programmen an. In diesem Kapitel wird die, für diese Arbeit relevante, technische Funktionsweise erläutert und Einstellungsmöglichkeiten zum Beeinflussen des Analyseergebnisses aufgezeigt.

2.3.1 Technische Funktionsweise

Die typische Funktionsweise vom WALA-Framework besteht darin eine Klassenhierarchie zu erzeugen [Conc]. Das bedeutet der Programmcode wird in den Speicher geladen und es werden generelle Informationen über Typen und deren Hierarchie gesammelt. Anschließend wird ein CG erzeugt [Conb]. Dabei wird auch eine Points-To-Analyse durchgeführt, um dynamische Aufrufe auflösen zu können [Conf]. Schlussendlich wird auf dem entstandenen CG eine Analyse durchgeführt oder weitere Datenstrukturen erzeugt, wie etwa der SDG. Im Folgenden werden diese einzelnen Schritte im Detail analysiert.

Die Klassenhierarchie wird verwendet, um die Vererbungshierarchie eines Programmes darzustellen. Sie bildet die Gesamtheit des zu analysierenden Quelltextes ab. Dabei kann mithilfe des `AnalysisScope` der zu analysierende Bereich des Programmquelltextes und Bibliotheksquelltextes eingegrenzt werden [Cona]. Aus diesem zu analysierenden Bereich wird die Klassenhierarchie erstellt. Der `AnalysisScope` bietet die Möglichkeit explizit einzelne Klassen oder gesamte Pakete, wie zum Beispiel der Java-Standardbibliothek, auszuschließen.

Die WALA Klassenhierarchie ist in ihrer Struktur ähnlich der Java-Klassenhierarchie. Die Klassenhierarchie wird über Namensräume für Klassen modelliert. Dabei wird zum Beispiel zwischen `Application` und `Primordial` unterschieden. Klassen die unter

`Application` fallen, sind Klassen aus dem zu analysierenden Programm. Klassen, welche unter `Primordial` fallen, sind aus der Java-Standardbibliothek, also beispielsweise aus dem Paket `java.*`. Für jeden dieser Namensräume werden `ClassLoader` angelegt. Diese `ClassLoader` werden hierarchisch angeordnet und helfen beim Auffinden der Klassen. Diese Abbildung von Namensräumen ähnelt derer von Java.

Um die Anweisungen von Methoden darzustellen, besitzt das WALA-Framework eine eigene IR, diese verwendet die SSA-Form mit einigen Anpassungen für Java-Bytecode [Cone]. Die IR des Frameworks beinhaltet den CFG und seine Bestandteile in SSA-Form.

Nach dem Erzeugen der Klassenhierarchie kann ein CG mithilfe der Points-To Analyse erzeugt werden [Conf]. Mithilfe des CG, der Klassenhierarchie und des CFG kann schlussendlich ein SDG erzeugt und zum Slicing verwendet werden.

Im Folgenden werden die verschiedenen Einstellungsmöglichkeiten, welche die erklärte technische Funktionsweise und das Slicing-Ergebnis beeinflussen, erläutert.

2.3.2 Einstellungsmöglichkeiten

Um die Ergebnisse der Fallstudien im Kapitel 5 und die Komplexität der Entwicklungen dieser Arbeit nachvollziehen und bewerten zu können, werden im folgenden Kapitel die verschiedenen Einstellungsmöglichkeiten des WALA-Frameworks vorgestellt. Dabei werden nur für die Arbeit relevante Einstellungen in Betracht gezogen.

2.3.2.1 Datenabhängigkeitsoptionen

Mithilfe der Datenabhängigkeitsoptionen kann festgelegt werden, welche Datenabhängigkeiten für die Erzeugung des SDG von Relevanz sind. Mithilfe der Optionen können zum Beispiel die Datenabhängigkeiten, welche durch Exceptions oder Heap-Zugriffe entstehen, ausgeschlossen werden. Auch können Datenabhängigkeiten komplett ausgestellt werden.

Die Option `NO_BASE_PTRS` besagt, das alle Abhängigkeiten, welche durch Zeiger auf Attribute von Klassen entstehen, ignoriert werden. `NO_HEAP` lässt alle Abhängigkeiten von und zu Heap Positionen aus. `NO_EXCEPTIONS` ignoriert Abhängigkeiten zu und von `try`- und `catch`-Anweisungen. Diese verschiedenen Optionen sind auch als Kombinationen vorhanden. Zusätzlich gibt es die Optionen `FULL` und `NONE`, die entweder alle Datenabhängigkeiten mit einbeziehen oder keine.

2.3.2.2 Kontrollabhängigkeitsoptionen

Ähnlich der Datenabhängigkeitsoptionen wird mithilfe der Kontrollabhängigkeitsoptionen angegeben, welche Kontrollabhängigkeiten für die Erzeugung des SDG herangezogen werden sollen. Dabei können z. B. Exception- oder interprozedurale Aufruf-Kontrollflüsse

ausgelassen werden. Auch hier gibt WALA die Optionen als vordefinierte Kombinationen vor.

Die Option `NO_EXCEPTIONAL_EDGES` besagt das keine Kontrollabhängigkeiten durch mögliche Exceptions im SDG aufgenommen werden. `NO_INTERPROC_EDGES` gibt WALA an alle interprozeduralen Abhängigkeiten zwischen Callern und Callees zu ignorieren.

2.3.2.3 Points-To Analyse

Die Points-To Analyse ist bei WALA in zwei Varianten implementiert, der *Anderson-style* Variante [And94, Conf] und der *demand-driven* Variante [Sri07, SB06, Cond]. In diesem Kapitel wird speziell auf die Einstellmöglichkeiten der *Anderson-style* Variante eingegangen, da die *demand-driven* Variante für diese Arbeit nicht von Relevanz ist, da sie in der Evaluierung nicht verwendet wird.

Über bestimmte Regeln kann die Points-To Analyse eingestellt werden, dabei sind zwei Dimensionen zu betrachten. Erstens kann ausgewählt werden, wie die Analyse Zeiger und Heap-Positionen abstrahiert. Zweitens kann entschieden werden, wie die Analyse bei der Erzeugung des CG (vgl. Kapitel 2.2.2.1) Methoden anhand vom Kontext klont.

In WALA gibt es die Möglichkeit die Abstraktion der Zeiger- und Heap-Positionen genau festzulegen. Allerdings werden in dieser Arbeit nur vorgefertigte Optionen verwendet.

2.3.2.4 Exclusionfile

Mithilfe der Exclusionfile kann der Analysebereich eingeschränkt werden. Der Analysebereich gibt z. B. an, welcher Bereich des Programmes sowie verwendeten Classpaths für den Aufbau des CG herangezogen werden. Dabei können bestimmte externe (z. B. *de.example.CoolClazz*) oder Java-interne Klassen (z. B. *java.lang.Integer*) ignoriert werden, die den CG und das Analyseergebnis mitunter unnötig vergrößern würden. Natürlich muss diese Datei mit bedacht gewählt werden, da sie das Ergebnis möglicherweise verfälscht und wichtige Teile für die spätere Analyse ausschließt. In der Arbeit wurde die in Anhang B gelistete Datei verwendet.

2.4 Informationssicherheit

Für das Verständnis und Bewertung der Ergebnisse der späteren Fallstudien, wird in diesem Kapitel auf die Informationssicherheit, ihre Ziele und Möglichkeiten zur Erreichung dieser, eingegangen.

Allgemein wird bei der Sicherheit zwischen Funktionssicherheit (*safety*), Informationssicherheit (*security*) und Datensicherheit (*protection*) unterschieden [Eck18]. Die Funktions-

sicherheit ist die Eigenschaft eines System, dass die erwartete Funktionalität, z. B. aus einer Spezifikation, mit der reellen Funktionalität des implementierten Systems übereinstimmt. Datensicherheit ist die Eigenschaft eines Systems, keine Systemzustände anzunehmen, welche unautorisierten Zugriff auf Daten sowie Systemresourcen führen und somit Datenverlust hervorrufen könnten. Informationssicherheit ist die Eigenschaft eines System, nur Systemzustände anzunehmen, welche nicht zu einem Informationsgewinn oder zu einer Informationsveränderung durch unautorisierte Dritte führt.

Im folgenden Unterunterabschnitt 2.4.1 wird auf die Unterscheidung zwischen Objekt und Subjekt im Kontext der Informationssicherheit eingegangen, um anschließend in Unterunterabschnitt 2.4.2 die Schutzziele aufzulisten. Anschließend werden in den darauffolgenden Kapiteln Möglichkeiten zur Erfüllung der vorgestellten Schutzziele dargelegt.

2.4.1 Subjekt und Objekt

In der Informationssicherheit gibt es zwei wichtige Akteure, die Objekte und Subjekte. Bei den Objekten wird zwischen aktiven Objekten, also z. B. Prozesse, und passiven Objekten, wie z. B. Dateien oder Datenbankeinträge, unterschieden.

Die von außen nutzbaren Schnittstellen eines IT-Systems werden insbesondere von seinen Nutzern verwendet. Die Nutzer eines IT-Systems gehören zu den Subjekten, ebenso wie von ihm erzeugte Prozesse oder Prozeduren.

2.4.2 Schutzziele

Um die Sicherheit eines Systems bewerten zu können, werden für die Informationssicherheit verschiedene Ziele definiert [BS18, Eck18]. Diese werden im vorliegenden Kapitel erläutert.

Das Ziel der **Geheimhaltung** (engl. *secrecy*) besagt, dass der Zugriff für Personen eingeschränkt werden muss.

Das Ziel der **Vertraulichkeit** (engl. *confidentiality*) besagt, dass es eine Verpflichtung zur Geheimhaltung von Informationen anderer geben muss. Das bedeutet es darf kein ungewollter Informationsfluss von einem Subjekt zum Anderen geben. Dieses Problem ist auch als Confinement-Problem bekannt [Lam73]. Auch darf nicht erkennbar sein, dass Dateien angelegt wurden, wenn das Subjekt keine Berechtigung hat, diese zu lesen. Diese Problematik ist als Interferenz-Kontrolle bekannt [DE82].

Das Ziel des **Datenschutzes** (engl. *privacy*) besagt, dass es ein Recht auf Schutz eigener persönlicher und unpersönlicher Informationen geben muss. Die Weitergabe von Informationen persönlicher Art kann durch die betroffene Person kontrolliert werden. Diese

Anforderungen regelt der Gesetzgeber beispielsweise über das Bundesdatenschutzgesetz (BDSG-neu) und die Datenschutzgrundverordnung der EU (DSGVO) [DWWS18].

Das Ziel der **Integrität** (engl. *integrity*) besagt, dass keine unautorisierte und unbemerkte Manipulation von zu schützenden Daten passieren darf. In bestimmten Systemen ist eine solche Manipulation allerdings nicht verhinderbar, dann sind Maßnahmen nötig, die eine solche Manipulation erkennbar machen.

Das Ziel der **Authentizität** (engl. *authenticity*) besagt, dass es die Möglichkeit zu Bestätigung der Echtheit eines Objekts bzw. Subjekts geben muss. Diese Bestätigung kann durch eine eindeutige Identität und spezielle Eigenschaften erbracht werden (durch **Authentifikation**, engl *authentication*).

Oft ist die Möglichkeit der Authentifikation nur für Subjekte vorhanden. Allerdings müssen sich auch Objekte, wie WLAN-Access-Points, Web-Server und auch Programme authentifizieren und damit ihre Echtheit beweisen.

Das Ziel der **Verbindlichkeit** (engl. *non-repudiation*) besagt, dass eine Menge von Aktionen, welche ein bestimmtes Subjekt oder Objekt durchgeführt hat, im Nachhinein nicht bestreitbar ist. Hierbei ist oft eine Überwachung und Protokollierung von Aktivitäten nötig.

Das Ziel der **Verfügbarkeit** (engl. *availability*) besagt, dass authentifizierte und autorisierte Subjekte oder Objekte, bestimmte Handlungen durchführen können ohne dabei unautorisierte Beeinträchtigungen zu erfahren.

Ein sogenannter Denial-of-Service (*DoS*) Angriff stellt eine solche unautorisierte Beeinträchtigungen von autorisierten Aktionen der Subjekte und Objekte dar. Denn mithilfe eines DoS-Angriffs kann der Service oder die Verfügbarkeit eines IT-Systems eingeschränkt werden. Damit werden Aktionen von Subjekten unmöglich, z.B. durch Verzögerung.

Normale informationstechnische Verwaltungsmaßnahmen sind durch den Begriff der Verfügbarkeit nicht mit einbezogen. Bedeutet z.B. Prozess-Scheduling, also das Vergeben von Rechenzeit gehören nicht in die Angriffssparte auf die Verfügbarkeit. Allerdings verschwimmen hier die Grenzen, denn ein Angreifer könnte eben diese Verwaltungsmaßnahme manipulieren.

Das Ziel der **Anonymität** (engl. *anonymity*) besagt, dass die Durchführung von Handlungen ohne Preisgabe der Identität vollbracht werden muss. Es gibt auch die Möglichkeit der Anonymisierung. Die Anonymisierung von Daten macht den Aufwand der Deanonymisierung, also der Identitätsoffenlegung, unverhältnismäßig schwierig bis unmöglich. Dagegen steht die Pseudonymisierung, welche eine natürliche Person hinter einem Pseudonym versteckt.

In den folgenden Kapiteln wird auf Möglichkeiten eingegangen, welche bei der Erfüllung

dieser Ziele unterstützen. Zuerst wird mit der symmetrischen Verschlüsselung begonnen. Dabei wird auf die allgemeine Funktionsweise sowie Vor- und Nachteile eingegangen. Die kryptografischen Systeme und Algorithmen werden hier nur angerissen. Weiterführende Literatur über mathematische und formale Definitionen finden´ sich in [CJB10, Mar17].

2.4.3 Symmetrische Verschlüsselung

Die symmetrische Verschlüsselung bietet die Möglichkeit eine Kommunikation zu verschlüsseln, dabei wird der gleiche Schlüssel für die Vorgänge des Verschlüsselns und Entschlüsseln verwendet. Symmetrische Verschlüsselungsverfahren sind, gegenüber den asymmetrischen Verfahren, deutlich einfacher anzuwenden und benötigen weniger Berechnungszeit. Beispiel 7 erläutert den Ablauf einer verschlüsselten Kommunikation zwischen Alice und Bob.

Beispiel 7 *Alice generiert einen symmetrischen Schlüssel* `S`*. Diesen Schlüssel übergibt sie Bob an einem sicheren nicht überwachten völlig unbekanntem Ort. Bob will Alice nun über einen total unsicheren Kommunikationsweg, z.B. das Internet, eine wichtige Nachricht* `N` *zukommen lassen. Er nimmt den Schlüssel* `S` *und verschlüsselt damit die Nachricht* `N`*. Die verschlüsselte Nachricht* `E` *kann er nun, ohne Rücksicht auf Mithörer an Alice, übertragen. Alice empfängt die verschlüsselte Nachricht* `E` *und kann den Schlüssel* `S` *erneut anwenden, um* `E` *zu entschlüsseln. Schlussendlich kann Alice die Nachricht* `N` *lesen.*

Die Sicherheit von symmetrischer Verschlüsselung hängt allerdings nicht nur von der Stärke des kryptografischen Schlüssels ab, sondern auch von der Verwaltung und dem Austausch dieses Schlüssels. Zusätzlich muss für jeden Kommunikationspartner ein Schlüssel generiert werden, sonst könnten Andere mitlesen. Außerdem müssen sich die Kommunikationspartner gegenseitig Vertrauen, denn der Empfänger einer Nachricht, kann die gleichen Operationen durchführen, wie der Absender. Damit kann das Ziel der Verbindlichkeit unmöglich erreicht werden. Schlussendlich muss der generierte Schlüssel vor dem Start der verschlüsselten Kommunikation über ein sicheres Medium übertragen werden, dies stellt oft eine Herausforderung an sich dar.

2.4.4 Asymmetrische Verschlüsselung

Bei der asymmetrischen Verschlüsselung legt, gegenüber der symmetrischen, jeder Kommunikationspartner ein Schlüsselpaar an. Dieses Paar besteht aus dem privaten Schlüssel (engl. *private key*) und dem öffentlichen Schlüssel (engl. *public key*). Dabei hat der öffentliche Schlüssel die Aufgabe der Verschlüsselung eines Geheimnisses und der private Schlüssel die der Entschlüsselung. Anhand der Aufgaben und Namen wird deutlich, dass der private

Schlüssel besonders geschützt werden muss, wobei der öffentliche Schlüssel öffentlich sein darf. Der öffentliche Schlüssel kann öffentlich gemacht werden, weil die Eigenschaft gilt, dass der private Schlüssel nicht mit vertretbarem Aufwand aus dem öffentlichen Schlüssel errechnet werden könnte. Zur Verdeutlichung nachfolgend das Beispiel 8 mit Alice und Bob als Kommunikationspartner.

Beispiel 8 *Alice möchte Bob eine Nachricht* `N` *übersenden, ohne das ein Anderer die Nachricht lesen kann. Dafür fordert sie Bob auf, ein Schlüsselpaar zu generieren und ihr den öffentlichen Schlüssel* `Public` *zukommen zu lassen. Der Aufforderung kommt Bob nach. Dabei muss Bob nicht darauf achten, dass der Schlüssel in falsche Hände gerät. Alice verwendet den öffentlichen Schlüssel* `Public` *zur Verschlüsselung von* `N`*. Alice sendet nun die verschlüsselte Nachricht* `E` *über einen unsicheren Kommunikationskanal an Bob. Da nur Bob den privaten Schlüssel* `Private` *besitzt, kann ein Außenstehender nur die Kommunikation an sich verfolgen, allerdings den Inhalt nicht entschlüsseln. Bob nutzt seinen privaten Schlüssel* `Private`*, um die verschlüsselte Nachricht* `E` *zu entschlüsseln. Bob kann nun die Nachricht* `N` *lesen.*

Einerseits ist damit die Frage geklärt, wie die Kommunikationspartner ihren Schlüssel austauschen. Denn der öffentliche Schlüssel darf auch über ungeschützte Kommunikationskanäle verschickt werden. Allerdings entstehen hieraus neue Probleme. Denn ein Angreifer könnte als Man-in-the-Middle den öffentlichen Schlüsselaustausch abfangen, dem Absender seinen öffentlichen Schlüssel zuspielen und damit die verschlüsselten Geheimnisse des Absenders entschlüsseln und an den Empfänger weiterleiten.

Asymmetrische Verschlüsselung wird nicht nur als Kommunikationsverschlüsselung verwendet. Sie löst auch Probleme der symmetrischen Verschlüsselung, wie den Schlüsselaustausch über Verfahren wie *Diffie-Hellman-Key-Exchange* (DHKE) oder *Rivest-Shamir-Aldemann-Key-Transport* (RSA). Auch um das Ziel der Verbindlichkeit zu erfüllen, werden asymmetrische Verfahren verwendet. Darunter Beispiele wie Digital-Signature-Algorithm (DSA), Elliptic-Curve-Digital-Signature-Algorithm (ECDSA), RSA oder auch NTRU. Zusätzlich können mithilfe von sogenannten Challenge-Response Verfahren und Signaturen Kommunikationspartner eindeutig identifiziert werden.

Die asymmetrische Verschlüsselung benötigt deutlich längere Schlüssellängen und Berechnungszeit. Deshalb werden oft hybride Verfahren aus symmetrischer und asymmetrischer Verschlüsselung eingesetzt. Zum Beispiel beim bekannten Verfahren, welches für die Verschlüsselung des Transportwegs beim Hypertext-Transfer-Protocol (HTTP) zuständig ist, Transport-Layer-Security (TLS) bzw. Secure Socket Layer (SSL).

2.4.5 Kryptografische Hashfunktion

Kryptografische Hashfunktionen bilden digitale Fingerabdrücke von Dateiobjekten. Diese Fingerabdrücke können von jedem, mit Kenntnis des Algorithmus, überprüft werden und stellen somit die Integrität des Dateiobjektes bzw. der geteilten Information sicher. Damit soll die unautorisierte Modifikation des Dateiobjekts erkennbar gemacht werden.

Hashfunktionen bilden den unendlich großen Wertebereich der Dateiobjekte auf einen endlichen Hashbereich ab. Durch diese Abbildung kann es zu Kollisionen kommen, also unterschiedlichen Dateiobjekten, welche denselben Hashwert generieren. Diese Kollisionen sind für die Prüfung der Integrität und Manipulation ein großes Hindernis. Denn mithilfe von Kollisionen kann ein sogenannter Geburtstagsangriff (engl. *birthday-attack*) durchgeführt werden. Dieser beruht auf der Annahme des Geburtstags-Paradoxons und ermöglicht das Herausfinden eines Kollisionspaars auf der Annahme bestimmter Wahrscheinlichkeiten.

Die Konstruktion sicherer Hashfunktionen ist notwendig, um den Geburtstagsangriffe zu verhindern und das Zurückrechnen von Hashwerten zum eigentlichen Datenobjekt ineffizient zu gestalten. Deshalb basieren sichere Hashfunktionen auf sogenannten Einweg-Funktionen, wo die Berechnung des eigentlichen Wertes effizient ist. Die Umkehr dieser Berechnung ist mit dem aktuellen Stand der Technik nur in nicht annehmbarer Zeit möglich, also mitunter die Lebenszeit von Menschen übertrifft.

Hashfunktionen können einerseits auf symmetrischen Blockchiffren basieren. Dabei wird der letzte errechnete Hashwert als Schlüssel verwendet. Der erste Hashwert ist ein sogenannter Initialisierungsvektor (IV). Andererseits können auch dedizierte, also für das die Aufgabe des Hashs erfundene, Hashfunktionen verwendet werden. Der Secure-Hash-Algorithm (SHA) ist eines dieser speziell für das Hashen erfundene Verfahren.

Mit Hashwerten kann die Integrität von Dateiobjekten sichergestellt und Manipulationen sowie Modifikationen erkennbar gemacht werden. Allerdings kann nicht sichergestellt werden, dass die Nachricht trotz bewiesener Integrität auch vom gewollten Absender stammt, also die Authentizität. Dafür werden Hashfunktionen mit einem Geheimnis kombiniert. Diese Familie von Hashfunktionen wird auch Message Authentication Code (MAC) genannt.

2.4.6 Message Authentication Code

Um zusätzlich zur Integritätsprüfung durch die Hashfunktionen die Authentizität eines Datenobjektes zu bestätigen, werden häufig Hashfunktionen mit einem Geheimnis der Kommunikationspartner kombiniert. Diese Familie der Hashfunktionen heißt MAC. Durch dieses Geheimnis kann die Authentizität der Daten sichergestellt werden, also ist eine

eindeutige Zuordnung des Urhebers möglich. Allerdings ist keine Aussage über den Inhalt der Daten möglich. Beispiel 9 stellt solch eine Kommunikation zwischen *Alice* und *Bob* dar.

Beispiel 9 *Alice tauscht über einen sicheren Kanal einen Schlüssel K_{ab} mit Bob aus. Nun will Alice an Bob eine authentische Nachricht schicken. Dafür berechnet sie den MAC der Nachricht* `M`*:* `MAC(M, K_ab) = mac`*. Alice sendet die Nachricht* `M` *mit dem* `mac` *an Bob. Bob kann nun die Authentizität der Nachricht überprüfen, indem er selbst den MAC der empfangenen Nachricht* `M'` *berechnet:* `MAC(M', K_ab) = mac'`*. Stimmen die MACs der beiden Nachrichten überein, ist die übersendete Nachricht authentisch.*

MAC kann, wie Hashfunktionen, einerseits auf symmetrischen Blockchiffren oder dedizierten Hashfunktionen aufsetzen. Zum Beispiel gibt es MAC auf AES-Basis im CBC-Modus. Dedizierte Hashfunktionen mit Schlüssel werden auch als *Keyed-Hash* bezeichnet. Die verwendeten Schlüssel sollen Einfluss auf den IV der Hashfunktionen nehmen. Dafür wird der Schlüssel oft als Bestandteil der zu hashenden Daten verwendet. Durch dieses Verfahren können Angriffsszenarien entstehen, wie etwa wenn der Schlüssel als Präfix oder Suffix an die Nachricht gehangen wird. Denn dann kann der entstandene Hash als Basis für das Hashen weiterer Nachrichten verwendet und somit von einem Angreifer missbraucht werden. Um diese Angriffsszenarien auszuschließen wird das sogenannte HMAC-Verfahren eingesetzt. HMAC verschleiert den internen Zustand der Hashfunktion und erschwert das Wiederverwenden des Hashwertes für einen Angreifer.

Um die Authentizität eines Datenobjektes zu überprüfen, können MACs eingesetzt werden. Um analoge Signaturen digital zu ersetzen, sind bestimmte Anforderungen zu erfüllen. Mithilfe elektronischer Signaturen kann ein Rahmenwerk geschaffen werden, um diese Anforderungen zu erfüllen. Im folgenden Kapitel werden elektronische Signaturen erklärt.

2.4.7 Elektronische Signatur

An elektronische Signaturen werden bestimmte Anforderungen gestellt, um die Eigenschaften des analogen Vorgängers beizubehalten. Eine elektronische Signatur muss den Absender eindeutig identifizieren. Außerdem muss nachgewiesen werden können, dass ein digital signiertes Dokument dem Aussteller der Signatur vorgelegen hat und er es inhaltlich anerkennt. Darüber hinaus muss die Signatur den Inhalt des Dokuments als richtig und vollständig ausgeben. Schlussendlich muss durch die Signatur auch die rechtliche Bedeutung des Dokuments beim Empfänger bestätigt werden.

Diese Anforderungen sind nicht leicht zu erfüllen. Trotzdem gibt es einige Vorteile gegenüber dem analogen Pendant. Einerseits lassen sich elektronisch unterschriebene Dokumente zusätzlich verschlüsseln. Anderseits können sie mithilfe von MAC gegenüber Manipulationen geschützt werden und ein Zeitstempel kann die Gültigkeitsdauer der Signatur festlegen.

Verfahren für elektronische Signaturen können mithilfe von Eigenschaften existierender Verschlüsselungsfunktionen, also z.B. RSA, oder mit dedizierten Verfahren wie dem Digital Signature Algorithm (*DSA*) arbeiten. Beispiel 10 zeigt einen Kommunikationsablauf mithilfe einer durch RSA signierten Nachricht.

Beispiel 10 *Alice generiert ein Schlüsselpaar* $(Private_A, Public_A)$ *und hinterlegt den öffentlichen Schlüssel* $Public_A$ *in einer Schlüsseldatenbank. Nun möchte Alice an Bob eine signierte Nachricht* `M` *zukommen lassen. Sie signiert die Nachricht mithilfe der Entschlüsselungsfunktion* `D` *und dem privaten Schlüssel* $Private_A$*:* $D(M, Private_A) = sig$*. Anschließend sendet Alice die signierte Nachricht* `sig` *an Bob. Bob kann nun den öffentlichen Schlüssel von Alice* $Public_A$ *beziehen sowie die Verschlüsselungsfunktion verwenden, um damit die signierte Nachricht* `sig` *zu überprüfen:* $M = E(sig, Public_A)$*.*

Hierbei ist eindeutig ein Problem zu erkennen. Bob besitzt nicht das Wissen, ob Alice ihren öffentlichen Schlüssel in der Schlüsseldatenbank hinterlegt hat oder ein Dritter. Dabei kommen Lösungen, wie das *Web-of-Trust* (WoT), welches *Pretty-Good-Privacy* (PGP) als Grundlage nutzt, ins Spiel. Auf die Erklärung dieser Lösungen wird hier verzichtet.

In den vorherigen Kapiteln wurden die Grundlagen, Ziele und Möglichkeiten der Informationssicherheit dargelegt. Damit sind alle nötigen Grundlagen für das Verständnis der Arbeit erklärt worden. Im folgenden Kapitel 3 wird auf verwandte Arbeiten eingegangen.

3 Verwandte Arbeiten

In den folgenden Kapiteln werden verwandte Arbeiten, welche für die vorliegende Arbeit relevant sind, vorgestellt. Dabei wird eine breite Basis an Forschung untersucht, um auch folgenden Arbeiten eine Übersicht zu verschaffen.

Erst wird, in Kapitel 3.1, auf die vorherige Entwicklung und Entstehung des Analysewerkzeuges an der Universität Bremen eingegangen. Dafür werden vorangegangene Arbeiten kurz zusammengefasst und anschließend Erkenntnisse sowie Vorschläge aus den Ausblicken dieser gesammelt.

Anschließend werden, in Kapitel 3.2, verwandte Arbeiten, welche den Themenbereich vom Slicing im Kontext der Informationssicherheit abdecken, dargelegt. Dabei werden entstandene Werkzeuge betrachtet und eine kurze Zusammenfassung der Arbeiten gegeben.

3.1 Entwicklung des Slicers

In diesem Kapitel wird die Entwicklung des statischen Analysewerkzeuges anhand der Historie von Abschlussarbeiten an der Universität Bremen und des Technologie-Zentrum Informatik und Informationstechnik (TZI) chronologisch vorgestellt und dabei auf die Entstehung der zugehörigen Komponenten und Algorithmen, welche Bestandteil des vorhandenen Analysewerkzeuges sind, eingegangen.

[Gul14] untersuchte den Einfluss des Slicing auf das Programmverstehen von Android-Systemservices. Dabei wurden hauptsächlich Services in Betracht gezogen, welche Basisfunktionalität, wie z.B. Bluetoothverbindungen, für Androidapplikationen zur Verfügung stellen und gleichzeitig Berechtigungen überprüfen. Diese Basisservices wurden dann mithilfe des Slicing hinsichtlich sicherheitsrelevanter Prüfungen evaluiert. Dafür wurde ein erstes Werkzeug mithilfe der WALA-Bibliotheken entwickelt, welches zusätzlich zum Analyseergebnis Annotationen in der Java-Modeling-Language (JML) bereitstellt. Die Ergebnisse der Untersuchungen von [Gul14] zeigten, dass ein solcher Ansatz durchaus zur Durchführung von sicherheitsrelevanten Prüfungen geeignet ist. Zusätzlich stelle [Gul14] die Möglichkeit einer Ausweitung der Annotationen und eine mögliche Beweisführung, auf Grundlage dieser, in Ausblick.

[Ger15] erweiterte die Funktionalität des von [Gul14] entwickelten Werkzeuges um eine Möglichkeit zum Analysieren von Java-Enterprise-Edition (Java EE) und evaluierte es anschließend mithilfe von Enterprise-Java-Bean (EJB)-Anwendungen in Fallstudien. Dabei wurde festgestellt, dass im Ergebnis durchaus Zeilen vorhanden sind, welche keinen Einfluss

auf das Seed-Statement nehmen. Zusätzlich optimierte [Ger15] die von [Gul14] begonnene Exclusionfile.

[Det16] nahm Ideen und Erkenntnisse der vorangegangenen Arbeiten auf, entwickelte eine eigene Version des Werkzeuges und evaluierte allgemeine sicherheitsrelevante Java-Standard-Edition (Java SE) Programme zur Aufdeckung von Schwachstellen und Limitierungen der WALA-Bibliotheken. Festgestellt wurde außerdem, dass im Analyseergebnis oft unbeteiligte Objekte erschienen. Zusätzlich wurden die verschiedenen Möglichkeiten des Aufrufgraphen-Aufbaus als möglicher weiterer Untersuchungsgegenstand genannt.

[Ngu18] hat die vorangegangenen Arbeiten auf ihre Schwachpunkte bezüglich der Implementierung und Umsetzung untersucht. Mithilfe der gewonnenen Erkenntnisse wurde ein Kommandozeilen-Werkzeug geschaffen, welches ermöglicht, automatisierte Sicherheitsaudits mithilfe der Slicing-Technik durchführen zu lassen. Das entstandene Programm wurde durch Fallstudien auf Tauglichkeit geprüft. Außerdem stellte [Ngu18] eine verfeinerte Exlusionfile bereit und löste einige Implementierungsprobleme der Vorgänger, wie z.B. unerwünschte Objekte im Ergebnis und Verbesserungen der Rekonstruktion vom Quelltext. Das Auftreten unerwünschter Objekte wurde durch eine Objektverfolgung gelöst. Auch implementierte [Ngu18] eine erweiterte Einstiegspunktsuche auf Basis der Entwicklungen der vorherigen Arbeiten, WALA internen Algorithmen und eigener Implementierungen. Zusätzlich wurden die Ergebnisse in separaten Ordnern abgelegt, um die anschließende Analyse zu vereinfachen. Zusätzlich wurde die Möglichkeit implementiert, bestimmte Zwischenergebnisse, z.B. CG oder SDG, zu exportieren.

[Ker19] evaluierte das von [Ngu18] entwickelte Werkzeug und nutzte dafür Java-Programme, welche die Java Cryptography Extension (JCE), Java Cryptography Architecture (JCA), Google Tink und Apache Shiro verwenden. Zusätzlich wurde eine grafische Oberfläche zur Konfiguration des Werkzeuges entwickelt, da sich die Konfiguration des Werkzeugs mithilfe von Dateien als kompliziert und aufwendig herausstellte. Außerdem verbesserte [Ker19] die Wartbarkeit des Werkzeugs durch Umstrukturierungen des Quellcodes und machte es möglich, die Konfigurationsdatei auch im JSON-Format anzugeben.

[Cyl19] identifizierte den strukturellen Aufbau des entstandenen Werkzeuges als Hindernis. Einerseits wurde damit die Erweiterung der grafischen Oberfläche erschwert, andererseits war die Veränderung und Implementierung weiterer Funktionalität des Werkzeuges durch starke Koppelung des Quellcodes nur eingeschränkt möglich. Deshalb wurden die Implementierungen der Vorgänger in eine Server-Client-Architektur zusammengeführt, dies ermöglichte die getrennte Entwicklung von grafischen Oberflächen, schwächte die

Kopplung der Werkzeug-Komponenten und vereinfachte somit die Erweiterung sowie Fehlerbehebung. Diese erste Entwicklung bildet die Basis der vorliegenden Arbeit.

Zusätzlich verbesserte [Cyl19] die Rekonstruktion des Quellcodes aus dem Analyseergebnis, erweiterte die Suche der Einstiegspunkte und evaluierte das neu entstandene Analysewerkzeug auf seine Tauglichkeit in Bezug zu Android-Systemservices.

[Mö20] reaktivierte einige, beim Zusammenführen verloren gegangene, Funktionalitäten. Darunter die Möglichkeit, Java-Programme mithilfe des Werkzeuges zu analysieren. Zusätzlich wurde die Verfolgung von Objekten verbessert. [Mei21] implementierte Algorithmen der Vorgänger, welche durch die Umstrukturierung von [Cyl19] verloren gegangen waren, analysierte anschließend die neuen Implementierungen auf Tauglichkeit und wandelte Beispielprogramme vorheriger Arbeiten in Integrations-Tests um.

In sieben Jahren sind acht Arbeiten entstanden, die verschiedene Aspekte des Slicing mithilfe der WALA-Bibliotheken im Rahmen der Softwaresicherheit evaluieren, dabei auf vorherige Ergebnisse eingehen und neue Erkenntnisse, Vorschläge sowie Grundlagen für eine weitere Evaluation und Entwicklung aufbauen.

Zusammenfassend sind einige der Entwicklungen durch wiederholt erneute Implementierung verloren gegangen, doch Teile der Techniken und Erkenntnisse sind erhalten geblieben. Auch wurde in jeder Arbeit ein Ausblick formuliert, die dort entstandenen Ideen und Anregungen sollen folgend kurz zusammengefasst werden.

Einige der Arbeiten bemängelten, dass die Algorithmen zum Finden von Einstiegspunkten nicht ausgereift seien. Außerdem war die Dokumentation der Analyse-Werkzeuge im Verlauf der Zeit unvollständig und unverständlich, darunter fallen die Dokumentation im Werkzeug (z.B. JavaDoc) selbst, als auch die externen Beschreibungen zur Anwendung des Werkzeuges. Weitere schlugen weitreichendere Evaluierungen vor, darunter Programme mit über 200.000 Zeilen Quelltext, alle Kontroll- und Datenabhängigkeitsoptionen in verschiedenen Kombinationen sowie deren Auswirkung auf Laufzeit und Ergebnis, das Verwenden der GraalVM, eine Expertenstudie, um die Benutzbarkeit des Werkzeuges festzustellen und die Möglichkeit Java-EE Programme zu untersuchen. Auch wurde die Implementierung einer weiteren Analysebibliothek in Betracht gezogen, da WALA die Java Version beschränkt. Verschiedene Slicing-Varianten, wie beispielsweise das Forward-Slicing, Thin-Slicing oder Chopping, waren ebenfalls ein Vorschlag aus den vorherigen Arbeiten. Da das Werkzeug nur kompilierte Programme analysierte, wurde ein Buildserver vorgeschlagen, welcher Programme, die nur als Quelltext vorhanden sind, kompilieren soll. Die Seed-Statments waren im Werkzeug bisher nur per RegEx auswählbar und diese Auswahl beschränkte sich auf Methodenaufrufe. Da auch große Bibliotheken in analysierten Programmen verwendet wurden, kam der Vorschlag auf diese mit Hilfe von leeren Stubs

und Mockups zu ersetzen, um die Laufzeit zu verbessern. Stubs und Mockups sind leere Funktionskörper und Klassendefinitionen. Die Quelltextrekonstruktion war weiterhin unvollständig und wurde von einigen Arbeiten als Verbesserungsvorschlag aufgenommen. Bei der Quellcoderekonstruktion wurden zum Beispiel fehlende `import`-Anweisungen, innere sowie anonyme Klassen, Klammern bei `catch`-Blöcken oder Klassenattribute genannt. In fast allen Arbeiten wurde außerdem die Benutzbarkeit als Makel erkannt. Unter Benutzbarkeit wurden Punkte, wie das bessere Ausgeben des Ergebnisses, Ausgabe von verschiedenen Datenstrukturen des Prozesses (z.B. CG, CFG, SDG), Integration in eine IDE, Sortierung von Klassen bei der Anzeige des Ergebnisses und das Slicen von Quelltext, genannt. Schlussendlich wurde auch ein Dekompilierer vorgeschlagen, dieser könnte Programme, welche ohne Quelltext vorhanden sind, zum Analysieren vorbereiten.

Diese Zusammenfassung bietet nachfolgenden Arbeiten einen groben Überblick und hat zusätzlich Anhaltspunkte und Motivation für die in dieser Arbeit vorgenommenen Implementierungen gegeben.

3.2 Slicing im Kontext der Informationssicherheit

In diesem Kapitel wird ein aktueller Überblick über die Verwendung der Slicing-Technik im Kontext der Software-Sicherheit gegeben.

[CW07] bietet eine Übersicht von Anwendungen der statischen Analyse für die Entwicklung sicherer Software.

[Gra16] verwendete viele der existierenden Konstrukte und fügte sie zu einem Gesamtkonstrukt zusammen, dem JOANA-Tool. Dabei wurde die prototypische Implementierung von [HS09] als Basis verwendet. Auf Grundlage vorheriger Arbeiten wurde die Erzeugung von Abhängigkeitesgraphen für objektorientierte Sprachen verbessert, ein spezielles Parametermodell präsentiert und vorherige Erkenntnisse verwendet, um die Erzeugung des SDG zu beschleunigen. Entstanden ist das JOANA-Tool, welches Programme über 100.000 Quellcodezeilen untersuchen kann und bereits in verschiedenen Applikationen zum Einsatz kam. Unter die Anwendungen fallen z.B. die Analyse eines elektronischen Wahlsystems, die statische Codeanalyse für die Absicherung von Cordova Applikationen oder die Entwicklung eines Systems mit sicherem Informationsfluss, genannt IFlow [KSBR13, KFS+13, SKBR14].

[TSBB18, TSB15] wendet die Slicing-Technik im Kontext von Webanwendungen ein. Dabei wird überprüft, ob sogenannte *injection vulnerabilities* im untersuchten Quelltext vorhanden sind. [TSBB18, TSB15] nennt die entstehenden Slices *security slices*, welche deutlich kleiner sind, als vergleichbare Slices. Auch [WLG11] nutzt die Slicing-Technik

im Kontext von Webanwendungen. Allerdings konzentriert sich [WLG11] auf sogenannte Cross-Site Scripting (XSS) Attacken, dabei speziell *Stored XSS*. [FCKV10] stellt eine Möglichkeit zur automatischen Erkennung von Logikfehlern bei der Entwicklung von Webanwendungen vor, dabei wird die Slicing-Technik verwendet.

[EBFK13] untersucht die falsche Verwendung von kryptografischen APIs. Dabei werden Slices verwendet, um den zu untersuchenden Quelltext zu reduzieren. Für die Untersuchung wurden bestimmte Regeln aufgestellt, welche die Quelltexte erfüllen musste, damit eine fehlerhafte Verwendung ausgeschlossen wird. Darunter z. B. die Regel, dass der ECB-Modus nicht für die Verschlüsselung verwendet wird.

[MS08] stellt das sogenannte *confidentiality slicing criterion* vor, welches ein Seed-Statement darstellt. Dieses vereinfacht die Identifizierung von Fehlern in Programmen mit Sicherheitskontext ohne dabei die Geheimhaltung von Informationen zu verändern.

In [Cav08] wird das sogenannte *secure slicing* vorgestellt. Diese Slicing-Technik ermöglicht es ein Programm, welches Informationen über unsichere Kanäle durchsickern lässt, in ein sicheres Programm zu transformieren. Das entstehende Programm lässt keine Information über unsichere Kanäle durchsickern.

4 Erweiterungen

Diese Kapitel beschreibt und begründet die entstandenen möglichen Erweiterungen des Analysewerkzeuges und geht auf die Veränderungen des Quelltextes ein, um anschließend einen Überblick über die implementierten Erweiterungen, die Probleme bei der Entwicklung und spezielle Lösungsansätze zu geben.

Diese Arbeit dient als Grundlage für weitere Arbeiten zur Weiterentwicklung und Erforschung des . Diese Kapitel motiviert sich durch diese Anforderung und bietet eine Übersicht über die Erweiterungen, welche bei der Evaluierung des Analysewerkzeuges in Kapitel 5 zum Tragen kommen.

Die Kapitel 4.1 bis 4.4 geben Aufschluss über die umgesetzten Erweiterungen in den Bereichen Refactoring, Funktionalität und Benutzbarkeit, Slicing-Techniken und Rekonstruktion. Zusätzlich werden dabei Schwierigkeiten bei der Implementierung aufgezeigt und besondere Lösungsansätze aufgeschlüsselt und begründet.

4.1 Aktualisierung und Refactoring

Auch wenn die Verbesserung der Codequalität, das Aktualisieren von Softwareabhängigkeiten sowie die Erweiterung und Verbesserung der Dokumentation nicht direkt einen wissenschaftlichen Bezug haben, sind sie doch für diese Arbeit relevant. Denn diese Punkte wurden als Ausblicke verwandter Arbeiten aufgetan und auch Dr. Karsten Sohr hat die Wiederverwendung des entstandenen Quelltextes als Anforderung festgelegt.

Um das Analysewerkzeug aktuell zu halten wurde am Buildsystem einiges verändert. Dafür wurde der Frontend-Teil aus dem Gradle Bauprozess herausgenommen und verwendet nun das von Angular eigene Commandlineinterface (CLI). Für diesen Schritt wurde sich entschieden, damit keine aufwändige Webpack-Configuration gepflegt werden muss. Das Backend verwendet weiterhin den Gradle Bauprozess.

Anschließend wurden die Abhängigkeiten des Analysewerkzeuges aktualisiert. Dazu zählen die großen Bibliotheken wie Spring (Java-Framework), WALA (für das Slicing), JavaParser (für die Rekonstruktion), JUnit (für das Testen) und weitere. Im Frontend wurden Angular (Web-Frontend-Framework), PrimeNG (UI-Komponenten Bibliothek), einzelne Komponenten, deren Abhängigkeiten und die Buildwerkzeuge aktualisiert.

Das Analysewerkzeug wurde von [Cyl19] mithilfe von JHipster einem Framework zum Generieren von einfachen Java-Web-Anwendungen erzeugt. Nach dem Aktualisieren der Abhängigkeiten sind allerdings Unstimmigkeiten mit diesem Framework aufgetaucht, eine versuchte erneute Generierung zur Aktualisierung des generierten Quelltextes ist gescheі-

Tabelle 1: Versionen nach der Aktualisierung

Abhängigkeit	Version
Gradle Wrapper	7.3.3
Spring Boot	2.2.5
WALA	1.5.5
JavaParser	3.24.9
JUnit	0.13.1
Angular	13.3.7
PrimeNG	13.4.0
monaco-editor	0.33.0
bootstrap	5.1.3

tert. Deswegen wurde entschieden, die Abhängigkeit zu diesem Framework zu entfernen. Aufgrund der obsoleten Unterstützung-Bibliothek[2] des Frameworks wurden Teile neu implementiert, um die Aktualisierung abzuschließen.

Die Tabelle 1 gibt die Versionen der Abhängigkeiten nach der Aktualisierung an.

Der Aufbau des Projektes wurde stark verändert, da bei der Entwicklung festgestellt wurde, das viele der integrierten Werkzeuge und genutzten Funktionalitäten externer Abhängigkeiten keine Relevanz für die Entwicklung des Analysewerkzeuges haben. Dazu gehören z.B. End-2-End-Tests (ungenutzt), Aspekt-orientierte Programmierung (in Teilen ungenutzt), Pre-Commit-Hooks (ungenutzt), Codequalitätsanalysewerkzeuge (ungenutzt) und Containerunterstützung. Zudem haben sie das Verständnis des Quellcodes für den Autor dieser Arbeit erschwert.

Der Code im Front- sowie Backend wurde, wie in Abbildung 7 zu sehen, neu organisiert und größtenteils umbenannt.

Im gleichen Zuge wurde die Dokumentation ausgeweitet, um zukünftigen Entwicklern die Arbeit zu erleichtern. Dabei ist Dokumentation eine über die Verwendung für Benutzer sowie Entwickler entstanden. Das bezieht die Dokumentation über Konfigurationsdateien, die Struktur des Quelltextes im Front- sowie Backend und die Verwendung besonderer Funktionalität externer Abhängigkeiten, wie z. B. WALA oder dem JavaParser, mit ein.

Im Zuge der Entwicklung der gesamten Arbeit wurden 2.942 Dateien geändert, 32.492 Zeilen eingefügt und 537.991 Zeilen gelöscht[3]. Dabei wird erwähnt, dass diese Zeilen und Dateien nicht nur neue Funktionalität enthalten, sondern auch durch das allgemeine Refactoring entstanden sind.

[2] https://github.com/jhipster/ng-jhipster

[3] Ermittelt mit `git diff v1.0.0 --shortstat`

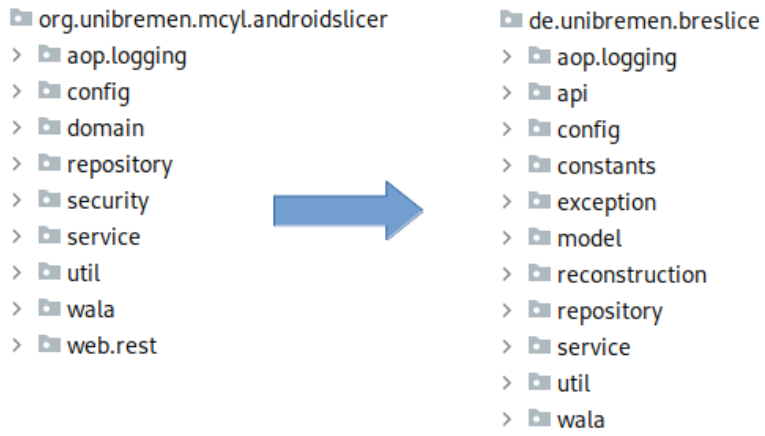


Abbildung 7: Refactoring der Java-Package Struktur

In diesem Kapitel wurde auf die Aktualisierung und die Refactorings im Zuge der Implementierungen eingegangen. Im folgenden soll auf die Implementierung der verschiedenen Slicing Techniken eingegangen werden.

4.2 Funktionalität und Benutzbarkeit

In diesem Kapitel soll auf die Erweiterungen des Analysewerkzeuges eingegangen werden, welche im Bezug auf Funktionalität und Benutzbarkeit getätigt wurden. Viele der genannten Erweiterungen können in beide der genannten Kategorien eingeordnet werden. Deswegen wird auf die Unterscheidung zwischen reiner Funktionalität oder reiner Benutzbarkeit verzichtet.

Um die Benutzbarkeit der grafischen Oberfläche zu verbessern, wurden nicht für das eigentliche Analysieren benötigte Abschnitte, wie Metriken, Healthchecks und bestimmte Konfigurationspunkte, entfernt. Stattdessen wurde das Programmmenü in folgende Punkte unterteilt.

1. Slicer
   (a) Create Slice
   (b) List Slice
2. Sources
   (a) List Sources
   (b) Upload New Sources

3. Configuration
    (a) Slicer Settings
    (b) Slicer Options

Unter dem Unterpunkt *Create Slice* kann ein neuer Slicing-Auftrag erstellt werden. Dabei wurde die Auswahl der in Kapitel 4.3 implementierten Slicing-Techniken hinzugefügt und für die jeweilige Technikauswahl ändert sich die Maske für die Eingabe der Seed-Statements. Der Unterpunkt *List Slice* bietet einen Überblick über vorherige Ergebnisse des Analysewerkzeuges und auf Wunsch kann hierüber eine Detailansicht eines bestimmten Ergebnisses aufgerufen werden.

Unter dem Punkt *Sources*, welcher vorher als Popup-Fenster implementiert war, sind nun zwei weitere Unterpunkte untergebracht. Zum einen der Punkt *List Sources* unter welchem alle sich momentan auf dem Dateisystem befindlichen Programmquelltexte angezeigt werden. Zum anderen der Punkt *Upload New Sources*, welcher es ermöglicht zusätzlichen Programmquelltext nachzuladen.

Der Punkt *Configuration* beinhaltet zum einen die allgemeinen Einstellungen des Analysewerkzeuges unter *Slicer Settings*, wie z. B. Ablageort der Programmquelltexte, Ablageort der Ergebnisse oder ob die Ergebnisse überhaupt gespeichert werden sollen. Der Unterpunkt *Slicer Options* bietet die Möglichkeit per Hand neue Optionen, welche im WALA-Framework implementiert wurden, anzulegen. Darunter fallen die Datenabhängigkeitsoptionen oder Kontrollabhängigkeitsoptionen.

Die grafische Oberfläche für die Detailansicht der einzelnen Ergebnisse wurde auch angepasst. Bestimmte Punkte des Ergebnisses wurden gegliedert, um die Seite übersichtlicher zu gestalten und dem Nutzer das Ausblenden von für ihn nicht nützlicher Information zu ermöglichen, wie dem eigentlichen Slicing-Auftrag mit allen Optionen, Parametern und Kriterien oder das Protokoll des Slicing-Vorgangs.

Das Protokoll des Slicing-Vorgangs ist nun detaillierter. Die Berechnungszeit der einzelnen Phasen des Vorgangs, wie beispielsweise CG-Konstruktion oder Slicing, wird angezeigt und für die einzelnen Phasen gibt es mehr Informationen, unter anderem welche Eintrittspunkte durch die gegebenen Einschränkungen infrage kamen.

Um die Ergebnisse einfacher Analysieren zu können, wurde außerdem die Ausgabe verändert. Die rekonstruierten Klassen werden in ihrer ursprünglichen Java-Paketstruktur abgelegt, damit eine Findung der originalen Dateien einfacher gelingt. Zusätzlich wird der SDG und CG als dot-Notation ausgegeben, um mögliche Ungenauigkeiten erkennen zu können und die nachfolgende Analysemöglichkeiten zu erweitern. Schlussendlich finden sich

in der Ausgabe auch der eigentliche Slicing-Auftrag mit Angabe der Optionen, Parameter und Kriterien, welche zum Ergebnis geführt haben und die für das Ergebnis relevanten Daten, welche die analysierten Klassen mit Quellcodezeilenangaben, die schlussendlich gewählten Seed-Statements und Berechnungszeit enthalten.

Die manuelle Eingabe über die Maske der grafischen Oberfläche, um viele verschiedene Programme mit verschiedenen Einstellungen zu analysieren, ist durchaus langsam. Aus diesem Grund wurde ein neuer API-Endpunkt unter `/process` geschaffen, dieser liest Slicing-Aufträge aus einem bestimmten Ablageort ein und verarbeitet sie. Dies ermöglicht eine Menge von verschiedenen Parametern auf eine Menge von unterschiedlichen Programmen wiederholbar auszuprobieren.

4.3 Slicing Techniken

In diesem Kapitel wird auf die Erweiterung des Analysewerkzeuges mit weiteren Slicing-Techniken eingegangen. Das Analysewerkzeug unterstützte vor den Implementierungen dieser Arbeit das Backward-Slicing. Das WALA-Framework unterstützt allerdings nicht nur Backward-Slicing, sondern auch Forward-Slicing und Thin-Backward-Slicing.

Um die Funktionalität umzusetzen, musste in der grafischen Oberfläche eine Unterscheidung zwischen den verschiedenen Techniken umgesetzt werden. Der Server muss diese Unterscheidung unterstützen und das WALA-Framework muss korrekt parametrisiert aufgerufen werden. Das Backward-Slicing in WALA unterstützt die Übergabe einer Menge von Seed-Statements. Das Forward-Slicing nicht, dadurch musste das Forward-Slicing für jedes Statement einzeln ausgeführt und anschließend die entstandenen Programmteile zu einem zusammengeführt werden. Das Thin-Backward-Slicing ist in WALA durch eine Klasse (`ThinSlicer`) bereitgestellt.

Für das Chopping wurde die Möglichkeit zum direkten Auswählen von Seed-Statements gegeben. Das Analysewerkzeug bot nur die Möglichkeit Seed-Statements anhand von RegEx-Ausdrücken zu finden und die Suche war auf Methodenaufrufe beschränkt. Da im Chopping genau zwei Seed-Statements benötigt werden, musste die Möglichkeit zur direkten Auswahl von Seed-Statements geschaffen werden. Ein Problem hierbei ist, dass in einer Zeile natürlich mehrere Programmstatements vorkommen können. Zum Beispiel ist in im Codeauszug 7 eine Zeile zu sehen, in der mehrere Statements vorkommen. Als Erstes wird die Klasse `System` referenziert, anschließend ist ein statischer Aufruf des Attributes `out` zu vermerken und danach findet der Methodenaufruf von `println` statt. Diese Besonderheit musste bei der Auswahl beachtet werden.

```
System.out.println("Hello, World!");
```

Codeauszug 7: Zeile mit mehreren Statements

Ein weiteres Problem bei der Implementierung des Choppings war, das WALA die Technik nicht direkt unterstützt. [Kri04] schreibt im Kapitel 10.2.1 dazu: „In the intraprocedural or context-insensitive case a chop for a chopping criterion (s, t) can basically computed by the intersection of a backward slice for t with a forward slice for s: $C(s, t) = SF(s) \cap SB(t)$“. Das bedeutet im Analysewerkzeug wurde Chopping als eine Kombination des Forward- und Backward-Slicing implementiert.

Allerdings erwähnt [Kri04] zusätzlich, dass der Algorithmus keine Schnittmenge des Forward-Slicing und Backward-Slicing Ergebnis bildet, sondern mithilfe einer anderen Herangehensweise in zwei Phasen das Ergebnis bekommt. In der ersten Phase wird das Backward-Slicing durchgeführt, um anschließend auf den besuchten Knoten des SDG das Forward-Slicing durchzuführen. Diesen Algorithmus nennt [Kri04] context-insensitive chopping (CIC). Diese Möglichkeit der Implementierung von CIC wurde, aufgrund der API des WALA-Frameworks, nicht behandelt.

4.4 Rekonstruktion

In diesem Kapitel werden die Verbesserung der Rekonstruktion von Quelltext anhand eines Beispiels dargelegt. Die Rekonstruktion des eigentlichen Quelltextes aus dem Ergebnis von WALA ist essenziell für die anschließende Analyse.

Im Folgenden soll anhand eines Beispielprogramms, welches einen String mithilfe von AES verschlüsselt, die Verbesserung der Rekonstruktion von Quelltext dargelegt werden. In Codeauszug 8 und Codeauszug 9 sind der Eintrittspunkt mit der `main`-Methode der Klasse `AESEncryptDecrypt` und die verwendete Klasse `AES` der zu untersuchenden Bibliothek zu sehen.

Für eine bessere Lesbarkeit wurde auf `import-` und `package-`Anweisungen verzichtet, diese werden in beiden Fällen korrekt rekonstruiert.

Im Codeauszug 8 wird in Zeile 3 ein Objekt vom Typ `AES` erzeugt. Anschließend wird die Nachricht aus Zeile 5 in Zeile 6 verschlüsselt, um direkt danach in Zeile 9 wieder entschlüsselt zu werden.

```
1 final String secretKey = "ThisIsATopSecretNeverToBeLostKey";
2
3 AES aesA = new AES(secretKey);
4
```

```
5  String originalMsg = "ThisIsATopSecretMessageNeverToBeDecrypted.";
6  String encrypted = aesA.encrypt(originalMsg);
7  System.out.println(encrypted);
8
9  String decrypted = aesA.decrypt(encrypted);
10 System.out.println(decrypted);
```

Codeauszug 8: `main`-Methode der Klasse `AESEncryptDecrypt`

Im Codeauszug 9 existiert die Klasse `AES`, welche die innere Klasse `KeyHolder` besitzt. Die innere Klasse bringt den Schlüssel in ein spezifiziertes Format. Zusätzlich besitzt die Klasse `AES` zwei Methoden `encrypt` und `decrypt`, welche für das Ver- und Entschlüsseln des eingegebenem String zuständig sind. Beim Ver- und Entschlüsseln wird derselbe Schlüssel aus dem `KeyHolder` verwendet. Beim Verschlüsseln wird der verschlüsselte String nachträglich als Base64 kodiert und beim Entschlüssel wird der String vorher aus Base64 dekodiert.

```
1  public class AES implements ICryptor {
2    private KeyHolder keyHolder;
3
4    public AES(String secretKey) throws NoSuchAlgorithmException {
5      this.keyHolder = new KeyHolder(secretKey);
6    }
7
8    public class KeyHolder {
9      private final SecretKeySpec secretKey;
10
11     protected SecretKeySpec getSecretKey() {
12       return this.secretKey;
13     }
14
15     public KeyHolder(String keyString) throws NoSuchAlgorithmException {
16       MessageDigest sha = null;
17
18       byte[] keyBytes = keyString.getBytes(StandardCharsets.UTF_8);
19
20       // create sha for key
21       sha = MessageDigest.getInstance("SHA-1");
22       keyBytes = sha.digest(keyBytes);
```

```
23
24        // truncate to 16 bytes
25        keyBytes = Arrays.copyOf(keyBytes, 16);
26        secretKey = new SecretKeySpec(keyBytes, "AES");
27      }
28    }
29
30    @Override
31    public String encrypt(final String strToEncrypt) {
32      try {
33        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
34        cipher.init(Cipher.ENCRYPT_MODE, this.keyHolder.getSecretKey());
35        return Base64.getEncoder()
36        .encodeToString(cipher.doFinal(strToEncrypt.getBytes(StandardCharsets↩
              .UTF_8)));
37      } catch (Exception e) {
38        System.out.println("Error while encrypting: " + e);
39      }
40
41      return null;
42    }
43
44    @Override
45    public String decrypt(final String strToDecrypt) {
46      try {
47        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5PADDING");
48        cipher.init(Cipher.DECRYPT_MODE, this.keyHolder.getSecretKey());
49        return new String(cipher.doFinal(Base64.getDecoder()
50        .decode(strToDecrypt)));
51      } catch (Exception e) {
52        System.out.println("Error while decrypting: " + e);
53      }
54
55      return null;
56    }
57  }
```

Codeauszug 9: AES-Klasse

Der Quelltext wird mit dem Analysewerkzeug untersucht. Dabei wird als Seed-Statement der `encrypt`-Aufruf aus Zeile 6 des Codeauszug 8 verwendet und als Eintrittspunkt die `main`-Methode der `AESEncryptDecrypt`-Klasse gewählt. Das Analysewerkzeug wurde mit folgenden Parametern ausgeführt, als Slicing-Technik Forward-Slicing, als CFA-Constraint `ZERO_ONE_CFA`, als Datenabhängigkeitsoption `NO_EXCEPTIONS` und als Kontrollabhängigkeitsoptionen `NO_EXCEPTIONAL_EDGES`. Diese Einstellungen wurden in vorherigen Arbeiten aufgrund eines guten Verhältnisses zwischen Ergebnis und Berechnungszeit als Standardeinstellungen festgelegt.

In den Codeauszügen 10 und 11 ist das Rekonstruktionsergebnis vor den Implementierungen der Arbeit zu sehen. Hierbei ist zu beobachten, dass die Definitionen der verschlüsselten sowie entschlüsselten Nachricht enthalten sind, nicht jedoch die des Verschlüsselungsobjektes, auf denen die Aufrufe der Methoden `encrypt` und `decrypt` stattfinden.

```
1 public class AESEncryptDecrypt {
2   public static void main(String[] args) throws NoSuchAlgorithmException {
3     String encrypted = aesA.encrypt(originalMsg);
4     System.out.println(encrypted);
5     String decrypted = aesA.decrypt(encrypted);
6     System.out.println(decrypted);
7   }
8 }
```

Codeauszug 10: Klasse mit Eintrittspunkt (rekonstruiert aus Ergebnis vor der Verbesserung)

Auch im Codeauszug 11 sind fehlende Anweisungen zu erkennen. Es fehlen Informationen, um herauszufinden von welchem Typ das Klassenattribut `keyHolder` ist und welche Anweisungen die Methode `getSecretKey` enthält. Auch fehlt die zweite `return`-Anweisung der beiden Methoden `encrypt` und `decrypt`.

Der Inhalt des `catch`-Blocks der `try`-Anweisung fehlt aufgrund der Datenabhängigkeitsoption `NO_EXCEPTIONS` und würde mit der Option `FULL` im Ergebnis vorhanden sein und anschließend korrekt rekonstruiert werden.

```
1 public class AES implements ICryptor {
2   @Override
3   public String encrypt(final String strToEncrypt) {
4     try {
5       Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
6       cipher.init(Cipher.ENCRYPT_MODE, this.keyHolder.getSecretKey());
```

```
7       return Base64.getEncoder()
8       .encodeToString(cipher.doFinal(strToEncrypt.getBytes(StandardCharsets↩
            .UTF_8)));
9     } catch (Exception e) {
10    }
11  }
12
13  @Override
14  public String decrypt(final String strToDecrypt) {
15    try {
16      Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5PADDING");
17      cipher.init(Cipher.DECRYPT_MODE, this.keyHolder.getSecretKey());
18      return new String(cipher.doFinal(Base64.getDecoder()
19      .decode(strToDecrypt)));
20    } catch (Exception e) {
21    }
22  }
23 }
```

Codeauszug 11: AES-Klasse rekonstruiert aus Ergebnis vor der Verbesserung

In den Codeauszügen 12 und 13 ist der rekonstruierte Quelltext nach den Implementierungen der Arbeit zu sehen. In diesem Fall ist der Konstruktoraufruf in Zeile 3 enthalten und gibt damit Aufschluss, welche Klasse verwendet wurde.

```
1 public class AESEncryptDecrypt {
2   public static void main(String[] args) throws NoSuchAlgorithmException {
3     AES aesA = new AES(secretKey);
4     String encrypted = aesA.encrypt(originalMsg);
5     System.out.println(encrypted);
6     String decrypted = aesA.decrypt(encrypted);
7     System.out.println(decrypted);
8   }
9 }
```

Codeauszug 12: Klasse AESEncryptDecrypt rekonstruiert nach Verbesserung

Im Codeauszug 13 ist nun die Klasse `KeyHolder` und die Methode `getSecretKey` enthalten. In diesem Fall ist direkt zu erkennen, woher der Schlüssel kommt und dass für das Ver- und Entschlüsseln derselbe Schlüssel verwendet wird.

```
1  public class AES implements ICryptor {
2    private KeyHolder keyHolder;
3    public class KeyHolder {
4      private final SecretKeySpec secretKey;
5      protected SecretKeySpec getSecretKey() {
6        return this.secretKey;
7      }
8    }
9
10   @Override
11   public String encrypt(final String strToEncrypt) {
12     try {
13       Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
14       cipher.init(Cipher.ENCRYPT_MODE, this.keyHolder.getSecretKey());
15       return Base64.getEncoder()
16       .encodeToString(cipher.doFinal(strToEncrypt.getBytes(StandardCharsets↩
             .UTF_8)));
17     } catch (Exception e) {
18     }
19   }
20
21   @Override
22   public String decrypt(final String strToDecrypt) {
23     try {
24       Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5PADDING");
25       cipher.init(Cipher.DECRYPT_MODE, this.keyHolder.getSecretKey());
26       return new String(cipher.doFinal(Base64.getDecoder()
27       .decode(strToDecrypt)));
28     } catch (Exception e) {
29     }
30   }
31 }
```

Codeauszug 13: Klasse AES rekonstruiert nach Verbesserung

In diesem Kapitel wurde anhand eines Beispiels gezeigt, welchen Einfluss die Rekonstruktion des Quellcodes auf die spätere Analyse des Ergebnisses hat und welche Rekonstruktionen im Zuge der Arbeit implementiert wurden. Damit sind alle Erweiterungen für das Analysewerkzeug, welche im Laufe der Arbeit entstanden sind, behandelt worden. Im

folgenden Kapitel 5 wird auf die Evaluation des Analysewerkzeuges durch Benchmarks und den Fallstudien eingegangen.

5 Evaluierungen

In diesem Kapitel werden die Evaluierungen des Analysewerkzeuges und deren Ergebnisse festgehalten. Dabei wird auch auf mögliche Interessenkonflikte eingegangen und Besonderheiten einzelner Evaluierungen werden hervorgehoben. Eine Diskussion findet im nachfolgenden Kapitel 6 statt.

Im Kapitel 5.1 geht es um die Evaluierung der Berechnungszeit. Dabei werden verschiedene Einstellungen der JVM und als Alternative die GraalVM untersucht. Die GraalVM ist eine Implementierung der JVM, bietet darüber hinaus besondere Funktionalität und soll eine bessere Ausführungszeit als die der JVM des OpenJDK besitzen.

Das Kapitel 5.2 stellt die Ergebnisse einzelner Fallstudien dar, dabei wurden im Rahmen der Arbeit entwickelte Programme, welche die Kryptografiefunktionen von Java verwenden, und die Beispielprogramme der Online Services Computer Interface (OSCI)-Bibliothek durch das Analysewerkzeug untersucht. Die OSCI-Bibliothek bietet Funktionalität für die sichere Kommunikation. Die Bibliothek wird auch in Bereichen der Behördenkommunikation eingesetzt. Sie basiert auf den OSCI-Standard und dient als Referenzimplementierung.

5.1 Benchmark

Diese Kapitel stellt die Evaluierungsergebnisse hinsichtlich der Berechnungszeit des Analyseergebnisses für verschiedene Programme mit vordefinierten Parametern der JVM oder GraalVM vor. Dabei werden die Umgebung in der die Evaluierung stattfand, die Menge der verschiedenen Kombinationen von Optionen, Parametern und Kriterien und schlussendlich die zeitlichen Ergebnisse festgelegt.

Die Evaluierung der Berechnungszeit fand auf einem System mit Debian 11 als Betriebssystem, i5 mit 4 Kernen, 16GB RAM und 32GB an Swapspeicher statt. Um die Umgebung der Java Virtual Machine (JVM) möglichst identisch zu halten, wurden bestimmte Parameter beim Start übergeben. Dabei wurde der minimal verfügbare Speicher mithilfe der Option `-Xms2G` auf 2GB beschränkt und der maximal verfügbare Speicher mithilfe der Option `-Xmx8G` auf 8GB.

Für die Evaluierung wurden 15 verschiedene Programme mit mindestens den Standardeinstellungen und allen 4 Slicing-Techniken evaluiert. Die Standardeinstellungen wurden in [Cyl19] festgelegt und sind `NO_EXCEPTIONAL_EDGES` für die Kontrollabhängigkeiten, `NO_HEAP` als Datenabhängigkeitsoptionen, `NONE` für Reflections und `ZERO_ONE_CFA` als CFA-Constraint.

Um die Berechnungszeiten einordnen zu können sind in Tabelle 2 die jeweiligen Program-

Programmname	Quellcodezeilen	Funktionen	Klassen
ArithmeticTest	17	1	1
DeepCallee	27	3	1
MultipleCallee	22	4	4
Kryptografie Beispiele	212	16	22
OSCI-Beispiele	12.628	1289	160

Tabelle 2: Quellcodegröße der Programmbeispiele

me, ihre Quellcodezeilenanzahl, Funktionsanzahl und Klassenanzahl aufgelistet[4]. Einige der Beispiele wurden zusammengefasst. Der Quelltext der Programme `DeepCallee` und `MultipleCallee` ist in Anhang A.1.2 und respektive in Anhang A.1.3 zu finden.

Jedes Programm wurde, mit der entsprechenden Slicing-Technik, 3 mal analysiert um zufällige Verlängerungen und Verkürzungen der Berechnungszeit auszuschließen. Diese drei Werte wurden zu einem Durchschnittswert zusammengefasst. In Tabelle 3 sind die getesteten JVM Optionen aufgelistet. Zusätzlich zu den getesteten Optionen wurde GraalVM als mögliche Alternative evaluiert. Die meisten JVM Optionen haben Einfluss auf den verwendeten Garbage-Collector, welcher sich um das Freigeben von Speicher kümmert. Der Speicher wird freigegeben, falls dieser von unbenutzten Objekten verwendet wird. In der JVM ist die Garbage Collection mithilfe eines Generationsmodells aufgebaut. Objekte die eine bestimmte Anzahl an Collectonphases des Garbage-Collectors in Benutzung bleiben, werden markiert. Diese Markierung sagt aus, das ein Objekt nicht oft überprüft werden muss, da es wahrscheinlich länger in Benutzung bleibt, also einer älteren Generation angehört. In der JVM gibt es mehrere Generationen, welche in verschiedenen Intervallen geprüft werden.

In allen folgenden Tabellen dieses Kapitels sind die besten Berechnungszeiten markiert. In Tabelle 4 sind die Berechnungszeiten der Standardeinstellungen aufgelistet. Dabei wurden die Slicing-Techniken abgekürzt. Backward-Slicing wird mit Ba, Chopping wird mit Ch, Forward-Slicing wird mit Fo und Thin-Backward-Slicing wird mit Tb abgekürzt. Die Beispiele aus [Ngu18] und dieser Arbeit zeigen klar, dass dort die Standardeinstellungen am besten für eine kurze Berechnungszeit der Slice-Ergebnisse sind.

Bei den Beispielen der OSCI-Bibliothek sind auch einige der kürzesten Berechnungszeiten bei der JVM mit den Standardeinstellungen. Allerdings haben hier vermehrt andere JVM Einstellungen, wie in den Tabellen 6, 7, 9, 10 und 11 die kürzesten Berechnungszeiten erbracht.

[4]Ermittelt mithilfe von JavaNCSS

JVM-Option	Beschreibung
Keine	Standardeinstellungen der JVM
-XX:+ScavengeBeforeFullGC	Durchlauf jüngerer Generationen bevor kompletter Speicher überprüft wird
-XX:+UseConcMarkSweepGC	Verwendet den Concurrent Mark Sweep Algorithmus für die alte Generation
-XX:+UseParallelGC	Paralleler Garbage-Collector für Scavenges (Überprüfung der jungen Generation)
-XX:+UseParallelOldGc	Paralleler Garbage-Collector für alle Generationen
-XX:+UseSerialGC	Serieller Garbage-Collector
-XX:+UseG1GC	Der Garbage First Garbage-Collector der JVM

Tabelle 3: Untersuchte JVM Optionen

Wenn z. B. der Concurrent Mark Sweep (CMS)-Algorithmus für die alte Generation verwendet wird, dann scheint es, ohne besonderes Muster, auch kürzere Berechnungszeiten zu geben. Allerdings sind diese Ergebnisse signifikant schneller als die Standardeinstellungen der JVM, z. B. ist beim Beispielprogramm `AuthorOriginatorContentInterchange` die Berechnung des Thin-Backward-Slicing ca. 50% schneller. Diese Ergebnisse können deshalb möglicherweise auch durch Zufälle bei den Intervallen des Garbage-Collectors entstanden sein. Zusätzlich sind keine speziellen Muster zu erkennen, bei welchen der CMS-Algorithmus die Berechnungszeit dauerhaft verkürzen würde. Auch der serielle Garbage-Collector zeigt ähnliche Ergebnisse, die ohne Muster schwer auf bestimmte Fälle anzuwenden sind.

Die Ergebnisse der Berechnungszeiten für die GraalVM aus Tabelle 5 geben keinen Aufschluss über eine mögliche Alternative zur JVM. Keine der gemessenen Berechnungszeiten ist kürzer als die der Standardeinstellungen. Dies trifft auch auf die gemessenen Berechnungszeiten der Einstellung `-XX:+UseParallelGC` in Tabelle 8 zu.

Auch die Ergebnisse anderen Einstellungen `-XX:+ScavengeBeforeFullGC` in Tabelle 7, `-XX:+UseParallelOldGC` in Tabelle 9 sowie `-XX:+UseG1GC` in Tabelle 11 lassen keine Muster erkennen, die zu einem Schluss kommen würden, dass bestimmte Einstellungen dauerhaft kürzere Berechnungszeiten ergeben.

Programmname	Ba	Ch	Fo	Tb
ArithmeticTest	**246** ms	**221** ms	**217** ms	**215** ms
DeepCallee	**217** ms	**220** ms	**207** ms	**203** ms
MultipleCallee	202 ms	**207** ms	208 ms	**202** ms
AESEncryptDecrypt	**337** ms	**340** ms	**360** ms	**316** ms
DigestVerify	**253** ms	**280** ms	**458** ms	**253** ms
FileEncryption	**575** ms	**291** ms	**319** ms	**279** ms
SignatureVerify	**271** ms	**294** ms	**273** ms	**274** ms
SignEncryptDecrypt	**651** ms	**698** ms	**575** ms	**623** ms
AuthorOriginatorContentInterchange	**2220** ms	2410 ms	**9499** ms	2190 ms
MultiFetchDelivery	**2418** ms	**2410** ms	8790 ms	2310 ms
OneWayMessage_ActiveRecipient	**2446** ms	**2494** ms	**8134** ms	2272 ms
OneWayMessage_PassiveRecipient	**2197** ms	2323 ms	6784 ms	**1362** ms
PartialOneWayMessage_ActiveRecipient	2345 ms	2661 ms	7564 ms	2917 ms
RequestResponse	2727 ms	2749 ms	9074 ms	2555 ms
SendEncryptedContent	2605 ms	2444 ms	9416 ms	2428 ms

Tabelle 4: Ergebnisse der Berechnungszeit der JVM mit Standardeinstellungen

Programmname	Ba	Ch	Fo	Tb
ArithmeticTest	432 ms	351 ms	286 ms	332 ms
DeepCallee	274 ms	325 ms	364 ms	371 ms
MultipleCallee	293 ms	418 ms	371 ms	300 ms
AESEncryptDecrypt	808 ms	661 ms	761 ms	579 ms
DigestVerify	388 ms	384 ms	761 ms	579 ms
FileEncryption	809 ms	426 ms	457 ms	409 ms
SignatureVerify	361 ms	414 ms	403 ms	554 ms
SignEncryptDecrypt	1021 ms	1026 ms	681 ms	684 ms
AuthorOriginatorContentInterchange	3495 ms	3164 ms	12876 ms	2405 ms
MultiFetchDelivery	2982 ms	2957 ms	11986 ms	2534 ms
OneWayMessage_ActiveRecipient	2694 ms	2755 ms	10864 ms	2535 ms
OneWayMessage_PassiveRecipient	2490 ms	2464 ms	9082 ms	2335 ms
PartialOneWayMessage_ActiveRecipient	2688 ms	2868 ms	8433 ms	2785 ms
RequestResponse	2612 ms	2594 ms	11251 ms	2344 ms
SendEncryptedContent	2433 ms	2514 ms	11119 ms	2452 ms

Tabelle 5: Ergebnisse der Berechnungszeit unter Verwendung der GraalVM

Programmname	Ba	Ch	Fo	Tb
ArithmeticTest	572 ms	388 ms	325 ms	330 ms
DeepCallee	342 ms	352 ms	317 ms	316 ms
MultipleCallee	291 ms	313 ms	294 ms	303 ms
AESEncryptDecrypt	680 ms	633 ms	591 ms	560 ms
DigestVerify	430 ms	443 ms	706 ms	370 ms
FileEncryption	844 ms	449 ms	509 ms	437 ms
SignatureVerify	415 ms	449 ms	422 ms	412 ms
SignEncryptDecrypt	342 ms	352 ms	317 ms	316 ms
AuthorOriginatorContentInterchange	3488 ms	3563 ms	14685 ms	**1093** ms
MultiFetchDelivery	3523 ms	3554 ms	13447 ms	3314 ms
OneWayMessage_ActiveRecipient	3497 ms	3580 ms	13082 ms	3390 ms
OneWayMessage_PassiveRecipient	3131 ms	3246 ms	10310 ms	3139 ms
PartialOneWayMessage_ActiveRecipient	5394 ms	4131 ms	9113 ms	2590 ms
RequestResponse	2396 ms	2430 ms	**7106** ms	**2131** ms
SendEncryptedContent	**2196** ms	2393 ms	**5896** ms	2205 ms

Tabelle 6: Ergebnisse der Berechnungszeit mit `-XX:+UseConcMarkSweepGC`

5.2 Fallstudie

In diesem Kapitel werden die Ergebnisse der Fallstudie aufbereitet und erläutert. Um das Analysewerkzeug und die im Rahmen der Arbeit implementierten Funktionen, speziell die verschiedenen Slicing-Techniken, und ihren Einfluss auf die Analyseergebnisse zu beurteilen werden in diesem Kapitel verschiedene Beispielprogramme untersucht.

Das Kapitel ist dabei so aufgebaut, dass erst das unmodifizierte Programm erklärt wird und anschließend auf die Ergebnisse des Slicing mit den verschiedenen Techniken eingegangen wird. Dabei werden Besonderheiten oder für die Analyse interessante Stellen hervorgehoben. Eine Diskussion der Evaluierungsergebnisse findet im nachfolgenden Kapitel 6 statt.

Die Evaluierungen zeigen auf inwiefern Fragen über das Programmverständnis mithilfe des Analysewerkzeuges beantwortet werden können. Dabei wird speziell auf Fragen der Informationssicherheit eingegangen. Beispielsweise wird erforscht, ob mithilfe des Analyseergebnisses fragen über die verwendeten Algorithmen gemacht werden können, aber auch ob der eigentliche Programmablauf erhalten bleibt, die Struktur des Quelltextes weiterhin erkennbar ist und ob die Herkunft von bestimmten Variablen, wie verwendete Schlüssel oder Hashwerte ersichtlich werden.

Im folgenden werden einige Programme aus [Ngu18] zur Analyse verwendet. Darauf

Programmname	Ba	Ch	Fo	Tb
ArithmeticTest	385 ms	358 ms	381 ms	321 ms
DeepCallee	327 ms	350 ms	296 ms	322 ms
MultipleCallee	294 ms	340 ms	321 ms	308 ms
AESEncryptDecrypt	750 ms	620 ms	758 ms	410 ms
DigestVerify	357 ms	449 ms	566 ms	317 ms
FileEncryption	1026 ms	438 ms	517 ms	460 ms
SignatureVerify	402 ms	443 ms	418 ms	419 ms
SignEncryptDecrypt	983 ms	1050 ms	881 ms	924 ms
AuthorOriginatorContentInterchange	2540 ms	3488 ms	16148 ms	3165 ms
MultiFetchDelivery	3570 ms	3682 ms	13967 ms	3261 ms
OneWayMessage_ActiveRecipient	3571 ms	3648 ms	13060 ms	3755 ms
OneWayMessage_PassiveRecipient	3019 ms	2901 ms	11003 ms	3018 ms
PartialOneWayMessage_ActiveRecipient	**2079** ms	3722 ms	9934 ms	**2129** ms
RequestResponse	2974 ms	2733 ms	9914 ms	2650 ms
SendEncryptedContent	2759 ms	3016 ms	10964 ms	2847 ms

Tabelle 7: Ergebnisse der Berechnungszeit mit `-XX:+ScavengeBeforeFullGC`

Programmname	Ba	Ch	Fo	Tb
ArithmeticTest	464 ms	293 ms	298 ms	304 ms
DeepCallee	293 ms	306 ms	297 ms	271 ms
MultipleCallee	258 ms	281 ms	288 ms	283 ms
AESEncryptDecrypt	577 ms	500 ms	550 ms	517 ms
DigestVerify	617 ms	727 ms	587 ms	312 ms
FileEncryption	699 ms	385 ms	434 ms	329 ms
SignatureVerify	311 ms	341 ms	311 ms	322 ms
SignEncryptDecrypt	749 ms	1146 ms	784 ms	846 ms
AuthorOriginatorContentInterchange	3153 ms	3157 ms	12914 ms	2658 ms
MultiFetchDelivery	3041 ms	3165 ms	11860 ms	2874 ms
OneWayMessage_ActiveRecipient	3049 ms	3315 ms	9515 ms	3061 ms
OneWayMessage_PassiveRecipient	2847 ms	2784 ms	9966 ms	2488 ms
PartialOneWayMessage_ActiveRecipient	2851 ms	3280 ms	8203 ms	2691 ms
RequestResponse	2638 ms	2757 ms	7954 ms	2269 ms
SendEncryptedContent	2288 ms	2413 ms	8768 ms	2307 ms

Tabelle 8: Ergebnisse der Berechnungszeit mit `-XX:UseParallelGC`

Programmname	Ba	Ch	Fo	Tb
ArithmeticTest	315 ms	446 ms	392 ms	262 ms
DeepCallee	257 ms	271 ms	263 ms	245 ms
MultipleCallee	240 ms	263 ms	248 ms	246 ms
AESEncryptDecrypt	519 ms	512 ms	481 ms	357 ms
DigestVerify	339 ms	367 ms	642 ms	324 ms
FileEncryption	722 ms	371 ms	400 ms	339 ms
SignatureVerify	331 ms	336 ms	342 ms	337 ms
SignEncryptDecrypt	759 ms	820 ms	662 ms	744 ms
AuthorOriginatorContentInterchange	2799 ms	2931 ms	13506 ms	2690 ms
MultiFetchDelivery	2919 ms	2890 ms	7334 ms	2636 ms
OneWayMessage_ActiveRecipient	2840 ms	2866 ms	10663 ms	**1874** ms
OneWayMessage_PassiveRecipient	2548 ms	2362 ms	8331 ms	2356 ms
PartialOneWayMessage_ActiveRecipient	2643 ms	**2009** ms	7834 ms	2905 ms
RequestResponse	2715 ms	2671 ms	8886 ms	2419 ms
SendEncryptedContent	2422 ms	2613 ms	9743 ms	2484 ms

Tabelle 9: Ergebnisse der Berechnungszeit mit `-XX:+UseParallelOldGC`

Programmname	Ba	Ch	Fo	Tb
ArithmeticTest	337 ms	319 ms	295 ms	281 ms
DeepCallee	302 ms	336 ms	276 ms	281 ms
MultipleCallee	280 ms	325 ms	280 ms	264 ms
AESEncryptDecrypt	526 ms	514 ms	552 ms	441 ms
DigestVerify	397 ms	406 ms	673 ms	343 ms
FileEncryption	851 ms	458 ms	423 ms	433 ms
SignatureVerify	390 ms	365 ms	340 ms	390 ms
SignEncryptDecrypt	918 ms	946 ms	782 ms	832 ms
AuthorOriginatorContentInterchange	3128 ms	**2266** ms	12901 ms	2977 ms
MultiFetchDelivery	3403 ms	2719 ms	**6828** ms	**2048** ms
OneWayMessage_ActiveRecipient	3126 ms	3047 ms	9969 ms	2677 ms
OneWayMessage_PassiveRecipient	2587 ms	2576 ms	**5486** ms	2537 ms
PartialOneWayMessage_ActiveRecipient	2719 ms	2190 ms	9465 ms	3868 ms
RequestResponse	3299 ms	3361 ms	10725 ms	3012 ms
SendEncryptedContent	3004 ms	3209 ms	11766 ms	**2035** ms

Tabelle 10: Ergebnisse der Berechnungszeit mit `-XX:+UseSerialGC`

Programmname	Ba	Ch	Fo	Tb
ArithmeticTest	266 ms	240 ms	231 ms	233 ms
DeepCallee	230 ms	230 ms	218 ms	204 ms
MultipleCallee	**195** ms	218 ms	**202** ms	207 ms
AESEncryptDecrypt	446 ms	412 ms	443 ms	375 ms
DigestVerify	303 ms	310 ms	537 ms	270 ms
FileEncryption	641 ms	310 ms	328 ms	290 ms
SignatureVerify	292 ms	307 ms	279 ms	285 ms
SignEncryptDecrypt	699 ms	766 ms	638 ms	681 ms
AuthorOriginatorContentInterchange	2736 ms	2729 ms	10572 ms	2327 ms
MultiFetchDelivery	2714 ms	2769 ms	10059 ms	2595 ms
OneWayMessage_ActiveRecipient	3014 ms	2944 ms	10262 ms	2471 ms
OneWayMessage_PassiveRecipient	2387 ms	**2289** ms	7878 ms	2226 ms
PartialOneWayMessage_ActiveRecipient	2614 ms	2408 ms	8777 ms	2580 ms
RequestResponse	**2381** ms	**1534** ms	8373 ms	2154 ms
SendEncryptedContent	2216 ms	**2366** ms	8666 ms	2280 ms

Tabelle 11: Ergebnisse der Berechnungszeit mit `-XX:+UseG1GC`

folgend sind im Rahmen der Arbeit entstandene Beispielprogramme, welche die JCA/JCE verwenden, analysiert worden und schlussendlich wurden die Beispielprogramme aus der OSCI-Bibliothek als Grundlage für die Analyse herangezogen.

Alle Evaluierungen wurden mit den Standardeinstellungen, welche aus [Cyl19] stammen, durchgeführt. Für die Kontrollabhängigkeiten wurde die Option `NO_EXCEPTIONAL_EDGES`, für die Datenabhängigkeiten `NO_HEAP`, für die Reflections `NONE` und als CFA-Constraint `ZERO_ONE_CFA` gesetzt. Als Eintrittspunkt wurde immer die `main`-Methode gewählt, aus dem Grund das auch untersucht wird, inwiefern das Analysewerkzeug dabei hilft die Bibliotheken korrekt in Programme einzubinden.

Der Quelltext wird an bestimmten Stellen gekürzt, auf diese wird entweder einmalig für jeglichen Quelltext oder im Einzelfall hingewiesen. In allen Beispielen wurden die `import`- und `package`-Anweisungen ausgelassen.

5.3 ArithmeticTest

In diesem Kapitel wird anhand eines Programmbeispiels die Funktionsweise des Slicing verdeutlicht und bietet Möglichkeit die verschiedenen Optionen und Techniken sowie ihre Auswirkungen auf das Ergebnis einordnen zu können.

Beispiel 11 *In Codeauszug 14 ist das Programm aus [Ngu18] aufgelistet. Die umliegende*

Klasse und die Definition der ***main****-Methode wurden dabei ausgelassen. Der unmodifizierte Quelltext ist in A.1.1 zu sehen.*

In diesem Programm werden drei Variablen angelegt die über verschiedene Weise miteinander addiert und multipliziert sowie inkrementiert und dekrementiert werden. Am Ende des Programms werden die Variablen ***value*** *und* ***value2*** *der Methode* ***println*** *übergeben.*

```
1  int value = 1;
2  int value2 = 2;
3  value += value;
4  int someOtherValue = 2;
5  value2 += someOtherValue;
6  someOtherValue -= someOtherValue;
7  value = value--;
8
9  while(someOtherValue < 2){
10   someOtherValue++;
11 }
12
13 while(value < 2){
14   value*= 2;
15 }
16
17 while(value2 < 10){
18   value2*= 2;
19 }
20
21 System.out.println(value);
22 System.out.println(value2);
```

Codeauszug 14: `main`-Methode der Klasse `ArithmeticTest`

Im Codeauszug 15 ist der Quelltext nach Ausführung des Forward-Slicing zu sehen. Als Seed-Statement wurde dabei die Zeile 4 gewählt. Eindeutig zu erkennen ist, dass die Variable `someOtherValue` und alle daraus resultierenden Abhängigkeiten nicht enthalten sind. Denn vom Seed-Statement zu `someOtherValue` gibt es keine Kontroll- oder Datenabhängigkeit.

Dadurch, dass in dem Ergebnis die Variablen `value` und `value2` vorkommen, wurden auch die Definitionen dieser Variablen rekonstruiert. Diese Evaluierung wurde mit der Option `NONE` als Kontrollabhängigkeiten wiederholt. Denn dann sind die Anweisungen mit

der Variable `value2` nicht im Ergebnis vorhanden. Das bedeutet in dem Ergebnis mit Variable `value2` gibt es eine Kontrollabhängigkeit zwischen Anweisungen mit der Variable `value` und `value2`.

```
1  int value = 1;
2  int value2 = 2;
3  value += value;
4  value = value--; // Seed-Statement
5
6  while(value < 2){
7    value*= 2;
8  }
9
10 while(value2 < 10){
11   value2*= 2;
12 }
13
14 System.out.println(value);
15 System.out.println(value2);
```

Codeauszug 15: `main`-Methode der Klasse `ArithmeticTest` nach dem Forward-Slicing (Seed-Statement in Zeile 4)

Für das Beispielprogramm aus [Ngu18] wurde das Forward-Slicing ein weiteres mal evaluiert. Diesmal wurde dafür in Codeauszug 16 die Zeile 3 als Seed Statement gewählt.

Hierbei ist auch die Rekonstruktion der Variablendefinitionen `value2` und `someOther Value` im Ergebnis vorhanden und nur Anweisungen, welche Abhängigkeiten vom Seed-Statement besitzen, sind vorhanden. Hieraus ist durchaus einfach zu erkennen, wie der Wert der Variable `value2` zustande kommt. Wie im vorherigen Beispiel wird erwähnt, dass mit der Kontrollabhängigkeitsoption `NONE` die Anweisungen mit der Variable `someOther Value` nicht mehr im Ergebnis vorkommen.

```
1  int value2 = 2;
2  int someOtherValue = 2;
3  value2 += someOtherValue; // Seed-Statement
4  while(value2 < 10){
5    value2*= 2;
6  }
7
```

```
8 System.out.println(value2);
```

Codeauszug 16: `main`-Methode der Klasse `ArithmeticTest` nach dem Forward-Slicing (Seed-Statement in Zeile 3)

Im Codeauszug 17 wurde das Programm mithilfe des Backward-Slicing und der Zeile 13 als Seed-Statement analysiert. Hier sind ebenfalls die rekonstruierten Variablendefinitionen vorhanden. Dass Anweisungen mit der Variable `someOtherValue` vorkommen scheint an den Kontrollabhängigkeitsoptionen zu liegen. Ein Vergleich mit denselben Optionen und dem Seed-Statement aus Zeile 7 beim Verwenden des Thin-Backward-Slicing belegt diese Annahme. Im Codeauszug 18 ist das Ergebnis der Analyse mit Thin-Backward-Slicing zu sehen. Das Ergebnis ist dasselbe, wenn beim Chopping zusätzlich die Zeile 2 gewählt wird.

```
1  int value = 1;
2  value += value;
3  int someOtherValue = 2;
4  someOtherValue -= someOtherValue;
5  while(someOtherValue < 2){
6    someOtherValue++;
7  }
8
9  while(value < 2){
10   value*= 2;
11 }
12
13 System.out.println(value); // Seed-Statement
```

Codeauszug 17: `main`-Methode der Klasse `ArithmeticTest` nach dem Backward-Slicing (Seed-Statement in Zeile 13)

Im Codeauszug 18 wurde das Programm mithilfe des Thin-Backward-Slicing untersucht, dabei wurde Zeile 7 als Seed-Statment verwendet. Der Codeauszug ist identisch mit dem Ergebnis bei Verwendung des Choppings. Dabei wurde die Zeile 2 als weiteres Seed-Statement verwendet.

Wenn ausschließlich das Ergebnis betrachtet wird, könnte der Schluss gefasst werden, dass eindeutig zu erkennen ist, wie der Wert der Variable `value` entsteht. Es fehlt allerdings eine Zuweisung der Variable `value`, welche den Wert einmalig dekrementiert (vgl. Codeauszug 14, Zeile 7).

```
1 int value = 1;
```

```
2 value += value; // zusätzliches Seed-Statement beim Chopping
3 while(value < 2){
4   value*= 2;
5 }
6
7 System.out.println(value); // Seed-Statement
```

Codeauszug 18: `main`-Methode der Klasse `ArithmeticTest` nach dem Thin-Backward-Slicing (Seed-Statement in Zeile 7) und Chopping (Seed-Statement zusätzlich aus Zeile 2)

Im Codeauszug 19 das Programm mithilfe des Backward-Slicing, allerdings mit Zeile 19 als Seed-Statement analysiert. Im Ergebnis sind auch Anweisungen mit den Variablen `value` und `someOtherValue` vorhanden, dass bedeutet zu diesen gibt es Kontroll- oder Datenabhängigkeiten.

```
1  int value = 1;
2  int value2 = 2;
3  value += value;
4  int someOtherValue = 2;
5  value2 += someOtherValue;
6  someOtherValue -= someOtherValue;
7  while(someOtherValue < 2){
8    someOtherValue++;
9  }
10
11 while(value < 2){
12   value*= 2;
13 }
14
15 while(value2 < 10){
16   value2*= 2;
17 }
18
19 System.out.println(value2); //Seed-Statment
```

Codeauszug 19: `main`-Methode der Klasse `ArithmeticTest` nach dem Backward-Slicing (Seed-Statement in Zeile 19)

Im Codeauszug 20 ist das Ergebnis des Analysevorgangs mithilfe des Thin-Backward-

Slicing und Chopping vorzufinden. Aus dem Codeauszug wird eindeutig ersichtlich, wie der Wert der Variable `value2` im Seed-Statement errechnet wird und welche Variablen Einfluss auf den endgültigen Wert der Variable nehmen.

```
1 int value2 = 2;
2 int someOtherValue = 2;
3 value2 += someOtherValue; // zusätzliches Seed-Statement beim Chopping
4 while(value2 < 10){
5   value2*= 2;
6 }
7
8 System.out.println(value2); // Seed-Statment
```

Codeauszug 20: `main`-Methode der Klasse `ArithmeticTest` nach dem Thin-Backward-Slicing (Seed-Statement in Zeile 8) und Chopping (Seed-Statement zusätzlich aus Zeile 3)

5.4 Kryptografische Beispiele

Diese Kapitel untersucht Beispielprogramme, welche im Rahmen dieser Arbeit entstanden sind. Die Programme arbeiten im Kontext der Informationssicherheit und verschlüsseln, signieren oder wenden Hashfunktionen auf Informationen an. Dabei sollen bestimmte Forschungsfragen beantwortet werden. Darunter, ob es mithilfe des Slicing möglich ist die verwendeten kryptografischen Algorithmen zu identifizieren oder ob es möglich ist die Informationen, welche verschlüsselt oder signiert wurden zu erkennen sowie ob die für das Verschlüsseln oder Signieren verwendeten Geheimnisse erkennbar sind.

Beispiel 12 *In Codeauszug 21 ist die* ***`main`****-Methode der Klasse* ***`AESEncryptDecrypt`*** *abgebildet. Die Methode verwendet einen Schlüssel, um per AES eine Nachricht zu verschlüsseln und anschließend zu entschlüsseln.*

Die Klasse ***`AES`****, welche dafür verwendet wird, ist in Codeauszug 22 abgebildet. Diese Klasse beinhaltet ein innere Klasse* ***`KeyHolder`*** *und implementiert das Interface* ***`ICryptor`****.*

Das Interface ***`ICryptor`*** *(Codeauszug 23)erweitert die Interfaces* ***`IDecrypter`*** *(Codeauszug 24) und* ***`IEncrypter`*** *(Codeauszug 25).* ***`IDecrypter`*** *definiert die Schnittstelle mit der* ***`decrypt`****-Methode und* ***`IEncrypter`*** *definiert die Schnittstelle mit der* ***`encrypt`****-Methode. In der Klasse* ***`AES`*** *werden die beiden Schnittstellenmethoden implementiert.*

```
1 final String secretKey = "ThisIsATopSecretNeverToBeLostKey";
```

```
2
3  AES aesA = new AES(secretKey);
4
5  String originalMsg = "ThisIsATopSecretMessageNeverToBeDecrypted.";
6  String encrypted = aesA.encrypt(originalMsg);
7
8  System.out.println(encrypted);
9
10 String decrypted = aesA.decrypt(encrypted);
11
12 System.out.println(decrypted);
```

Codeauszug 21: `main`-Methode der Klasse `AESEncryptDecrypt`

```
1  public class AES implements ICryptor {
2    private KeyHolder keyHolder;
3
4    public AES(String secretKey) throws NoSuchAlgorithmException {
5      this.keyHolder = new KeyHolder(secretKey);
6    }
7
8    public class KeyHolder {
9      private final SecretKeySpec secretKey;
10
11     protected SecretKeySpec getSecretKey() {
12       return this.secretKey;
13     }
14
15     public KeyHolder(String keyString) throws NoSuchAlgorithmException {
16       MessageDigest sha = null;
17
18       byte[] keyBytes = keyString.getBytes(StandardCharsets.UTF_8);
19
20       // create sha for key
21       sha = MessageDigest.getInstance("SHA-1");
22       keyBytes = sha.digest(keyBytes);
23
24       // truncate to 16 bytes
25       keyBytes = Arrays.copyOf(keyBytes, 16);
```

```
26        secretKey = new SecretKeySpec(keyBytes, "AES");
27      }
28    }
29
30    @Override
31    public String encrypt(final String strToEncrypt) {
32      try {
33        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
34        cipher.init(Cipher.ENCRYPT_MODE, this.keyHolder.getSecretKey());
35        return Base64.getEncoder()
36        .encodeToString(cipher.doFinal(strToEncrypt.getBytes(StandardCharsets↩
              .UTF_8)));
37      } catch (Exception e) {
38        System.out.println("Error while encrypting: " + e);
39      }
40
41      return null;
42    }
43
44    @Override
45    public String decrypt(final String strToDecrypt) {
46      try {
47        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5PADDING");
48        cipher.init(Cipher.DECRYPT_MODE, this.keyHolder.getSecretKey());
49        return new String(cipher.doFinal(Base64.getDecoder()
50        .decode(strToDecrypt)));
51      } catch (Exception e) {
52        System.out.println("Error while decrypting: " + e);
53      }
54
55      return null;
56    }
57  }
```

Codeauszug 22: Klasse AES

```
1  public interface ICryptor extends IDecrypter, IEncrypter {
```

```
2 }
```

Codeauszug 23: Interface `ICryptor`

```
1 public interface IDecrypter {
2   public String decrypt(String toDecrypt);
3 }
```

Codeauszug 24: Interface `IDecrypter`

```
1 public interface IEncrypter {
2   public String encrypt(String toEncrypt);
3 }
```

Codeauszug 25: Interface `IEncrypter`

In den folgenden Codeauszügen 26 und 27 ist das Ergebnis nach dem Forward-Slicing zu sehen. Als Seed-Statement wurde die Zeile 3 verwendet.

Die Variablen `aesA` und `originalMsg` wurden rekonstruiert, da sie in den anderen Anweisungen verwendet wurden. Der eigentliche `secretKey` wurde dabei nicht rekonstruiert. In den Funktionen ist die verwendete Methode zum verschlüsseln zu erkennen. In diesem Fall verschlüsselt die Klasse `AES` mithilfe vom Advanced Encryption Standard (AES) im Electronic Code Book (ECB) Modus und es wird mithilfe der Padding-Methode aus Password-Key Cryptography Standards (PKCS) gearbeitet. Padding bedeutet das Auffüllen von fehlenden Bits damit die anschließende Anzahl von Bits durch einen bestimmten Teiler teilbar ist. Da der ECB Modus immer einen Block verschlüsselt muss möglicherweise der letzte Block aufgefüllt werden, falls nicht genug Information zum Verschlüsseln vorhanden ist.

In dieser Rekonstruktion fehlt der Konstruktor der Klasse `AES`, obwohl er in der Klasse `AESEncryptDecrypt` verwendet, um ein Objekt vom Typ `AES` zu erzeugen. Im Konstruktor der Klasse `AES` (Codeauszug 22) wird ein Objekt vom Typ `AES.KeyHolder` erzeugt und die Variable `secretKey` wird übergeben. Das bedeutet auch der Konstruktor der Klasse `KeyHolder` fehlt im rekonstruierten Quelltext. Im Konstruktor der Klasse `KeyHolder` kann die Verarbeitung des Schlüssels nachvollzogen werden. Diese Verarbeitungsschritte fehlen im Ergebnis.

```
1 AES aesA = new AES(secretKey);
2 String originalMsg = "ThisIsATopSecretMessageNeverToBeDecrypted.";
3 String encrypted = aesA.encrypt(originalMsg);
4 System.out.println(encrypted);
```

```
5 String decrypted = aesA.decrypt(encrypted);
6 System.out.println(decrypted);
```

Codeauszug 26: `main`-Methode der Klasse `AESEncryptDecrypt` nach dem Forward-Slicing

```
1  public class AES implements ICryptor {
2    private KeyHolder keyHolder;
3    public class KeyHolder {
4      private final SecretKeySpec secretKey;
5      protected SecretKeySpec getSecretKey() {
6        return this.secretKey;
7      }
8    }
9
10   @Override
11   public String encrypt(final String strToEncrypt) {
12     try {
13       Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
14       cipher.init(Cipher.ENCRYPT_MODE, this.keyHolder.getSecretKey());
15       return Base64.getEncoder()
16       .encodeToString(cipher.doFinal(strToEncrypt.getBytes(StandardCharsets↩
             .UTF_8)));
17     } catch (Exception e) {
18       System.out.println("Error while encrypting: " + e);
19     }
20   }
21
22   @Override
23   public String decrypt(final String strToDecrypt) {
24     try {
25       Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5PADDING");
26       cipher.init(Cipher.DECRYPT_MODE, this.keyHolder.getSecretKey());
27       return new String(cipher.doFinal(Base64.getDecoder()
28       .decode(strToDecrypt)));
29     } catch (Exception e) {
30       System.out.println("Error while decrypting: " + e);
31     }
32   }
```

```
33 }
```

Codeauszug 27: Klasse `AES` nach dem Forward-Slicing

Die Codeauszüge 28 und 29 zeigen das Ergebnis, nachdem der Quelltext mithilfe von Backward- oder Thin-Backward-Slicing analysiert wurde. Dabei wurde die Zeile 4 als Seed-Statment verwendet. Mithilfe des Chopping zwischen den Zeilen 3 und 4 wird exakt das selbe Ergebnis erreicht.

Dabei fällt auf, dass die `decrypt`-Methode fehlt. Die ist dadurch zu erklären, dass keine Daten- oder Kontrollabhängigkeiten in rückwärtiger Richtung zwischen dem Seed-Statement aus Zeile 4 und den Anweisungen der `decrypt`-Methode bestehen. Durch die Schnittbildung zwischen Forward- und Backward-Slicing beim Chopping entfällt diese Methode ebenfalls.

```
1 AES aesA = new AES(secretKey);
2 String originalMsg = "This good stuff, send more.";
3 String encrypted = aesA.encrypt(originalMsg); // zusätzliches Seed-↩
    Statement beim Chopping
4 String decrypted = aesA.decrypt(encrypted); // Seed-Statement
```

Codeauszug 28: `main`-Methode der Klasse `AESEncryptDecrypt` nach dem Backward-Slicing, Thin-Backward-Slicing und Chopping

```
1  public class AES implements ICryptor {
2    @Override
3    public String encrypt(final String strToEncrypt) {
4      try {
5        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
6        return Base64.getEncoder()
7        .encodeToString(cipher.doFinal(strToEncrypt.getBytes(StandardCharsets↩
             .UTF_8)));
8      } catch (Exception e) {
9      }
10
11     return null;
12   }
13 }
```

Codeauszug 29: Klasse `AES` nach dem Backward-Slicing, Thin-Backward-Slicing und Chopping

Beispiel 13 *Der Codeauszug 30 zeigt die* `main`*-Methode der Klasse* `DigestVerify` *in dieser Methode wird von mehreren Nachrichten der Hashwert berechnet, um anschließend zu überprüfen, ob eine der Nachrichten korrupt ist.*

Für diese Berechnung wird die Klasse `Digest` *aus Codeauszug 31 verwendet. Dem Konstruktor der Klasse kann der zu verwendende Hashalgorithmus übergeben werden und die Nachricht, zu welcher der Hashwert berechnet werden soll. Anschließend speichert das Objekt den Hashwert und mittels der* `toEquals`*-Methode kann der Hashwert mit einem Anderen verglichen werden.*

```
1  final String msg = "Is this message corrupt?";
2  final String msg2 = "This msg is corrupted!";
3  final String corrupt = "$$$" + msg2;
4
5  Digest msgDigest = new Digest(msg, null);
6  Digest msgDigest2 = new Digest(msg, null);
7  Digest msg2Digest = new Digest(msg2, null);
8  Digest corruptDigest = new Digest(corrupt, null);
9
10 System.out.println("MSG: " + msg);
11 System.out.println(msgDigest);
12 System.out.println(msgDigest2);
13 System.out.println(msgDigest.equals(msgDigest2));
14
15 System.out.println("MSG: "+ msg2);
16 System.out.println(msg2Digest);
17 System.out.println(corruptDigest);
18 System.out.println(corruptDigest.equals(msg2Digest));
```

Codeauszug 30: `main`-Methode der Klasse `DigestVerify`

```
1  public class Digest {
2    private final DigestAlgorithms digestAlgorithmsUsed;
3    private final String digest;
4    private final static DigestAlgorithms DEFAULT_DIGEST_ALGORITHMS = ↩
       DigestAlgorithms.SHA3_512;
5
6    public enum DigestAlgorithms {
7      MD2,
8      MD5,
```

```
9     SHA1,
10    SHA224,
11    SHA256,
12    SHA384,
13    SHA512_224,
14    SHA512_256,
15    SHA3_224,
16    SHA3_256,
17    SHA3_384,
18    SHA3_512
19  }
20
21  private final static Map<DigestAlgorithms, String> algorithmMap = ↩
        createAlgorithmMap();
22
23  private static Map<DigestAlgorithms, String> createAlgorithmMap() {
24    Map<DigestAlgorithms, String> algorithmStringMap = new HashMap<>();
25
26    algorithmStringMap.put(DigestAlgorithms.MD2, "MD2");
27    algorithmStringMap.put(DigestAlgorithms.MD5, "MD5");
28    algorithmStringMap.put(DigestAlgorithms.SHA1, "SHA-1");
29    algorithmStringMap.put(DigestAlgorithms.SHA224, "SHA-224");
30    algorithmStringMap.put(DigestAlgorithms.SHA256, "SHA-256");
31    algorithmStringMap.put(DigestAlgorithms.SHA384, "SHA-384");
32    algorithmStringMap.put(DigestAlgorithms.SHA512_224, "SHA-512/224");
33    algorithmStringMap.put(DigestAlgorithms.SHA512_256, "SHA-512/256");
34    algorithmStringMap.put(DigestAlgorithms.SHA3_224, "SHA3-224");
35    algorithmStringMap.put(DigestAlgorithms.SHA3_256, "SHA3-256");
36    algorithmStringMap.put(DigestAlgorithms.SHA3_384, "SHA3-384");
37    algorithmStringMap.put(DigestAlgorithms.SHA3_512, "SHA3-512");
38
39    return algorithmStringMap;
40  }
41
42  public Digest(String of, DigestAlgorithms digestAlgorithms) throws ↩
        NoSuchAlgorithmException {
43    if(digestAlgorithms == null) digestAlgorithms = ↩
          DEFAULT_DIGEST_ALGORITHMS;
```

```
44      this.digestAlgorithmsUsed = digestAlgorithms;
45
46      MessageDigest messageDigest = MessageDigest.getInstance(algorithmMap.↩
            get(digestAlgorithmsUsed));
47
48      this.digest = Base64.getEncoder()
49      .encodeToString(messageDigest.digest(of.getBytes(StandardCharsets.↩
            US_ASCII)));
50    }
51
52    @Override
53    public boolean equals(Object obj) {
54      if(!(obj instanceof Digest)) return false;
55
56      Digest other = (Digest) obj;
57      return Objects.equals(other.digest, this.digest)
58      && Objects.equals(other.digestAlgorithmsUsed, this.↩
            digestAlgorithmsUsed);
59    }
60
61    @Override
62    public String toString() {
63      return this.getBase64();
64    }
65
66    public String getBase64() {
67      return this.digest;
68    }
```

Codeauszug 31: Klasse `Digest`

In den Codeauszügen 32 und 33 ist das Ergebnis des Forward-Slicing mit der Zeile 1 als gewähltes Seed-Statement abgebildet. In diesem Fall ist im Konstruktor die verwendete Hashfunktion zu erkennen. In diesem Fall wird `SHA3` mit 512 Bytes Hashlänge verwendet. Im Konstruktor ist auch zu erkennen, das der Hashwert als Base64 kodierter String gespeichert wird und die `toEquals`-Methode ist im Ergebnis vorhanden. Das bedeutet es ist nachvollziehbar, wie die Hashwerte verglichen werden. In diesem Fall wird zum Hashwert zusätzlich der Algorithmus verglichen.

Das in der Klasse `DigestVerify` auch Zeilen mit der Variable `corruptDigest` vorkommen, liegt an einer vorhandenen Kontrollabhängigkeit.

```
1 Digest msg2Digest = new Digest(msg2, null); // Seed-Statment
2 Digest corruptDigest = new Digest(corrupt, null);
3 System.out.println(msg2Digest);
4 System.out.println(corruptDigest.equals(msg2Digest));
```

Codeauszug 32: `main`-Methode der Klasse `DigestVerify` nach dem Forward-Slicing

```
1  public class Digest {
2    private final DigestAlgorithms digestAlgorithmsUsed;
3    private final String digest;
4    private final static DigestAlgorithms DEFAULT_DIGEST_ALGORITHMS = ↩
         DigestAlgorithms.SHA3_512;
5    private final static Map<DigestAlgorithms, String> algorithmMap = ↩
         createAlgorithmMap();
6    public Digest(String of, DigestAlgorithms digestAlgorithms) throws ↩
         NoSuchAlgorithmException {
7      if(digestAlgorithms == null) digestAlgorithms = ↩
           DEFAULT_DIGEST_ALGORITHMS;
8      this.digestAlgorithmsUsed = digestAlgorithms;
9      MessageDigest messageDigest = MessageDigest.getInstance(algorithmMap.↩
           get(digestAlgorithmsUsed));
10     this.digest = Base64.getEncoder()
11     .encodeToString(messageDigest.digest(of.getBytes(StandardCharsets.↩
           US_ASCII)));
12   }
13
14   @Override
15   public boolean equals(Object obj) {
16     if(!(obj instanceof Digest)) return false;
17     Digest other = (Digest) obj;
18     return Objects.equals(other.digest, this.digest)
19     && Objects.equals(other.digestAlgorithmsUsed, this.↩
           digestAlgorithmsUsed);
20   }
21 }
```

Codeauszug 33: Klasse `Digest`

Im Codeauszug 34 ist das Ergebnis bei Verwendung vom Backward- oder Thin-Backward-Slicing zu erkennen. Dabei wurde die Zeile 2 als Seed-Statement gewählt. Auch das Chopping enthält als Ergebnis nur die Zeilen, welche auch als Seed-Statement angegeben wurden.

```
1 Digest msg2Digest = new Digest(msg2, null); // zusätzliches Seed-Statement↩
      beim Chopping
2 System.out.println(msg2Digest); // Seed-Statement
```

Codeauszug 34: `main`-Methode der Klasse `DigestVerify` nach dem Backward- oder Thin-Backward-Slicing sowie Chopping

Beispiel 14 *Im Codeauszug 35 ist die* ***main****-Methode der Klasse* ***SignatureVerify*** *aufgelistet. In dieser Methode wird ein Objekt vom Typ* ***StringSigner*** *erzeugt. Mithilfe dieses Objektes wird ein String signiert. Anschließend wird die Signatur auf Korrektheit überprüft.*

Der Quelltext der Klasse ***StringSigner*** *ist im Codeauszug 36 zu sehen. Die Klasse besitzt, ähnlich den Verschlüsselungsbeispielen, zwei Methoden die* ***sign****-Methode und die* ***verify****-Methode.*

```
1 StringSigner stringSigner = new StringSigner();
2
3 final String toSign = "a signed message from a friend";
4
5 String base64Signature = stringSigner.sign(toSign);
6
7 System.out.println("Is Signature valid for msg: " + (stringSigner.verify(↩
    base64Signature, toSign) ? "Yes" : "No"));
```

Codeauszug 35: `main`-Methode der Klasse `SignatureVerify`

```
1 public class StringSigner {
2   private final KeyPair keyPair;
3   private final String cryptoAlgorithm = "Ed25519";
4
5   public StringSigner() throws NoSuchAlgorithmException {
6     KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance(↩
        cryptoAlgorithm);
7
8     this.keyPair = keyPairGenerator.generateKeyPair();
```

```
9   }
10
11  public String sign(String toSign) throws InvalidKeyException, ↩
        NoSuchAlgorithmException, SignatureException {
12    Signature signature = Signature.getInstance(cryptoAlgorithm);
13    signature.initSign(keyPair.getPrivate());
14
15    signature.update(toSign.getBytes());
16    String signString = Base64.getEncoder().encodeToString(signature.sign↩
          ());
17
18    return signString;
19  }
20
21  public boolean verify(String signString, String signedContent) throws ↩
        InvalidKeyException, NoSuchAlgorithmException, SignatureException {
22    Signature signature = Signature.getInstance(cryptoAlgorithm);
23    signature.initVerify(keyPair.getPublic());
24
25    signature.update(signedContent.getBytes());
26
27    return signature.verify(Base64.getDecoder().decode(signString));
28  }
29 }
```

Codeauszug 36: Klasse `StringSigner`

In den Codeauszügen 37 und 38 ist das Ergebnis nach dem Forward-Slicing dargestellt. Als Seed-Statement wurde die Zeile 3 verwendet.

In diesem Ergebnis sind die Methoden `sign` und `verify` vorhanden. Es ist also zu erkennen, dass die Klasse die Signatur als Base64 kodierter String ausgibt. Zusätzlich wurde die Variable `cryptoAlgorithm` rekonstruiert, da sie in den anderen Anweisungen vorkommt. Dadurch ist bei der Analyse bekannt, welcher Algorithmus zum signieren verwendet wurde.

Bei diesem Beispiel fehlt der Konstruktor. Der Konstruktor würde Aufschluss über das verwendete Schlüsselpaar geben. In diesem Fall ist nur bekannt, dass das Schlüsselpaar intern in der Klasse `StringSigner` über die Variable `keyPair` verwaltet wird. Es ist aber nicht bekannt, auf welchem Wege das Schlüsselpaar erzeugt wurde. Der Konstruktor fehlt,

weil die Erzeugung des Objektes `stringSigner` rekonstruiert wurde. Nach der Rekonstruktion wird nicht erneut geprüft, ob bestimmte Anweisungen oder Methodenkörper im Ergebnis vorhanden sein müssten, wie in diesem Fall der Konstruktor.

```
1 StringSigner stringSigner = new StringSigner();
2 final String toSign = "a signed message from a friend";
3 String base64Signature = stringSigner.sign(toSign); // Seed-Statement
4 System.out.println("Is Signature valid for msg: " + (stringSigner.verify(↩
    base64Signature, toSign) ? "Yes" : "No"));
```

Codeauszug 37: `main`-Methode der Klasse `SignatureVerify` nach dem Forward-Slicing

```
1  public class StringSigner {
2    private final KeyPair keyPair;
3    private final String cryptoAlgorithm = "Ed25519";
4    public String sign(String toSign) throws InvalidKeyException, ↩
       NoSuchAlgorithmException, SignatureException {
5      Signature signature = Signature.getInstance(cryptoAlgorithm);
6      signature.initSign(keyPair.getPrivate());
7      signature.update(toSign.getBytes());
8      String signString = Base64.getEncoder().encodeToString(signature.sign↩
         ());
9      return signString;
10   }
11
12   public boolean verify(String signString, String signedContent) throws ↩
       InvalidKeyException, NoSuchAlgorithmException, SignatureException {
13     Signature signature = Signature.getInstance(cryptoAlgorithm);
14     signature.initVerify(keyPair.getPublic());
15     signature.update(signedContent.getBytes());
16     return signature.verify(Base64.getDecoder().decode(signString));
17   }
18 }
```

Codeauszug 38: Klasse `StringSigner` nach dem Forward-Slicing

In den Codeauszügen 39 und 40 sind die Ergebnisse nach dem Backward- und Thin-Backward-Slicing sowie Chopping aufgelistet. Für das Backward- und Thin-Backward-Slicing wurde die Zeile 4 als Seed-Statement verwendet. Für das Chopping wurde zusätzlich die Zeile 3 aus dem Forward-Slicing als Seed-Statement gewählt.

Der Codeauszug 39 ist identisch mit dem Codeauszug 37 des Forward-Slicing. Der Codeauszug 40 unterscheidet sich jedoch vom Forward-Slicing.

```
1 StringSigner stringSigner = new StringSigner();
2 final String toSign = "yadada this was a message from me...";
3 String base64Signature = stringSigner.sign(toSign); // zusätzliches Seed-Statement fürs Chopping
4 System.out.println("Is Signature valid for msg: " + (stringSigner.verify(base64Signature, toSign) ? "Yes" : "No")); // Seed-Statement
```

Codeauszug 39: `main`-Methode der Klasse `SignatureVerify` nach dem Backward- oder Thin-Backward-Slicing sowie Chopping

Im Codeauszug 40 sind die Methoden `sign` und `verify` im Ergebnis vorhanden. In dem Codeauszug ist zu erkennen, dass für die Methoden bestimmte Anweisungen fehlen. Es fehlt beispielsweise die Anweisung, welche den zu signierenden String dem `Signature`-Objekt übergibt. Auch die Übergabe des Schlüsselpaares ist nicht im Ergebnis vorhanden.

Bei der Analyse ist erkennbar, dass die Signatur in einen Base64 kodierten String geschrieben und dieser zurückgegeben wird. Auch in diesem Ergebnis fehlt der Konstruktor der Klasse `StringSigner`, welcher über die Herkunft des Schlüsselpaares Auskunft geben würde.

```
1  public class StringSigner {
2    private final String cryptoAlgorithm = "Ed25519";
3    public String sign(String toSign) throws InvalidKeyException, NoSuchAlgorithmException, SignatureException {
4      Signature signature = Signature.getInstance(cryptoAlgorithm);
5      String signString = Base64.getEncoder().encodeToString(signature.sign());
6      return signString;
7    }
8
9    public boolean verify(String signString, String signedContent) throws InvalidKeyException, NoSuchAlgorithmException, SignatureException {
10     Signature signature = Signature.getInstance(cryptoAlgorithm);
11     return signature.verify(Base64.getDecoder().decode(signString));
12   }
```

```
13 }
```

Codeauszug 40: Klasse `StringSigner` nach dem Backward- oder Thin-Backward-Slicing sowie Chopping

Beispiel 15 *Im Codeauszug 41 ist die* ***main****-Methode der Klasse* ***SignEncryptDigest*** *zu sehen. Diese Methode signiert eine Nachricht und berechnet anschließend den Hashwert der Nachricht mit der Signatur. Anschließend werden die Nachricht sowie Signatur verschlüsselt.*

Die einzelnen Komponenten der Nachricht werden nach dem Entschlüsseln extrahiert. Anschließend wird der Hashwert und die Signatur der Nachricht überprüft.

Für diesen Ablauf werden die Klassen ***AES*** *(Codeauszug 22) ,* ***Digest*** *(Codeauszug 31) und* ***StringSigner*** *(Codeauszug 36) der vorherigen Beispiele verwendet.*

Auf einige Anweisungen wurde aufgrund der Lesbarkeit verzichtet.

```
1  final String secretKey = "AgainATopSecretKey";
2  final String originalMsg = "MOTD: I like linux.";
3
4  StringSigner stringSigner = new StringSigner();
5  String base64Signature = stringSigner.sign(originalMsg);
6
7  final String toDigest = originalMsg + "--BEGIN-SIGNATURE--" + ↩
       base64Signature;
8  Digest msgDigest = new Digest(toDigest, null);
9
10 String toEncrypt = toDigest + "--BEGIN-DIGEST--" + msgDigest;
11
12 AES aes = new AES(secretKey);
13 String encrypted = aes.encrypt(toEncrypt);
14
15 AES decrypter = new AES(secretKey);
16 String decrypted = decrypter.decrypt(encrypted);
17
18 String digest = decrypted.split("--BEGIN-DIGEST--")[1];
19 String signature = decrypted.split("--BEGIN-DIGEST--")[0].split("--BEGIN-↩
       SIGNATURE--")[1];
20 String decryptedMsg = decrypted.split("--BEGIN-SIGNATURE")[0];
21
```

```
22 Digest newDigest = new Digest(decrypted.split("--BEGIN-DIGEST--")[0], null↩
       );
23
24 if(newDigest.equals(msgDigest)) {
25   System.out.println("Digest is correct.");
26 }else{
27   throw new RuntimeException();
28 }
29
30 if(stringSigner.verify(signature, decryptedMsg)) {
31   System.out.println("Signature is correct.");
32 }else{
33   throw new RuntimeException();
34 }
```

Codeauszug 41: `main`-Methode der Klasse `SignEncryptDigest`

In den Codeauszügen 42 bis 45 ist das Ergebnis nach dem Forward-Slicing abgebildet. Dabei wurde als Seed-Statement die Zeile 2 gewählt.

Es ist hervorzuheben, dass in diesem Ergebnis die Klassen `AES` und `Digest` vorhanden sind, denn in der `main`-Methode sind keine Aufrufe dieser Klassen zu erkennen und die Erzeugung der jeweiligen wurde nicht rekonstruiert. Die Klasse `StringSigner` wurde, wie schon im Codeauszug 38, so rekonstruiert, dass alle wichtigen Merkmale zu erkennen sind und damit für die Analyse bereitstehen.

```
1 final String originalMsg = "MOTD: I like linux.";
2 StringSigner stringSigner = new StringSigner(); // Seed-Statement
3 String base64Signature = stringSigner.sign(originalMsg);
4 final String toDigest = originalMsg + "--BEGIN-SIGNATURE--" + ↩
      base64Signature;
```

Codeauszug 42: `main`-Methode der Klasse `SignEncryptDigest` nach dem Forward-Slicing

```
1 public class AES implements ICryptor {
2   public class KeyHolder {
3     public KeyHolder(String keyString) throws NoSuchAlgorithmException {
4       MessageDigest sha = null;
5       byte[] keyBytes = keyString.getBytes(StandardCharsets.UTF_8);
6       keyBytes = sha.digest(keyBytes);
7     }
```

```
8   }
9 }
```

Codeauszug 43: Klasse `AES` nach dem Forward-Slicing

```
1  public class Digest {
2    private final DigestAlgorithms digestAlgorithmsUsed;
3    private final String digest;
4    private final static Map<DigestAlgorithms, String> algorithmMap = ↩
         createAlgorithmMap();
5    public Digest(String of, DigestAlgorithms digestAlgorithms) throws ↩
         NoSuchAlgorithmException {
6      MessageDigest messageDigest = MessageDigest.getInstance(algorithmMap.↩
           get(digestAlgorithmsUsed));
7      this.digest = Base64.getEncoder()
8      .encodeToString(messageDigest.digest(of.getBytes(StandardCharsets.↩
           US_ASCII)));
9    }
10 }
```

Codeauszug 44: Klasse `Digest` nach dem Forward-Slicing

```
1  public class StringSigner {
2    private final KeyPair keyPair;
3    private final String cryptoAlgorithm = "Ed25519";
4    public String sign(String toSign) throws InvalidKeyException, ↩
         NoSuchAlgorithmException, SignatureException {
5      Signature signature = Signature.getInstance(cryptoAlgorithm);
6      signature.initSign(keyPair.getPrivate());
7      signature.update(toSign.getBytes());
8      String signString = Base64.getEncoder().encodeToString(signature.sign↩
           ());
9      return signString;
10   }
11
12   public boolean verify(String signString, String signedContent) throws ↩
         InvalidKeyException, NoSuchAlgorithmException, SignatureException {
13     Signature signature = Signature.getInstance(cryptoAlgorithm);
14     signature.update(signedContent.getBytes());
```

```
15    }
16  }
```

Codeauszug 45: Klasse `StringSigner` nach dem Forward-Slicing

Die Codeauszüge 46, 44 und 48 zeigen die Ergebnisse beim Backward-Slicing. Dabei wurde die Anweisung in Zeile 14 als Seed-Statement verwendet. In der `main`-Methode sind mehr Anweisungen vorhanden als beim Forward-Slicing. Dabei wird der Ablauf des Programms deutlicher und in diesem Ergebnis ist zu erkennen, dass die Nachricht nicht nur signiert sondern auch der Hashwert berechnet wird und die Nachricht, Signatur und Hashwert anschließend zusammen verschlüsselt werden. Es ist nicht ersichtlich, welcher Schlüssel für die Verschlüsselung verwendet wird, denn die Variable `secretKey` wurde nicht rekonstruiert.

Beim Thin-Backward-Slicing ist das Ergebnis ähnlich, allein die Zeile 11 fehlt. Die Zeile 12 ist nicht im eigentlichen Ergebnis vorhanden, sondern wird rekonstruiert, da die Variable `signature` in der Zeile des Seed-Statements verwendet wird.

```
1  final String originalMsg = "MOTD: I like linux.";
2  StringSigner stringSigner = new StringSigner();
3  String base64Signature = stringSigner.sign(originalMsg);
4  final String toDigest = originalMsg + "--BEGIN-SIGNATURE--" + ↩
       base64Signature;
5  Digest msgDigest = new Digest(toDigest, null);
6  String toEncrypt = toDigest + "--BEGIN-DIGEST--" + msgDigest;
7  AES aes = new AES(secretKey);
8  String encrypted = aes.encrypt(toEncrypt);
9  AES decrypter = new AES(secretKey);
10 String decrypted = decrypter.decrypt(encrypted);
11 String digest = decrypted.split("--BEGIN-DIGEST--")[1];
12 String signature = decrypted.split("--BEGIN-DIGEST--")[0].split("--BEGIN-↩
       SIGNATURE--")[1];
13 String decryptedMsg = decrypted.split("--BEGIN-SIGNATURE")[0];
14 if(stringSigner.verify(signature, decryptedMsg)) { // Seed-Statement
15 }else{
16   throw new RuntimeException();
17 }
```

Codeauszug 46: `main`-Methode der Klasse `SignEncryptDigest` nach dem Backward-Slicing

Für die Klasse `AES` ist nur die `decrypt`-Methode im Ergebnis vorhanden. Allerdings sind hier nur die Rückgabewerte zu erkennen, die zumindest Aufschluss darüber geben, dass etwas verschlüsselt und anschließend in Base64 kodiert wird. Darüber ob der zu verschlüsselnde String noch verändert wird, kann keine Aussage getroffen werden. Auch von welchem Typ das `cipher`-Objekt ist und wie es erzeugt wird ist nicht ersichtlich.

```
1  public class AES implements ICryptor {
2    @Override
3    public String decrypt(final String strToDecrypt) {
4      try {
5        return new String(cipher.doFinal(Base64.getDecoder()
6        .decode(strToDecrypt)));
7      } catch (Exception e) {
8      }
9
10     return null;
11   }
12 }
```

Codeauszug 47: Klasse `AES` nach dem Backward-Slicing

Die Klasse `Digest` ist im Fall des Backward-Slicing mit angegebenem Seed-Statement nicht im Ergebnis vorhanden. Damit fehlen Aussagen über die verwendete Hashfunktion und wie der Vergleich der Hashwerte stattfindet.

Der Codeauszug 48 enthält die rekonstruierte Klasse `StringSigner`. Die Methode `verify` ist dabei im Ergebnis vorhanden. Die Methode `sign` nicht. In der Methode `verify` fehlen Anweisungen, die darüber Aufschluss geben würden, ob die übergebende `signedContent` Variable überhaupt im Zusammenhang mit dem Objekt `signature` steht. Trotzdem ist der verwendete Signaturalgorithmus zu entnehmen und das die entstandene Signatur Base64 kodiert wird, bevor sie vom `signature` Objekt überprüft wird.

```
1  public class StringSigner {
2    private final String cryptoAlgorithm = "Ed25519";
3    public boolean verify(String signString, String signedContent) throws ↩
         InvalidKeyException, NoSuchAlgorithmException, SignatureException {
4      Signature signature = Signature.getInstance(cryptoAlgorithm);
5      return signature.verify(Base64.getDecoder().decode(signString));
6    }
```

```
7 }
```

Codeauszug 48: Klasse `StringSigner` nach dem Backward-Slicing

5.5 OSCI Beispielprogramme

Das vorliegende Kapitel wird die Beispielprogramme der OSCI-Bibliothek (Version 2.3.0) untersuchen und dabei die Beantwortung der Forschungsfragen in den Vordergrund stellen. Die folgenden Beispiele haben eine deutlich höhere Komplexität als die der vorherigen Kapitel, deshalb werden, wenn möglich, Ergebnisse gekürzt dargestellt.

Im Codeauszug 49 ist die Klasse `AuthorOriginatorContentInterchange` zu sehen. Dieses Beispiel demonstriert das Erstellen einer gespeicherten Nachricht mit Anhang, welche anschließend auf dem Dateisystem gespeichert wird.

Die Methodeninhalte der Methoden `createStoredMessageFile` und `importToStore Delivery` sowie der leere Konstruktor wurden ausgelassen. Außerdem ist der Quelltext syntaktisch verändert, beispielsweise wurden Klammern verschoben, um die Anzeige kompakter zu gestalten. Durch die Anzahl der verwendeten Klassen wird auf dessen Darstellung verzichtet. Dafür werden in den Ergebnissen interessante oder besondere Ausschnitte, welche für die Analyse möglicherweise relevant sind, aufgenommen und kommentiert.

```
1  public class AuthorOriginatorContentInterchange {
2    public File createStoredMessageFile(String data, String ↩
         attachmentFilename)
3    throws Exception {
4      [...]
5    }
6
7    public StoreDelivery importToStoreDelivery(String intermedURL, File ↩
         storedFile)
8    throws Exception {
9      [...]
10   }
11
12   public static void main(String[] args) {
13     try {
14       AuthorOriginatorContentInterchange authOrigSample = new ↩
             AuthorOriginatorContentInterchange();
```

```
15      File stored = authOrigSample.createStoredMessageFile(args[1], args[2]); // Seed-Statement
16      StoreDelivery storeDel = authOrigSample.importToStoreDelivery(args[0], stored);
17      System.out.println("\nStoreDelivery-message with imported data:\n" + storeDel.toString());
18    } catch (Exception ex) {
19      ex.printStackTrace();
20    }
21  }
22 }
```

Codeauszug 49: Klasse `AuthorOriginatorContentInterchange`

Im weiteren Verlauf wird die Methode `createStoredMessageFile` analysiert. Hierbei werden die Forschungsfragen in den Vordergrund gestellt und mögliche Probleme bei der Analyse, welche durch fehlende Anweisungen im Ergebnis des Analysewerkzeuges entstehen, werden hervorgehoben.

Das Ergebnis des Forward-Slicing mit Zeile 15 als Seed-Statement ist identisch mit der originalen Klasse `AuthorOriginatorContentInterchange`, dabei sind nur Leerzeilen nicht enthalten. Das bedeutet der gesamte Quelltext der Klasse ist im Ergebnis vorhanden und wurde vollständig rekonstruiert. In den folgenden Codeauszügen wird untersucht, ob es möglich ist, herauszufinden, wie die Struktur der Daten aufgebaut wird und welche kryptografischen Verfahren gewählt wurden.

Im Codeauszug 50 ist ein Ausschnitt der Methode `createStoredMessageFile` nach dem Forward-Slicing gegeben. Zuerst wird ein Objekt vom Typ `ForwardDelivery` angelegt. Das Objekt bildet einen Weiterleitungsauftrag für den sogenannten Intermediär ab, dieser Intermediär ist eine Zwischenstelle beim OSCI Protokoll. Anschließend wird ein Objekt vom Typ `ContentContainer` angelegt. In diesen Container werden die zu übertragenden Daten und die anzuhängende Datei hinzugefügt. Anschließend wird der Container mithilfe des Zertifikats von *Alice* signiert. Der Container wird daraufhin mithilfe eines Objekts vom Typ `EncryptedDataOSCI` verschlüsselt. Die Verschlüsselung verwendet das Zertifikat von *Dave*. Die verschlüsselten Daten werden dem `ForwardDelivery` hinzugefügt und schlussendlich wird die Nachricht mithilfe eines statischen Aufrufs der `storeMessage`-Methode auf dem Dateisystem abgelegt.

```
1 [...]
```

```
2  ForwardDelivery forwardDel = new ForwardDelivery(clientDialog, null, "XX",↩
       "YY");
3  [...]
4  ContentContainer encrypted_container = new ContentContainer();
5  encrypted_container.addContent(new Content(data));
6  [...]
7  encrypted_container.addContent(new Content(new Attachment(new ↩
       FileInputStream(attachmentFilename), attachmentFilename.substring(path↩
       .length()), Constants.DEFAULT_SYMMETRIC_CIPHER_ALGORITHM)));
8
9  Author author = new Author(new de.osci.osci12.samples.impl.crypto.↩
       PKCS12Signer("/de/osci/osci12/samples/zertifikate/test_alice_signature↩
       .p12",
10 "123456"), null);
11 encrypted_container.sign(author);
12
13 EncryptedDataOSCI encryptedData = new EncryptedDataOSCI(Constants.↩
       SYMMETRIC_CIPHER_ALGORITHM_AES256_GCM,
14 encrypted_container);
15
16 Reader reader = new Reader(de.osci.helper.Tools.createCertificate(getClass↩
       ().getResourceAsStream("/de/osci/osci12/samples/zertifikate/↩
       test_dave_cypher.cer")));
17 encryptedData.encrypt(reader);
18 forwardDel.addEncryptedData(encryptedData);
19
20 File store = new File(path + "/StoredForwardDelivery.osci");
21 StoredMessage.storeMessage(forwardDel, new FileOutputStream(store));
22 return store;
```

Codeauszug 50: Ausschnitt der `createStoredMessageFile`-Methode der Klasse `Author OriginatorContentInterchange` nach dem Forward-Slicing

Nachfolgend werden die einzelnen Zeilen des Codeauszug 50 untersucht und dabei relevante Ausschnitte anderer Klassen gezeigt.

In Zeile 5 wird der Konstruktor der Klasse `Content` aufgerufen und die Variable `data` als Parameter übergeben. Das erzeugte Objekt dient als Parameter für den Aufruf der Methode `addContent` der Variable `encrypted_container`.

Im Codeauszug 51 ist der rekonstruierte Konstruktor der Klasse `Content` und relevan-

te Variablen, wie `NEXT_ID` und `DATA`, abgebildet. Der Ablauf zeigt, dass für den Inhalt ein Referenz Identifikator gesetzt wird. Dabei wird ein atomarer Integer inkrementiert. Die Daten werden anschließend geladen und der Inhaltstyp wird auf `DATA` gesetzt. Anschließend wird der Variable `transformers` die Variable `b64` hinzugefügt. Die Variable `transformers` kommt aus der Klasse `MessagePart`, dargestellt in Codeauszug 52. Die Variable `b64` kommt ursprünglich auch aus dieser Klasse, allerdings ist diese Variable nicht im Ergebnis vorhanden. Deshalb lässt sich in diesem Fall nur vermuten, dass es sich höchstwahrscheinlich um die Kodierung zu Base64 handelt.

```
1  public class Content extends MessagePart {
2    private static final AtomicInteger NEXT_ID = new AtomicInteger(0);
3
4    [...]
5
6    public static final int DATA = 2;
7
8    [...]
9
10   public Content(String data) throws IOException {
11     if (data == null)
12       throw new IllegalArgumentException(DialogHandler.text.getString(↩
             LanguageTextEntries.invalid_firstargument.name()) + " data");
13     setRefID("content" + NEXT_ID.getAndIncrement());
14     load(new ByteArrayInputStream(data.getBytes(Constants.CHAR_ENCODING)))↩
           ;
15     contentType = DATA;
16     transformers.add(b64);
17   }
18
19   [...]
20 }
```

Codeauszug 51: Konstruktor der Klasse `Content` nach dem Forward-Slicing

```
1  public abstract class MessagePart {
2    [...]
3    protected Vector<String> transformers = new Vector<String>();
4    [...]
```

```
5 }
```

Codeauszug 52: Klasse `MessagePart` nach dem Forward-Slicing

Das entstandene `Content`-Objekt wird der Methode `addContent` übergeben (vgl. Codeauszug 50, Zeile 5). Die relevanten Progrmmausschnitte sind in Codeauszug 53 aufgelistet. Die Methode ruft `addContentInternal` auf. Die Methode `addContentInternal` führt erst einige Überprüfungen durch. Es wird überprüft, ob bereits eine Signierung des Inhalts stattgefunden hat und somit kein weiterer Inhalt hinzukommen kann, denn sonst würde die Signatur nicht für den gesamten Inhalt gelten. Desweiteren wird überprüft, ob der Referenz Identifikator mit dem des Inhalts übereinstimmt und ob der Inhalt bereits in dem Container vorhanden ist. Anschließend wird auf einen duplizierten Identifikator überprüft. Schlussendlich wird der Inhalt dem Container hinzugefügt. Damit ist der Aufruf in Codeauszug 50 aus Zeile 5 vollendet.

```
1  public class ContentContainer extends MessagePart implements Serializable ↩
       {
2    [...]
3
4    Vector<OSCISignature> signerList = new Vector<OSCISignature>();
5
6    [...]
7
8    public void addContent(Content content) {
9      addContentInternal(content, true);
10   }
11
12   void addContentInternal(Content content, boolean checkForDuplicateId) {
13     if ((signerList.size() > 0) && (stateOfObject == ↩
         STATE_OF_OBJECT_CONSTRUCTION))
14     throw new IllegalStateException(DialogHandler.text.getString("↩
         signature_violation"));
15     boolean containsElement = contentList.contains(content);
16     if (ParserHelper.isSecureContentDataCheck() && this.getRefID() != null
17     && this.getRefID().equals(content.getRefID())) {
18       throw new IllegalArgumentException("refId " + content.getRefID() + " ↩
           equals ContentContainer ID "
19       + this.getRefID());
20     }
```

```
21
22      if (!containsElement) {
23        if (checkForDuplicateId && listContainsRefid(content.getRefID())) {
24          throw new IllegalArgumentException("refId " + content.getRefID() + ↩
                " is already in ContentContainer");
25        }
26
27        contentList.add(content);
28
29        [...]
30      }
31    }
32
33  [...]
34  }
```

Codeauszug 53: Konstruktor der Klasse `ContentContainer` nach dem Forward-Slicing

In Zeile 7 (vgl. Codeauszug 50) wird ein weiterer `addContent`-Aufruf durchgeführt, dabei wird ein Objekt vom Typ `Attachment` dem Konstruktor der Klasse `Content` als Parameter übergeben. Im Codeauszug 54 sind zwei Konstruktoren der rekonstruierten Klasse `Attachment` aufgelistet. Dem Konstruktor der Klasse `Attachment` wird die anzuhängende Datei, der Pfad und eine Konstante `DEFAULT_SYMMETRIC_CIPHER_ALGORITHM` aus der Klasse `Constants` übergeben. Im Ergebnis des Analysewerkzeuges ist diese Klasse der Konstanten nicht enthalten, damit ist nicht festzustellen, welcher Algorithmus als Standard festgelegt wurde und hier verwendet wird.

Im Codeauszug 54 fällt auf, dass der rekonstruierte Konstruktor in Zeile 1 nicht korrekt rekonstruiert wurde. Einige Übergabeparameter des stattfindenden Konstruktoraufrufs fehlen. Es wird vermutet, dass der Zweite im Codeauszug aufgelistete Konstruktor aufgerufen wird, aufgrund dessen das nur dieser eine passende Signatur besitzt.

Im Konstruktor finden null-Überprüfungen der übergebenden Parameter `ins`, `refId` und `symmetricCipherAlgorithm` statt. Dabei ist festzustellen, dass in der zweiten `if`-Anweisungen wohl ein falscher Enumeration-Eintrag der `LanguageTextEntries` verwendet wird. Der Wert der Variable `ivLength` und `symmetricCipherAlgorithm` ist nicht bekannt. Der Wert der Variable `secretKey` wird innerhalb des Konstruktor aus Zeile 54 erzeugt. Es wird überprüft, ob `secretKey` null ist, sollte der Schlüssel null sein, wird ein symmetrischer Schlüssel erzeugt. Andernfalls wird überprüft, ob der Schlüssel durch einen unterstützen Algorithmus erzeugt wurde (in diesem Fall DESede oder AES).

Der Codeauszug 55 zeigt die Methode, welche zum Erzeugen des symmetrischen Schlüssels verwendet wird. Es findet eine Übersetzung der Algorithmusbezeichnungen von OSCI zu JCA/JCE statt, ein `KeyGenerator` wird erzeugt und dieser wird mit der Schlüsselgröße initialisiert, um schlussendlich einen Schlüssel zu erzeugen.

```
1  public Attachment(java.io.InputStream ins, String refId, String ↩
       symmetricCipherAlgorithm) throws IllegalArgumentException, IOException↩
       , NoSuchAlgorithmException {
2    this(ins, refId, de.osci.osci12.encryption.Crypto.createSymKey(↩
         symmetricCipherAlgorithm),
3  }
4
5  public Attachment(java.io.InputStream ins, String refId, SecretKey ↩
       secretKey, String symmetricCipherAlgorithm, int ivLength) throws ↩
       IllegalArgumentException, IOException, NoSuchAlgorithmException {
6    if (ins == null) {
7      throw new IllegalArgumentException(DialogHandler.text.getString(↩
           LanguageTextEntries.invalid_firstargument.name()) + " ins");
8    }
9
10   if (refId == null) {
11     throw new IllegalArgumentException(DialogHandler.text.getString(↩
           LanguageTextEntries.invalid_firstargument.name()) + " refId");
12   }
13
14   if (symmetricCipherAlgorithm == null) {
15     throw new IllegalArgumentException(DialogHandler.text.getString("↩
           invalid_fourthargument")
16     + " symmetricCipherAlgorithm");
17   }
18
19   this.ivLength = ivLength;
20   this.symmetricCipherAlgorithm = symmetricCipherAlgorithm;
21   if (secretKey == null) {
22     this.secretKey = de.osci.osci12.encryption.Crypto.createSymKey(↩
           symmetricCipherAlgorithm);
23   }else{
```

```
24      if (!secretKey.getAlgorithm().equals("DESede") && !secretKey.↩
            getAlgorithm().equals("AES"))
25        throw new IllegalArgumentException(DialogHandler.text.getString("↩
              encryption_algorithm_not_supported"));
26      this.secretKey = secretKey;
27    }
28
29    [...]
30  }
```

Codeauszug 54: Konstruktoren der Klasse `Attachment` nach dem Forward-Slicing

```
1  public class Crypto {
2    public static javax.crypto.SecretKey createSymKey(String algorithm) ↩
         throws NoSuchAlgorithmException {
3      String algo = Constants.JCA_JCE_MAP.get(algorithm);
4      KeyGenerator keyGenerator;
5      if (DialogHandler.getSecurityProvider() == null)
6        keyGenerator = javax.crypto.KeyGenerator.getInstance(algo.substring↩
             (0, algo.indexOf('/')));
7      else
8        keyGenerator = javax.crypto.KeyGenerator.getInstance(algo.substring↩
             (0, algo.indexOf('/')), DialogHandler.getSecurityProvider());
9      [Schlüsselgenerierung aufgrund von Schlüsselgröße]
10     return keyGenerator.generateKey();
11   }
12 }
```

Codeauszug 55: Methode `createSymKey` der Klasse `Crypto` nach dem Forward-Slicing

Der Aufruf in Zeile 7 (vgl. Codeauszug 50) wird im Folgenden zusammengefasst. Es wird ein `Attachment`-Objekt erzeugt, dieses Objekt wird innerhalb eines `Content`-Objektes verpackt und anschließend an die Variable `encrypted_container` angehängt.

Der Codeauszug 56 zeigt die nächsten zu untersuchenden Zeilen. In diesen Zeilen wird ein Objekt vom Typ `Author` erzeugt. Als Parameter wird ein `PKCS12Signer`-Objekt und `null` übergeben. Anschließend wird die Methode `sign` des Containers aufgerufen.

```
1  Author author = new Author(new de.osci.osci12.samples.impl.crypto.↩
       PKCS12Signer("/de/osci/osci12/samples/zertifikate/test_alice_signature↩
       .p12",
```

```
2 "123456"), null);
3 encrypted_container.sign(author);
```

Codeauszug 56: Auschnitt der Klasse `AuthorOriginatorContentInterchange` nach dem Forward-Slicing

Die Klasse `Author` repräsentiert einen Autor, hält Informationen über diesen Autor und bietet einen eindeutigen Identifikator. Die Klasse erbt von der Klasse `Role`, welche eine bestimmte Rolle im OSCI-Protokoll repräsentiert.

Im Codeauszug 57 ist ein Teil der Klasse `PKCS12Signer` aufgelistet. Der Konstruktor in Zeile 5 ruft den Konstruktor in Zeile 9 auf, dieser lädt den `keyStore` und iteriert durch diesen, um den richtigen Schlüssel zu finden. Im Konstruktor wird deutlich, dass der übergebende String „123456“der PIN des eingegebenen PKCS12 Schlüsselspeichers ist.

```
1  public class PKCS12Signer extends de.osci.osci12.extinterfaces.crypto.↩
       Signer {
2    [...]
3
4    public PKCS12Signer(String p12_fileName, String pin)
5    throws KeyStoreException, CertificateException, NoSuchAlgorithmException↩
       , IOException {
6      this(PKCS12Signer.class.getResourceAsStream(p12_fileName), pin, false)↩
         ;
7    }
8
9    public PKCS12Signer(InputStream in, String pin, boolean usePSSforRSAkey)↩
         throws KeyStoreException, CertificateException, ↩
       NoSuchAlgorithmException, IOException {
10     this.pin = pin.toCharArray();
11     [keyStore Variable setzen]
12
13     keyStore.load(in, this.pin);
14     String al = null;
15     Enumeration<String> e = keyStore.aliases();
16     while (e.hasMoreElements())
17       if (keyStore.isKeyEntry(al = e.nextElement()))
18         break;
19
20     if (al == null)
```

```
21      throw new NullPointerException("No private key found in keystore.");
22    [Klasseattribute setzen]
23  }
24 }
```

Codeauszug 57: Methode `createSymKey` der Klasse `Crypto` nach dem Forward-Slicing

Im Codeauszug 58 ist die Methode `sign` der Klasse `ContentContainer` abgebildet. Die aufgerufene Methode, welche als Übergabeparameter nur eine Role erwartet, ruft eine weitere Methode mit anderer Signatur auf. Diese Methode erzeugt eine `OSCISignature` und fügt, falls schon ein Signierer vorhanden ist, die Referenzen dieses Signierers der Signatur hinzu. Andernfalls werden die Referenzen aus denen des `ContentContainer` hinzugefügt.

```
1  public void sign(Role signer)
2  throws OSCIException, NoSuchAlgorithmException, SignatureException, ↩
       IOException {
3    sign(signer, DialogHandler.getDigestAlgorithm(), null);
4  }
5
6  @Deprecated
7  public void sign(Role signer, String digestAlgorithm, String time) throws ↩
       OSCIException, NoSuchAlgorithmException, SignatureException, ↩
       IOException {
8    [...]
9
10   OSCISignature sig = new OSCISignature();
11   if (signerList.size() == 0) {
12     [...]
13     Content cnt;
14     for ( int i = 0 ; i < contentList.size() ; i++ ) {
15       cnt = contentList.get(i);
16       [...]
17       addSignatureReference(sig, cnt, digestAlgorithm);
18     }
19
20     addAttachmentSigRefs(sig, this, digestAlgorithm);
21     EncryptedDataOSCI enc;
22     for ( int i = 0 ; i < encryptedDataList.size() ; i++ ) {
```

```
23        enc = encryptedDataList.get(i);
24        [...]
25        addSignatureReference(sig, enc, digestAlgorithm);
26      }
27    }else{
28      [Signaturreferenzen der ersten Signierers werden der Signatur hinzugef↩
            ügt]
29    }
30
31    [Zeitstempel für Signatur]
32    sig.sign(signer);
33    signerList.add(sig);
34  }
```

Codeauszug 58: Methode `sign` der Klasse `ContentContainer` nach dem Forward-Slicing

Die verwendete Hashfunktion ist im rekonstruierten `DialogHandler` in der statischen Methode `getDigestAlgorithm`, welche im Codeauszug 59 aufgelistet ist, zu erkennen. Es wird SHA256 verwendet, um den Hashwert der Signaturreferenz zu erzeugen. Auf eine genauere Betrachtung der Methode `addSignatureReference` wird hier verzichtet.

```
1  public class DialogHandler {
2    private static String digestAlgorithm = Constants.↩
         DIGEST_ALGORITHM_SHA256;
3
4    [...]
5
6    public static String getDigestAlgorithm() {
7      return digestAlgorithm;
8    }
9
10   [...]
11 }
```

Codeauszug 59: Methode `sign` der Klasse `ContentContainer` nach dem Forward-Slicing

Im Codeauszug 58 ist zu erkennen, dass die Signatur nur die Referenzen der Inhalte enthält. Möglicherweise ist es möglich die Signatur für weitere Nachrichten mit Inhalten der gleichen Referenz zu verwenden. Damit enthält der Container (vgl. Codeauszug 49) eine Signatur, welche die Referenzen der Inhalte signiert. Im Codeauszug 60 sind die folgenden

zu untersuchenden Zeilen aufgelistet. Mithilfe des vorher signierten Containers wird ein `EncryptedDataOSCI`-Objekt erstellt. Dabei wird angegeben mit welchem Algorithmus die Daten verschlüsselt werden. In diesem Fall sollen die Daten symmetrisch mit AES256 im Galois-Counter-Mode (GCM) verschlüsselt werden. Anschließend wird ein `Reader`-Objekt, welcher das Zertifikat von *Dave* einliest, erzeugt. Das `Reader`-Objekt wird als Parameter an die Methode `encrypt` des `EncryptedDataOSCI`-Objektes übergeben.

```
1  EncryptedDataOSCI encryptedData = new EncryptedDataOSCI(Constants.↩
       SYMMETRIC_CIPHER_ALGORITHM_AES256_GCM,
2  encrypted_container);
3
4  Reader reader = new Reader(de.osci.helper.Tools.createCertificate(↩
       getClass().getResourceAsStream("/de/osci/osci12/samples/zertifikate/↩
       test_dave_cypher.cer")));
5  encryptedData.encrypt(reader);
6  forwardDel.addEncryptedData(encryptedData);
```

Codeauszug 60: Ausschnitt der `createStoredMessageFile`-Methode der Klasse `Author OriginatorContentInterchange` nach dem Forward-Slicing

Im Codeauszug 61 ist die Methode `encrypt` der Klasse `EncryptedDataOSCI` dargestellt. Die aufgerufene Methode `encrypt`, welche als Parameter nur eine `Role` erwartet, ruft eine weitere `encrypt`-Methode auf. Diese bekommt die Konstante `DEFAULT_ASYMMETRIC_CIPHER_ALGORITHM` als weiteren Übergabeparameter übergeben. Da die Klasse `Constants` nicht im Ergebnis vorhanden ist, kann keine Aussage über die verwendeten Algorithmen gemacht werden.

In der schlussendlich aufgerufenen `encrypt`-Methode wird der Zustand des Objektes verändert und der im Konstruktor der Klasse `EncryptedDataOSCI` (aus Lesbarkeit ausgelassen) erzeugte symmetrische Schlüssel verschlüsselt. Die dafür aufgerufene Methode lässt zumindest eine Vermutung über den verwendeten asymmetrischen Verschlüsselungsalgorithmus zu. Die Methode heißt `doRSAEncryption`, deshalb wird vermutet das der verwendete Algorithmus RSA ist. Anschließend findet ein weiterer Aufruf einer anderen `encrypt`-Methode (im Codeauszug ausgelassen) statt. In der besagten Methode finden einige Überprüfungen statt und es wird die Methode `createEncryptedAttachments` aufgerufen, welche verschiedene Objekte erzeugt.

```
1  public class EncryptedDataOSCI extends MessagePart {
2    public void encrypt(Role reader) throws OSCICipherException, ↩
         OSCIRoleException, IOException, NoSuchAlgorithmException {
```

```
3       encrypt(reader, Constants.DEFAULT_ASYMMETRIC_CIPHER_ALGORITHM);
4     }
5
6     public void encrypt(Role reader, String algorithm) throws ↩
          OSCICipherException, OSCIRoleException, IOException, ↩
          NoSuchAlgorithmException {
7       [...]
8
9       this.stateOfObject = ENCRYPTEDDATA_ENCRYPTED;
10
11      [Debugausgaben und null-Überprüfung]
12
13      byte[] encryptedSymKey = Crypto.doRSAEncryption(reader.↩
            getCipherCertificate(), secretKey, algorithm);
14      encrypt(encryptedSymKey, reader, algorithm);
15    }
16  }
```

Codeauszug 61: Ausschnitte der Klasse `EncryptedDataOSCI` nach dem Forward-Slicing

Der Codeauszug 62 zeigt die letzten zu analysierenden Zeilen der `createStoredMessage File`-Methode. In diesen Zeilen wird die Nachricht in eine Datei geschrieben und das dazugehörige `File`-Objekt zurückgegeben.

```
1 File store = new File(path + "/StoredForwardDelivery.osci");
2 StoredMessage.storeMessage(forwardDel, new FileOutputStream(store));
3 return store;
```

Codeauszug 62: Ausschnitt der `createStoredMessageFile`-Methode der Klasse `Author OriginatorContentInterchange` nach dem Forward-Slicing

Im Codeauszug 63 ist die `storeMessage`-Methode der Klasse `StoredMessage` abgebildet. Diese ruft die Methode `writeXML` der Klasse `OSCIMessage` auf. Der Codeauszug 64 zeigt diese Methode. Die Methode macht einige Vorbereitung für das Schreiben von XML, um dann `writeXML`-Methoden verschiedener Teile der Nachricht aufzurufen. Im Ergebnis sind diese Methoden nicht vorhanden. Im originalen Quelltext definiert die Klasse `MessagePart` die abstrakte Methode `writeXML` und die verschiedenen Nachrichtenteile implementieren diese Methode.

Der Aufruf der `writeXML`-Methode wird im originalen Quelltext durch verschiedene Klassen geleitet und landet schlussendlich bei der Klasse `CipherValue`, welche die Daten

mithilfe des `SymCipherOutputStream` verschlüsselt. Diese Klassen und Methoden fehlen im Ergebnis, möglicherweise hat die Points-To Analyse in diesem Fall nicht alle möglichen Klassen verschiedener Variablen korrekt aufgelöst.

```
1 public static void storeMessage(OSCIMessage msg, OutputStream output) ↩
     throws IOException, OSCIException, NoSuchAlgorithmException {
2   if (msg instanceof StoredMessage)
3   throw new UnsupportedOperationException();
4   msg.writeXML(output);
5 }
```

Codeauszug 63: Methode `storeMessage` der Klasse `StoredMessage` nach dem Forward-Slicing

```
1  void writeXML(OutputStream out) throws IOException, de.osci.osci12.↩
      OSCIException, NoSuchAlgorithmException {
2    [Vorbereitungen für XML]
3
4    for ( int i = 0 ; i < (messageParts.size() - 1) ; i++ ) {
5      if (messageParts.get(i) != null)
6      MessagePartsFactory.writeXML(messageParts.get(i), out);
7    }
8
9    [...]
10
11   MessagePartsFactory.writeXML((Body)messageParts.lastElement(), out);
12
13   [...]
14
15   for ( Attachment att : attachments.values() ) {
16     MessagePartsFactory.writeXML(att, out);
17   }
18
19     [...]
20 }
```

Codeauszug 64: Methode `writeXML` der Klasse `OSCIMessage` nach dem Forward-Slicing

Damit ist die Analyse der Methode `createStoredMessageFile` aus der Klasse `Author OriginatorContentInterchange` (Codeauszug 49) abgeschlossen. Die Analyse hat Auf-

schlüsse über die entstandene Struktur der Nachricht ermöglicht und zeigt wie und mit welchen Algorithmen die Nachricht aufgebaut, signiert und verschlüsselt wird. Einige relevante Teile des originalen Programms fehlen allerdings im Ergebnis. Aufgrund dessen sind einige Schlüsse über verwendete Algorithmen und die Struktur der anschließend verschlüsselten Daten nicht möglich oder lassen nur Vermutungen zu. Ein Vergleich zwischen der Komplexität des eigentlichen Quelltextes und dem Ergebnis des Analysewerkzeuges zeigt, dass das Analysewerkzeug den zu analysierenden Quelltext deutlich reduziert. Der originale Programmcode hat 12.628 Lines of Code (LoC), also Quelltextzeilen. Das Ergebnis des Analysewerkzeuges enthält dagegen 2.695 LoC. Auch die Anzahl der Klassen ist von 160 auf 52 reduziert worden, ebenso die Anzahl der Funktionen von 1.289 zu 242.

Die Klasse `AuthorOriginatorContentInterchange` wurde auch mit den Slicing-Techniken Backward-, Thin-Backward-Slicing und Chopping untersucht. Hierbei wurde ausschließlich ein Teil der Klasse `AuthorOriginatorContentInterchange` rekonstruiert. Dadurch geben diese Techniken keinen Aufschluss über die Struktur der entstandenen Daten oder verwendeten kryptografischen Funktionen sowie Algorithmen und den benutzen Geheimnissen, wie Schlüssel oder Schlüsselspeicherpins.

Im Codeauszug 65 ist der Quelltext der Klasse `MultiFetchDelivery` aufgelistet. In der `main`-Methode des Beispielprogramms werden mehrere `StoreDelivery` Nachrichten versendet. Diese geben dem Intermediär zu verstehen, dass einen Nachricht zu speichern ist, bis der Kommunikationspartner diese abruft. Das Programm modelliert also die asynchrone Kommunikation über den Intermediär. Die `FetchDelivery` Nachricht ermöglicht das Abrufen einer Nachricht beim Intermediär.

```
1 public class MultiFetchDelivery {
2   Intermed intermed;
3
4   public MultiFetchDelivery(String intermedURL) throws ↩
        CertificateException, URISyntaxException {
5     java.security.cert.X509Certificate intermedCipherCert = de.osci.helper↩
          .Tools.createCertificate(getClass().getResourceAsStream("/de/osci/↩
          osci12/samples/zertifikate/test_osci-manager_cypher.cer"));
6
7     intermed = new Intermed(null, intermedCipherCert, new java.net.URI(↩
          intermedURL));
8   }
9
```

```
10  public ResponseToStoreDelivery sendStoreDelivery() throws ↩
        GeneralSecurityException, IOException, OSCIException {
11    [...]
12  }
13
14  public ResponseToFetchDelivery[] sendFetchDelivery(String messageId) ↩
        throws GeneralSecurityException, IOException, OSCIException {
15    [...]
16  }
17
18  public ResponseToFetchProcessCard sendFetchProcessCard(String messageId)↩
       throws GeneralSecurityException, IOException, OSCIException {
19    [...]
20  }
21
22  public static void main(String[] args) {
23    try {
24      MultiFetchDelivery scenario_1 = new MultiFetchDelivery(args[0]);
25      ResponseToStoreDelivery responseStore = scenario_1.sendStoreDelivery↩
            ();
26      responseStore = scenario_1.sendStoreDelivery();
27      responseStore = scenario_1.sendStoreDelivery();
28      responseStore = scenario_1.sendStoreDelivery();
29      System.out.println("\nResponseToStoreDelivery:\n" + responseStore.↩
            toString());
30
31      ResponseToFetchDelivery responseFetchDel =null;// scenario_1.↩
            sendFetchDelivery(responseStore.getMessageId());
32      Reader reader = new Reader(new de.osci.osci12.samples.impl.crypto.↩
            PKCS12Decrypter("/de/osci/osci12/samples/zertifikate/↩
            test_dave_cypher.p12",
33      "123456"));
34      System.out.println("\nResponseToFetchDelivery:\n" + responseFetchDel.↩
            toString());
35
36      [...]
37      // When a message of unknown content structure is received, the ↩
            content types must
```

```
38        // be analyzed (e.g. using Content.getContentType())
39        System.out.println("\nCONTENT DATA:\n" +
40        responseFetchDel.getContentContainer()[0].getContents()[0].↩
              getContentData());
41
42        InputStream in = responseFetchDel.getContentContainer()[0].↩
              getContents()[1].getAttachment().getStream();
43        StringBuffer attachment = new StringBuffer();
44
45        [...]
46
47        System.out.println("\nATTACHMENT:\n" + attachment);
48        System.out.println("\nRESULT OF SIGNATURE CHECK: " +
49        responseFetchDel.getContentContainer()[0].checkAllSignatures());
50        System.out.println("\nENCRYPTED CONTENT DATA:\n" +
51        responseFetchDel.getEncryptedData()[0].decrypt(reader).getContents()↩
              [0].getContentData());
52      }catch (Exception ex) {
53        ex.printStackTrace();
54      }
55    }
56  }
```

Codeauszug 65: Ausschnitt der der Klasse `MultiFetchDelivery`

Der Codeauszug 66 zeigt die Klasse `MultiFetchDelivery` nach dem Forward-Slicing. Dabei wurden die Zeilen 47 bis 50 als Seed-Statements verwendet. Auffallend ist, dass die Anweisungen mit der Variable `responseFetchDel` nicht im Ergebnis vorhanden sind. Allerdings fällt im Original (vgl. Codeauszug 65) auf, dass diese Zeile tatsächlich auskommentiert ist. Das bedeutet die Abhängigkeiten zu dieser Variable fehlen und das erklärt weshalb die besagten Anweisungen nicht im Ergebnis vorhanden sind. Es bleibt die Methode `sendStoreDelivery`, welche in der `main`-Methode 4 mal aufgerufen wird.

Die Methode `sendStoreDelivery` erstellt zuerst ein `Originator`-Objekt. Der Konstruktor erwartet ein `Signer`- und `Decrypter`-Objekt. Die Klassen `Signer` und `Decrypter` sind im Ergebnis enthalten, allerdings leer. Die Klasse `Originator` ist im Ergebnis vorhanden und der passende Konstruktor wurde rekonstruiert. Dem Konstruktor werden Objekte vom Typ `PKCS12Signer` und `PKCS12Decrypter` übergeben. Beim `PKCS12Signer` wird das Zertifikat von *Alice* als Übergabeparameter mitgegeben. Der `PKCS12Decrypter` erhält

auch ein Zertifikat von *Alice*. Es scheint, als wenn das OSCI Protokoll für das Unterschreiben und das Verschlüsseln verschiedene Zertifikate verwendet. Das `Originator`-Objekt repräsentiert die Herkunft der Nachricht.

Nach der Erzeugung des `Originator`-Objektes wird ein `Addressee`-Objekt erzeugt. Die Klasse erwartet im Konstruktor ein Signierzertifikat und ein Verschlüsselungszertifikat. Die Klasse `Addressee` und der Konstruktor sind rekonstruiert worden. Das Signierzertifikat wird auf `null` gesetzt, dass bedeutet das die Signatur einer Rückantwort nicht mit diesem Objekt geprüft werden soll. Da im Fall des `MultiFetchDelivery` Beispielprogramms die asynchrone Kommunikation abgebildet wird, ist dieser Übergabeparameter valide. Als Verschlüsselungszertifikat wird das Zertifikat von *Bob* übergeben.

Nach der Erzeugung des `Addressee`-Objektes wird eine `StoreDelivery` Nachricht erzeugt. Die genannte Nachricht erhält einen Betreff und es wird unverschlüsselter (betreffende Zeilen wurden ausgelassen) und verschlüsselter Inhalt hinzugefügt. Der Inhalt wird, wie im vorherigen Beispielprogramm (vgl. Codeauszug 61), durch das symmetrische Verfahren AES256 im Modus GCM verschlüsselt. Es scheint, als sei diese Angabe der Verschlüsselung nur zum temporären Speichern des Anhanges auf dem Dateisystem und nicht zur Transportverschlüsselung.

Nach dem Hinzufügen von den Inhalten und Anhängen der Nachricht wird ein Objekt vom Typ `EncryptedDataOSCI` erzeugt. Dieses Objekt verschlüsselt die Daten mit dem symmetrischen Verfahren AES256 im GCM Modus. Anschließend werden die verschlüsselten Inhalte und Anhänge der Nachricht `StoreDelivery` hinzugefügt.

```
1  public class MultiFetchDelivery {
2    public MultiFetchDelivery(String intermedURL) throws ↩
         CertificateException, URISyntaxException {}
3
4    public ResponseToStoreDelivery sendStoreDelivery() throws ↩
         GeneralSecurityException, IOException, OSCIException {
5      Originator user_1 = new Originator(new de.osci.osci12.samples.impl.↩
           crypto.PKCS12Signer("/de/osci/osci12/samples/zertifikate/↩
           test_alice_signature.p12",
6      "123456"),
7      new de.osci.osci12.samples.impl.crypto.PKCS12Decrypter("/de/osci/↩
           osci12/samples/zertifikate/test_alice_cypher.p12",
8      "123456"));
9
10     [...]
```

```
11
12      Addressee user_2 = new Addressee(null,
13      de.osci.helper.Tools.createCertificate(getClass().getResourceAsStream(↩
            "/de/osci/osci12/samples/zertifikate/test_bob_cypher.cer")));
14
15      StoreDelivery storeDel = new StoreDelivery(clientDialog, user_2, ↩
            rsp2GetMsgID.getMessageId());
16      storeDel.setSubject("Subject");
17
18      [...]
19
20      storeDel.addContentContainer(not_encrypted_container);
21
22      Reader reader = new Reader(de.osci.helper.Tools.createCertificate(↩
            getClass().getResourceAsStream("/de/osci/osci12/samples/zertifikate↩
            /test_dave_cypher.cer")));
23
24      ContentContainer encrypted_container = new ContentContainer();
25      encrypted_container.addContent(new Content("Any encrypted content data↩
            ."));
26      encrypted_container.addContent(new Content(new Attachment(new ↩
            ByteArrayInputStream("Any encrypted attachment data.".getBytes()),
27      "enc_test.txt",
28      Constants.SYMMETRIC_CIPHER_ALGORITHM_AES256_GCM)));
29
30      EncryptedDataOSCI encryptedData = new EncryptedDataOSCI(Constants.↩
            SYMMETRIC_CIPHER_ALGORITHM_AES256_GCM,
31      encrypted_container);
32      encryptedData.encrypt(reader);
33      storeDel.addEncryptedData(encryptedData);
34
35      ResponseToStoreDelivery rsp2StoreDel = storeDel.send();
36
37      [Fehlerbehandlung]
38
39      return rsp2StoreDel;
40    }
41
```

```
42   public static void main(String[] args)
43   {
44     try
45     {
46       MultiFetchDelivery scenario_1 = new MultiFetchDelivery(args[0]);
47       ResponseToStoreDelivery responseStore = scenario_1.sendStoreDelivery↩
             (); // Seed-Statement
48       responseStore = scenario_1.sendStoreDelivery();
49       responseStore = scenario_1.sendStoreDelivery();
50       responseStore = scenario_1.sendStoreDelivery();
51       System.out.println("\nResponseToStoreDelivery:\n" + responseStore.↩
             toString());
52       ResponseToFetchProcessCard responseFetchProcCard = scenario_1.↩
             sendFetchProcessCard(responseStore.getMessageId());
53     }
54
55     catch (Exception ex)
56     {
57       ex.printStackTrace();
58     }
59   }
60 }
```

Codeauszug 66: Ausschnitt der der Klasse `MultiFetchDelivery` nach dem Forward-Slicing

Das Beispielprogramm `MultiFetchDelivery` wurde zusätzlich mit dem Backward- und Thin-Backward-Slicing sowie Chopping untersucht. Diese Ergebnisse lassen keine der vorher mit dem Forward-Slicing gewonnenen Aufschlüsse über das Beispielprogramm zu.

Auf die genaue Analyse der Ergebnisse der weiteren Beispielprogramme wird verzichtet. Diese Beispielprogramme erzeugen verschlüsselte Daten und Nachrichten nach dem selben Muster der vorherigen Beispiele. In den Ergebnissen sind einige verwendete Algorithmen und Abläufe zu erkennen. Andere, wie beispielsweise die `Constants`-Klasse, fehlen auch hier in den Ergebnissen. Auch die Techniken verhalten sich ähnlich, beim Forward-Slicing werden deutlich bessere, überhaupt für die Analyse geeignete Ergebnisse erzeugt. Beim Backward- und Thin-Backward-Slicing sowie Chopping fehlen verwendete Klassen und bei den meisten Beispielprogrammen sind lediglich Teile der `main`-Methode im Ergebnis vorhanden und rekonstruiert worden.

In diesem Kapitel wurden verschiedene Beispielprogramme im Bereich der Software-Sicherheit mithilfe des Analysewerkzeuges untersucht und anschließend analysiert. Dafür wurden vorher Forschungsfragen formuliert. Diese Forschungsfragen dienten als Zielvorgabe für die Analysen der Ergebnisse. Hervorgehoben wurden Besonderheiten oder relevante aber fehlende Programmteile. Zusätzlich wurden die Ergebnisse der verschiedenen Slicing-Techniken analysiert und verglichen. Im folgenden Kapitel 6 werden die Ergebnisse der Fallstudien und Benchmarks diskutiert.

6 Diskussion

Das vorliegende Kapitel diskutiert die Ergebnisse der Evaluationen aus Kapitel 5 und ordnet den Einfluss der implementierten Funktionalität aus Kapitel 4 auf die Evaluierungen ein.

Im Kapitel 5.1 wurde der Einfluss verschiedener Einstellungen der JVM und die Verwendung der GraalVM auf die Berechnungszeit der Ergebnisse des Analysewerkzeuges untersucht. Festzustellen ist, dass die JVM mit ihren Standardeinstellungen in diesem Fall meistens die kürzesten Berechnungszeiten liefert. Die GraalVM hatte bei keinem der untersuchten Programme eine kürzere Berechnungszeit als die JVM mit ihren Standardeinstellungen. Auch die anderen Einstellungen der JVM haben kein Muster ergeben bei welchem diese eine signifikante Veränderung der Berechnungszeit zu verbuchen hätten.

Für die Evaluierung der Berechnungszeit muss allerdings erwähnt werden, dass hier die Einstellungen, welche aus [Cyl19] stammen, als einzige für die Evaluierung verwendet wurden. Einige Evaluierungen sollten erst mit bestimmten Optionen auf `FULL` durchgeführt werden, diese führten aber zu `OutOfHeapSpaceException`s. Aus den Evaluierungen ist zu erkennen, dass bei Berechnungszeiten, welche zwischen wenigen Millisekunden und einigen Sekunden liegen, keine Notwendigkeit der Änderungen von Einstellungen vorliegt. Nicht klar ist, ob eine signifikante Verbesserung bei Programmen und Optionen, welche mehrere Stunden, Tage oder sogar mehr Berechnungszeit benötigen, zu verzeichnen wäre.

Das Kapitel Kapitel 5.2 analysiert die Ergebnisse der verschiedenen Beispielprogramme, stellt Forschungsfragen auf und unternimmt den Versuch diese mithilfe einer Analyse der Ergebnisse zu beantworten. Daneben wurde ein Bezug zu der implementierten Funktionalität aus Kapitel 4 hergestellt und Besonderheiten hervorgehoben.

Die Analyse der Ergebnisse des Analysewerkzeuges zeigt auf, dass die gewählten Optionen, Slicing-Technik und Seed-Statement mitunter riesigen Einfluss auf die Komplexität und Verwendbarkeit der Ergebnisse hat. Es wird deutlich, dass keine Kombination der verschiedenen Parameter für alle Beispielprogramm gleichermaßen für die nachfolgende Analyse geeignete Ergebnisse liefert.

Allerdings wird deutlich, dass eine korrekte Rekonstruktion der Ergebnisse wichtig ist. Gerade die Rekonstruktion von Variablendefinitionen und Klassenattributen, welche im Rahmen der Arbeit implementiert wurde, bieten für die darauffolgende Analyse wichtige Aufschlüsse und erleichtern das Programmverständnis. Der Einfluss der Rekonstruktion wird deutlich bei inkorrekt rekonstruierten Anweisungen, denn wird ein Teil der Anweisung nicht rekonstruiert, beispielsweise weil die Anweisung über mehrere Zeilen geht, fehlen

mitunter Übergabeparameter oder Variablen im Ergebnis und erschweren das Programmverständnis oder machen es unmöglich.

Die Rekonstruktionen sind nötig, da der WALA Framework bestimmte Anweisungen, wie Variablendefinitionen nicht in der Zwischendarstellung behandelt und damit fehlen diese im Ergebnis [Cong].

7 Fazit

Das folgende Kapitel fasst die einzelnen Kapitel der Arbeit und ihren Beitrag zur Forschung zusammen und zieht im Hinblick auf Implementierungen, Evaluierungen und Diskussion ein Fazit über die vorliegende Arbeit.

Das Kapitel 2 legte die Grundlage für das Verständnis der Arbeit und half die Arbeit im richtigen Kontext einzuordnen. Dafür stellte 2.1 die Grundlagen der Programmanalyse vor. Zudem wurden die Grundlagen der Slicing-Technik (2.2), des WALA Frameworks (2.3) und der Informationssicherheit (2.4) erklärt.

Anschließend wurde in Kapitel 3 auf verwandte Arbeiten eingegangen, dabei wurde die Historie der Entwicklung an der Universität Bremen chronologisch dargestellt und Arbeiten, welche die Slicing-Technik zur Verbesserung der Software-Sicherheit verwenden, zusammengefasst.

Im Rahmen dieser Arbeit wurden Erweiterungen am Analysewerkzeug implementiert und allgemeine Verbesserungen der Codequalität vorgenommen, welche in Kapitel 4 vorgestellt wurden. Dabei wurde die grafische Oberfläche des Analysewerkzeuges angepasst, interne Abhängigkeiten aufgelöst und diese auf die neusten Versionen gebracht. Zudem wurden verschiedene Slicing-Techniken implementiert und die Möglichkeit zum direkten Auswählen von Seed-Statements wurde geschaffen. Ferner wurden Refactorings und Erweiterungen der internen API vorgenommen, welche die Weiterentwicklung des Analysewerkzeuges ermöglichen.

Die Evaluierungen des Kapitel 5.1 untersuchten den Einfluss der verschiedenen JVM-Einstellungen und die GraalVM auf die Berechnungszeit des Analysewerkzeuges.

Einige Ergebnisse des Analysewerkzeuges wurden in Kapitel 5.2 gezeigt. Dabei wurden Forschungsfragen formuliert und die Ergebnisse des Analysewerkzeuges analysiert. Es wurde gezeigt, inwiefern die Ergebnisse des Analysewerkzeuges das Programmverständnis unterstützen und welchen Einfluss fehlerhafte Rekonstruktion dabei hat. Daneben wurden Besonderheiten hervorgehoben und interessante Implementierungen aufgezeigt.

Die Evaluierungen aus Kapitel 5 wurden im Kapitel 6 diskutiert. Dabei wurde festgestellt, dass die Implementierungen der Arbeit und die Ergebnisse des Analysewerkzeuges beim Programmverständnis unterstützen. Allerdings sind auch weitere Forschungsfragen aufkommen, welche im Rahmen der Arbeit unbeantwortet bleiben. Diese Forschungsfragen und ein Ausblick der Weiterentwicklungen des Analysewerkzeuges werden im folgenden Kapitel 8 gegeben.

Schlussendlich erforschte die Arbeit den Einfluss der implementierten Funktionalität, zeigte potenzielle Schwachpunkte der momentanen Implementierung auf, bot einen Ein-

blick in das Potenzial des Analysewerkzeuges durch die Analyse verschiedener Beispielprogramme, stellte die Ergebnisse aufbereitet dar, diskutierte die aufgezeigten Ergebnisse im Kontext der Software-Sicherheit und zieht daraus Schlüsse für folgende Arbeiten.

8 Ausblick

Das vorliegende Kapitel gibt einen Ausblick auf weitere Forschung im Bereich des Analysewerkzeuges und Slicing im Kontext der Software-Sicherheit. Der Fokus wird dabei auf Forschung im Zusammenhang mit dem in dieser Arbeit verwendeten und weiterentwickelten Analysewerkzeug gelegt.

Die Evaluierungen in Kapitel 5 und die anschließende Diskussion in Kapitel 6 haben das mögliche Potenzial des Analysewerkzeuges im Hinblick auf die Software-Sicherheit aufgezeigt. Trotzdem ist auch deutlich geworden, wie komplex die Implementierung der Bibliotheken ist und wie komplex die anschließende Analyse des Ergebnisses ist. Es bleibt nicht einfach einzuordnen, ob das Analysewerkzeug den Entwicklerinnen und Entwicklern bei der Suche von Sicherheitslücken unterstützen kann. Eine Expertenstudie könnte weitere Anwendungsmöglichkeiten des Analysewerkzeuges aufdecken und Verständnis für die nötige Unterstützung der Experten deutlich machen.

Im Rahmen dieser Arbeit wurde die grafische Oberfläche verändert und erweitert. Einige Prozesse im Analysewerkzeuges bleiben aber umständlich und verlangen Geduld. Aus diesem Grund könnte eine breitere Nutzerstudie Verbesserungsmöglichkeiten der Benutzbarkeit des Analysewerkzeuges aufzuzeigen. Eine Möglichkeit die Benutzbarkeit des Analysewerkzeuges zu steigern, wäre es das Werkzeug in eine Entwicklungsumgebung zu integrieren. Mithilfe dieser Integration müsste ein Entwickler seine gewohnte Umgebung nicht verlassen. Für diese Integration könnte das Language-Server Protokoll[5] interessant sein.

Die nachfolgende Analyse komplexer Programme wird durch die Ausgabe des SDG und CG nicht unterstützt, denn diese Graphen besitzen selbst für kleine Programme eine große Komplexität durch eine hohe Anzahl von Knoten. Es müsste untersucht werden, inwiefern die Ausgabe der anderen entstandenen Datenstrukturen, wie z. B. der `AnalysisScope` oder die `ClassHierarchy` und Graphen, wie beispielsweise dem CFG, CDG, DDG und PDG, die anschließende Analyse der Ergebnisse unterstützt. Zusätzlich könnte untersucht werden, inwieweit die Generierung von PDGs für ausgewählte Funktionen die Analyse komplexer Programmabläufe unterstützt.

Die Rekonstruktion wurde in dieser Arbeit erweitert und angepasst. Trotzdem ist die Rekonstruktion aller Variablen, möglicher aufgerufener Konstruktoren, lokaler Variablen und anderen Sprachkonstrukten komplex und möglicherweise für ein bestimmtes Programm unerwünscht. Aus diesem Grund wäre eine Untersuchung von Slicing-Techniken, welche speziell für objektorientierte Sprachen erforscht wurden, interessant. Diese könnten die

[5] https://microsoft.github.io/language-server-protocol/

Rekonstruktion von Quelltext obsolet machen und helfen die Ergebnisse möglichst minimal zu halten. Abseits dieser Untersuchung ist die Rekonstruktion ein weiterer Punkt für Forschung und Entwicklung. Bestimmte Sprachkonstrukte werden weiterhin nicht korrekt rekonstruiert. Möglicherweise wäre verschiedene Rekonstruktionskonzepte als Auswahl für den Benutzer interessant.

In den Evaluierungen dieser Arbeit wurden Optionen des Analysewerkzeuges verwendet, welche in vorherigen Arbeiten als im Verhältnis gute Einstellungen befunden wurden. Die Evaluierungen zeigten allerdings das diese Standardeinstellungen nicht für alle Anwendungsfälle das gewünschte Ergebnis liefern. Aus diesem Grund wäre eine Untersuchung von verschiedenen Kombinationen dieser Option (Kontrollabhängigkeiten, Datenabhängigkeiten, Reflections, CFA-Constraint, Exclusionfiles) interessant, möglicherweise können für bestimmte Slicing-Techniken erforschte Kombinationen als Voreinstellungen dem Benutzer zur Verfügung gestellt werden. Ein Vergleich der aus diesen Optionen erzeugten SDG könnte ein besseres Verständnis für den Einfluss dieser Optionen bringen.

Nicht alle möglichen Weiterentwicklungen, welche aus den Ausblicken der in 3.1 verwandten Arbeiten entstammen, wurden in dieser Arbeit umgesetzt. Einige der Möglichkeiten werden hier möglicherweise wiederholt und ergänzt. Kurz erwähnt sei der Ausblick auf einen integrierten Buildserver, welcher es ermöglicht lediglich Quelltext hochzuladen oder die Vorbereitung für das Analysieren von Android vereinfacht. Das WALA Framework unterstützt allerdings auch die Möglichkeit direkt den Quelltext zu analysieren.

Die Verwendung einer weiteren Analysebibliothek ist weiterhin ein möglicher Anhaltspunkt für weitere Forschung, auch im Hinblick auf Slicing-Techniken speziell für objektorientierte Programme [MMK06, SS14]. Aus den Evaluierungen dieser Arbeit ist zu entnehmen, dass nicht jede Slicing-Technik für jedes Programm funktioniert. Aus diesem Grund könnte eine Untersuchung der Kombination von Techniken durchgeführt werden, beispielsweise Chopping mithilfe Forward-Slicing, aber zusätzlich als Backward-Slicing das Thin-Backward-Slicing. Ein weiterer Untersuchungsgegenstand könnte sein, dass die Techniken gemischt auf die interprozedurale und intraprozedurale Analyse angewandt werden, beispielsweise wird der CG mithilfe des Backward-Slicing traversiert und anschließend die verbleibenden Knoten und die dazugehörigen PDGs mithilfe des Forward-Slicing.

Mithilfe des Analysewerkzeuges ist es auch möglich Implementierungen der JVM (wie z. B. OpenJDK oder GraalVM) zu analysieren. Hierbei könnten Studien über die Implementierung der JCA/JCE ein Verständnis für die Komplexität aufzeigen und ob mögliche Sicherheitslücken identifiziert werden können.

A Quelltext

A.1 Quelltext aus [Ngu18]

A.1.1 ArithmeticTest

```
1  public class ArithmeticTest {
2    public static void main(String[] argsv){
3      int value = 1;
4      int value2 = 2;
5      value += value;
6      int someOtherValue = 2;
7      value2 += someOtherValue;
8      someOtherValue -= someOtherValue;
9      value = value--;
10     while(someOtherValue < 2){
11       someOtherValue++;
12     }
13     while(value < 2){
14       value*= 2;
15     }
16     while(value2 < 10){
17       value2*= 2;
18     }
19     System.out.println(value);
20     System.out.println(value2);
21   }
22 }
```

Codeauszug 67: Die Klasse `ArithmeticTest`

A.1.2 DeepCallee

```
1  import java.util.Random;
2
3  public class DeepCallee {
4
5    public static Integer important(Integer num){
6      Random randomgenerator = new Random();
```

```
7     Integer result = num + randomgenerator.nextInt(100);
8     return result;
9   }
10
11  public static Integer entry_method(String value) {
12    boolean stay = true;
13    int duration = 5;
14    int result = 0;
15    while (stay){
16      for (int i = 0; i<=duration; duration++){
17        if (i==0){
18          result+=10;
19        }
20        else if (i==5){
21          result+=5;
22        }
23        else {
24          if (i % 2 == 1){
25            result = important(i);
26          } else {
27            result +=2;
28          }
29        }
30      }
31    }
32    return result;
33  }
34
35
36  public static void main(String[] argsv){
37    String value = "Start Test:\n";
38    Integer result = entry_method(value);
39
40    System.out.println("test done");
41    System.out.println(result);
42  }
43 }
```

Codeauszug 68: Die Klasse `DeepCallee`

A.1.3 MultipleCallee

```
1  public class MultipleCalleeTest {
2    public static void main(String[] argsv){
3      String result = "Start\n";
4      ClassA object_a = new ClassA();
5      ClassB object_b = new ClassB();
6      ClassC object_c = new ClassC();
7
8      result += "Call some_method()\n";
9
10     String text_a = object_a.some_method();
11     String text_b = object_b.some_method();
12     String text_c = object_c.some_method();
13
14     System.out.println(result);
15     System.out.println(text_a + text_b + text_c);
16     System.out.println("test done");
17   }
18 }
```

Codeauszug 69: Die Klasse `MultpleCalleeTest`

```
1  public class ClassA {
2    public String some_method() {
3      return "this is Class A\n";
4    }
5  }
```

Codeauszug 70: Die Klasse `ClassA`

```
1  public class ClassB {
2    public String some_method() {
3      return "this is Class B\n";
4    }
5  }
```

Codeauszug 71: Die Klasse `ClassB`

```
1  public class ClassC {
```

```
2   public String some_method() {
3     return "this is Class C\n";
4   }
5 }
```

Codeauszug 72: Die Klasse `ClassC`

A.2 Kryptografie Beispielprogramme

A.2.1 AES

```
1  package de.unibremen.breslice.crypto;
2
3  import java.nio.charset.StandardCharsets;
4  import java.security.MessageDigest;
5  import java.security.NoSuchAlgorithmException;
6  import java.util.Arrays;
7  import java.util.Base64;
8
9  import javax.crypto.Cipher;
10 import javax.crypto.spec.SecretKeySpec;
11
12 public class AES implements ICryptor {
13   private KeyHolder keyHolder;
14
15   public AES(String secretKey) throws NoSuchAlgorithmException {
16     this.keyHolder = new KeyHolder(secretKey);
17   }
18
19   public class KeyHolder {
20     private final SecretKeySpec secretKey;
21
22     protected SecretKeySpec getSecretKey() {
23       return this.secretKey;
24     }
25
26     public KeyHolder(String keyString) throws NoSuchAlgorithmException {
27       MessageDigest sha = null;
28
```

```
29      byte[] keyBytes = keyString.getBytes(StandardCharsets.UTF_8);
30
31      // create sha for key
32      sha = MessageDigest.getInstance("SHA-1");
33      keyBytes = sha.digest(keyBytes);
34
35      // truncate to 16 bytes
36      keyBytes = Arrays.copyOf(keyBytes, 16);
37      secretKey = new SecretKeySpec(keyBytes, "AES");
38    }
39  }
40
41
42
43  @Override
44  public String encrypt(final String strToEncrypt) {
45    try {
46      Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
47      cipher.init(Cipher.ENCRYPT_MODE, this.keyHolder.getSecretKey());
48      return Base64.getEncoder()
49      .encodeToString(cipher.doFinal(strToEncrypt.getBytes(StandardCharsets↩
            .UTF_8)));
50    } catch (Exception e) {
51      System.out.println("Error while encrypting: " + e);
52    }
53
54    return null;
55  }
56
57  @Override
58  public String decrypt(final String strToDecrypt) {
59    try {
60      Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5PADDING");
61      cipher.init(Cipher.DECRYPT_MODE, this.keyHolder.getSecretKey());
62      return new String(cipher.doFinal(Base64.getDecoder()
63      .decode(strToDecrypt)));
64    } catch (Exception e) {
65      System.out.println("Error while decrypting: " + e);
```

```
66      }
67
68      return null;
69    }
70  }
```

Codeauszug 73: Klasse `AES`

A.2.2 Digest

```
1  package de.unibremen.breslice.crypto;
2
3  import java.nio.charset.StandardCharsets;
4  import java.security.MessageDigest;
5  import java.security.NoSuchAlgorithmException;
6  import java.util.Base64;
7  import java.util.HashMap;
8  import java.util.Map;
9  import java.util.Objects;
10
11 public class Digest {
12   private final DigestAlgorithms digestAlgorithmsUsed;
13   private final String digest;
14   private final static DigestAlgorithms DEFAULT_DIGEST_ALGORITHMS = ↩
         DigestAlgorithms.SHA3_512;
15
16   public enum DigestAlgorithms {
17     MD2,
18     MD5,
19     SHA1,
20     SHA224,
21     SHA256,
22     SHA384,
23     SHA512_224,
24     SHA512_256,
25     SHA3_224,
26     SHA3_256,
27     SHA3_384,
```

```
28      SHA3_512
29    }
30
31    private final static Map<DigestAlgorithms, String> algorithmMap = ↩
          createAlgorithmMap();
32
33    private static Map<DigestAlgorithms, String> createAlgorithmMap() {
34      Map<DigestAlgorithms, String> algorithmStringMap = new HashMap<>();
35
36      algorithmStringMap.put(DigestAlgorithms.MD2, "MD2");
37      algorithmStringMap.put(DigestAlgorithms.MD5, "MD5");
38      algorithmStringMap.put(DigestAlgorithms.SHA1, "SHA-1");
39      algorithmStringMap.put(DigestAlgorithms.SHA224, "SHA-224");
40      algorithmStringMap.put(DigestAlgorithms.SHA256, "SHA-256");
41      algorithmStringMap.put(DigestAlgorithms.SHA384, "SHA-384");
42      algorithmStringMap.put(DigestAlgorithms.SHA512_224, "SHA-512/224");
43      algorithmStringMap.put(DigestAlgorithms.SHA512_256, "SHA-512/256");
44      algorithmStringMap.put(DigestAlgorithms.SHA3_224, "SHA3-224");
45      algorithmStringMap.put(DigestAlgorithms.SHA3_256, "SHA3-256");
46      algorithmStringMap.put(DigestAlgorithms.SHA3_384, "SHA3-384");
47      algorithmStringMap.put(DigestAlgorithms.SHA3_512, "SHA3-512");
48
49      return algorithmStringMap;
50    }
51
52    public Digest(String of, DigestAlgorithms digestAlgorithms) throws ↩
          NoSuchAlgorithmException {
53      if(digestAlgorithms == null) digestAlgorithms = ↩
            DEFAULT_DIGEST_ALGORITHMS;
54      this.digestAlgorithmsUsed = digestAlgorithms;
55
56      MessageDigest messageDigest = MessageDigest.getInstance(algorithmMap.↩
            get(digestAlgorithmsUsed));
57
58      this.digest = Base64.getEncoder()
59      .encodeToString(messageDigest.digest(of.getBytes(StandardCharsets.↩
            US_ASCII)));
60    }
```

```
61
62    @Override
63    public boolean equals(Object obj) {
64      if(!(obj instanceof Digest)) return false;
65
66      Digest other = (Digest) obj;
67      return Objects.equals(other.digest, this.digest)
68      && Objects.equals(other.digestAlgorithmsUsed, this.↩
            digestAlgorithmsUsed);
69    }
70
71    @Override
72    public String toString() {
73      return this.getBase64();
74    }
75
76    public String getBase64() {
77      return this.digest;
78    }
79  }
```

Codeauszug 74: Klasse `Digest`

A.2.3 FileHandler

```
1  package de.unibremen.breslice.crypto;
2
3  import java.io.File;
4  import java.io.FileWriter;
5  import java.io.IOException;
6  import java.nio.file.Files;
7
8  public class FileHandler {
9    private final File file;
10   public FileHandler(File file) {
11     this.file = file;
12   }
13   public void saveEncrypted(IEncrypter encrypter, String content) {
```

```
14      try{
15        FileWriter fileWriter = new FileWriter(file);
16        fileWriter.write(encrypter.encrypt(content));
17
18        fileWriter.flush();
19        fileWriter.close();
20      } catch (IOException e) {
21        throw new RuntimeException(e);
22      }
23    }
24    public String loadDecrypted(IDecrypter decrypter) {
25      try {
26        String fileContent = String.join("", Files.readAllLines(file.toPath()↩
              ));
27
28        return decrypter.decrypt(fileContent);
29      } catch (IOException e) {
30        throw new RuntimeException(e);
31      }
32    }
33
34  }
```

Codeauszug 75: Klasse `FileHandler`

A.2.4 ICryptor

```
1  package de.unibremen.breslice.crypto;
2
3  public interface ICryptor extends IDecrypter, IEncrypter {
4  }
```

Codeauszug 76: Interface `ICryptor`

A.2.5 IDecrypter

```
1  package de.unibremen.breslice.crypto;
2
3  public interface IDecrypter {
```

```
4   public String decrypt(String toDecrypt);
5 }
```

Codeauszug 77: Interface `IDecrypter`

A.2.6 IEncrypter

```
1 package de.unibremen.breslice.crypto;
2
3 public interface IEncrypter {
4   public String encrypt(String toEncrypt);
5 }
```

Codeauszug 78: Interface `IEncrypter`

A.2.7 StringSigner

```
1  package de.unibremen.breslice.crypto;
2
3  import java.security.*;
4  import java.util.Base64;
5
6  public class StringSigner {
7    private final KeyPair keyPair;
8    private final String cryptoAlgorithm = "Ed25519";
9
10   public StringSigner() throws NoSuchAlgorithmException {
11     KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance(↩
         cryptoAlgorithm);
12
13     this.keyPair = keyPairGenerator.generateKeyPair();
14   }
15
16   public String sign(String toSign) throws InvalidKeyException, ↩
       NoSuchAlgorithmException, SignatureException {
17     Signature signature = Signature.getInstance(cryptoAlgorithm);
18     signature.initSign(keyPair.getPrivate());
19
20     signature.update(toSign.getBytes());
```

```
21      String signString = Base64.getEncoder().encodeToString(signature.sign↩
            ());
22
23      return signString;
24    }
25
26    public boolean verify(String signString, String signedContent) throws ↩
          InvalidKeyException, NoSuchAlgorithmException, SignatureException {
27      Signature signature = Signature.getInstance(cryptoAlgorithm);
28      signature.initVerify(keyPair.getPublic());
29
30      signature.update(signedContent.getBytes());
31
32      return signature.verify(Base64.getDecoder().decode(signString));
33    }
34  }
```

Codeauszug 79: Klasse `StringSigner`

A.2.8 AESEncryptDecrypt

```
1  package de.unibremen.breslice.crypto.examples;
2
3  import de.unibremen.breslice.crypto.AES;
4
5  import java.security.NoSuchAlgorithmException;
6
7  public class AESEncryptDecrypt {
8    public static void main(String[] args) throws NoSuchAlgorithmException {
9      final String secretKey = "ThisIsATopSecretNeverToBeLostKey";
10
11
12     AES aesA = new AES(secretKey);
13
14     String originalMsg = "ThisIsATopSecretMessageNeverToBeDecrypted.";
15     String encrypted = aesA.encrypt(originalMsg);
16
17     System.out.println(encrypted);
```

```
18
19      String decrypted = aesA.decrypt(encrypted);
20
21      System.out.println(decrypted);
22
23    }
24  }
```

Codeauszug 80: Klasse `AESEncryptDecrypt`

A.2.9 DigestVerify

```
1  package de.unibremen.breslice.crypto.examples;
2
3  import de.unibremen.breslice.crypto.Digest;
4
5  import java.security.NoSuchAlgorithmException;
6
7  public class DigestVerify {
8    public static void main(String[] args) throws NoSuchAlgorithmException {
9      final String msg = "Is this message corrupt?";
10     final String msg2 = "This msg is corrupted!";
11     final String corrupt = "$$$" + msg2;
12
13     Digest msgDigest = new Digest(msg, null);
14     Digest msgDigest2 = new Digest(msg, null);
15     Digest msg2Digest = new Digest(msg2, null);
16     Digest corruptDigest = new Digest(corrupt, null);
17
18     System.out.println("MSG: " + msg);
19     System.out.println(msgDigest);
20     System.out.println(msgDigest2);
21     System.out.println(msgDigest.equals(msgDigest2));
22
23     System.out.println("MSG: "+ msg2);
24     System.out.println(msg2Digest);
25     System.out.println(corruptDigest);
26     System.out.println(corruptDigest.equals(msg2Digest));
```

```
27
28    }
29  }
```

Codeauszug 81: Klasse `DigestVerify`

A.2.10 FileEncryption

```
1  package de.unibremen.breslice.crypto.examples;
2
3  import de.unibremen.breslice.crypto.AES;
4  import de.unibremen.breslice.crypto.FileHandler;
5
6  import java.io.File;
7  import java.security.NoSuchAlgorithmException;
8
9  public class FileEncryption {
10   public static void main(String[] args) throws NoSuchAlgorithmException {
11     File file = new File("secret-rsa_key.txt");
12     FileHandler fileHandler = new FileHandler(file);
13     final String aesKey = "PleaseDontCopy";
14
15     AES aes = new AES(aesKey);
16
17     final String message = "VS-NfD";
18     fileHandler.saveEncrypted(aes, message);
19
20     final String decrypted = fileHandler.loadDecrypted(aes);
21     System.out.println(decrypted);
22   }
23 }
```

Codeauszug 82: Klasse `FileEncryption`

A.2.11 SignatureVerify

```
1  package de.unibremen.breslice.crypto.examples;
2
3  import de.unibremen.breslice.crypto.StringSigner;
```

```
4
5  import java.security.InvalidKeyException;
6  import java.security.NoSuchAlgorithmException;
7  import java.security.SignatureException;
8
9  public class SignatureVerify {
10   public static void main(String[] args) throws NoSuchAlgorithmException, ↩
         SignatureException, InvalidKeyException {
11     StringSigner stringSigner = new StringSigner();
12
13     final String toSign = "a signed message from a friend";
14
15     String base64Signature = stringSigner.sign(toSign);
16
17     System.out.println("Is Signature valid for msg: " + (stringSigner.↩
           verify(base64Signature, toSign) ? "Yes" : "No"));
18   }
19 }
```

Codeauszug 83: Klasse `SignatureVerify`

A.2.12 SignEncryptDecrypt

```
1  package de.unibremen.breslice.crypto.examples;
2
3  import de.unibremen.breslice.crypto.AES;
4  import de.unibremen.breslice.crypto.Digest;
5  import de.unibremen.breslice.crypto.StringSigner;
6
7  import java.security.InvalidKeyException;
8  import java.security.NoSuchAlgorithmException;
9  import java.security.SignatureException;
10
11 public class SignEncryptDigest {
12   public static void main(String[] args) throws NoSuchAlgorithmException, ↩
         SignatureException, InvalidKeyException {
13     final String secretKey = "AgainATopSecretKey";
14     final String originalMsg = "MOTD: I like linux.";
```

```
15
16      StringSigner stringSigner = new StringSigner();
17
18      String base64Signature = stringSigner.sign(originalMsg);
19
20      final String toDigest = originalMsg + "--BEGIN-SIGNATURE--" + ↩
            base64Signature;
21
22      Digest msgDigest = new Digest(toDigest, null);
23
24      System.out.println("Message Digest (Message is with Signature): " + ↩
            msgDigest);
25
26      String toEncrypt = toDigest + "--BEGIN-DIGEST--" + msgDigest;
27      System.out.println("Before encryption: " + toEncrypt);
28
29
30      AES aes = new AES(secretKey);
31
32      String encrypted = aes.encrypt(toEncrypt);
33
34      System.out.println("Encrypted message: " + encrypted);
35
36      AES decrypter = new AES(secretKey);
37
38      String decrypted = decrypter.decrypt(encrypted);
39
40      System.out.println("Decrypted message: " + decrypted);
41
42      String digest = decrypted.split("--BEGIN-DIGEST--")[1];
43      String signature = decrypted.split("--BEGIN-DIGEST--")[0].split("--↩
            BEGIN-SIGNATURE--")[1];
44      String decryptedMsg = decrypted.split("--BEGIN-SIGNATURE")[0];
45
46      System.out.println("Got the following msg: " + decryptedMsg + ", with ↩
            digest: " + digest + ", and signature: "+ signature);
47
```

```
48      Digest newDigest = new Digest(decrypted.split("--BEGIN-DIGEST--")[0], ↩
            null);
49
50      if(newDigest.equals(msgDigest)) {
51        System.out.println("Digest is correct.");
52      }else{
53        throw new RuntimeException();
54      }
55
56      if(stringSigner.verify(signature, decryptedMsg)) {
57        System.out.println("Signature is correct.");
58      }else{
59        throw new RuntimeException();
60      }
61    }
62  }
```

Codeauszug 84: Klasse `SignEncryptDecrypt`

B Exclusionfile

```
java.applet.*
java.awt.*
java.awt.color.*
java.awt.datatransfer.*
java.awt.dnd.*
java.awt.event.*
java.awt.font.*
java.awt.geom.*
java.awt.im.*
java.awt.im.spi.*
java.awt.image.*
java.awt.image.renderable.*
java.awt.print.*
java.beans.*
java.beans.beancontext.*
java.io.*
```

```
java.lang.annotation.*
java.lang.instrument.*
java.lang.invoke.*
java.lang.management.*
java.lang.ref.*
java.lang.reflect.*
java.math.*
java.nio.*
java.nio.channels.*
java.nio.channels.spi.*
java.nio.charset.*
java.nio.charset.spi.*
java.nio.file.*
java.nio.file.attribute.*
java.nio.file.spi.*
java.rmi.*
java.rmi.activation.*
java.rmi.dgc.*
java.rmi.registry.*
java.rmi.server.*
java.security.acl.*
java.security.cert.*
java.security.interfaces.*
java.security.spec.*
java.sql.*
java.text.*
java.text.spi.*
java.util.*
java.util.concurrent.*
java.util.concurrent.atomic.*
java.util.concurrent.locks.*
java.util.jar.*
java.util.logging.*
java.util.prefs.*
java.util.regex.*
java.util.spi.*
java.util.zip.*
javax.accessibility.*
```

```
javax.activation.*
javax.activity.*
javax.annotation.*
javax.annotation.processing.*
javax.crypto.interfaces.*
javax.crypto.spec.*
javax.imageio.*
javax.imageio.event.*
javax.imageio.metadata.*
javax.imageio.plugins.bmp.*
javax.imageio.plugins.jpeg.*
javax.imageio.spi.*
javax.imageio.stream.*
javax.jws.*
javax.jws.soap.*
javax.lang.model.*
javax.lang.model.element.*
javax.lang.model.type.*
javax.lang.model.util.*
javax.management.*
javax.management.loading.*
javax.management.modelmbean.*
javax.management.monitor.*
javax.management.openmbean.*
javax.management.relation.*
javax.management.remote.*
javax.management.remote.rmi.*
javax.management.timer.*
javax.naming.*
javax.naming.directory.*
javax.naming.event.*
javax.naming.ldap.*
javax.naming.spi.*
javax.net.*
javax.print.*
javax.print.attribute.*
javax.print.attribute.standard.*
javax.print.event.*
```

```
javax.rmi.*
javax.rmi.CORBA.*
javax.rmi.ssl.*
javax.script.*
javax.security.auth.*
javax.security.auth.callback.*
javax.security.auth.kerberos.*
javax.security.auth.login.*
javax.security.auth.spi.*
javax.security.auth.x500.*
javax.security.cert.*
javax.security.sasl.*
javax.sound.midi.*
javax.sound.midi.spi.*
javax.sound.sampled.*
javax.sound.sampled.spi.*
javax.sql.*
javax.sql.rowset.*
javax.sql.rowset.serial.*
javax.sql.rowset.spi.*
javax.swing.*
javax.swing.border.*
javax.swing.colorchooser.*
javax.swing.event.*
javax.swing.filechooser.*
javax.swing.plaf.*
javax.swing.plaf.basic.*
javax.swing.plaf.metal.*
javax.swing.plaf.multi.*
javax.swing.plaf.nimbus.*
javax.swing.plaf.synth.*
javax.swing.table.*
javax.swing.text.*
javax.swing.text.html.*
javax.swing.text.html.parser.*
javax.swing.text.rtf.*
javax.swing.tree.*
javax.swing.undo.*
```

```
javax.tools.*
javax.transaction.*
javax.transaction.xa.*
javax.xml.*
javax.xml.bind.*
javax.xml.bind.annotation.*
javax.xml.bind.annotation.adapters.*
javax.xml.bind.attachment.*
javax.xml.bind.helpers.*
javax.xml.bind.util.*
javax.xml.crypto.*
javax.xml.crypto.dom.*
javax.xml.crypto.dsig.*
javax.xml.crypto.dsig.dom.*
javax.xml.crypto.dsig.keyinfo.*
javax.xml.crypto.dsig.spec.*
javax.xml.datatype.*
javax.xml.namespace.*
javax.xml.parsers.*
javax.xml.soap.*
javax.xml.stream.*
javax.xml.stream.events.*
javax.xml.stream.util.*
javax.xml.transform.*
javax.xml.transform.dom.*
javax.xml.transform.sax.*
javax.xml.transform.stax.*
javax.xml.transform.stream.*
javax.xml.validation.*
javax.xml.ws.*
javax.xml.ws.handler.*
javax.xml.ws.handler.soap.*
javax.xml.ws.http.*
javax.xml.ws.soap.*
javax.xml.ws.spi.*
javax.xml.ws.spi.http.*
javax.xml.ws.wsaddressing.*
javax.xml.xpath.*
```

```
org.ietf.jgss.*
org.omg.CORBA.*
org.omg.CORBA_2_3.*
org.omg.CORBA_2_3.portable.*
org.omg.CORBA.DynAnyPackage.*
org.omg.CORBA.ORBPackage.*
org.omg.CORBA.portable.*
org.omg.CORBA.TypeCodePackage.*
org.omg.CosNaming.*
org.omg.CosNaming.NamingContextExtPackage.*
org.omg.CosNaming.NamingContextPackage.*
org.omg.Dynamic.*
org.omg.DynamicAny.*
org.omg.DynamicAny.DynAnyFactoryPackage.*
org.omg.DynamicAny.DynAnyPackage.*
org.omg.IOP.*
org.omg.IOP.CodecFactoryPackage.*
org.omg.IOP.CodecPackage.*
org.omg.Messaging.*
org.omg.PortableInterceptor.*
org.omg.PortableInterceptor.ORBInitInfoPackage.*
org.omg.PortableServer.*
org.omg.PortableServer.CurrentPackage.*
org.omg.PortableServer.POAManagerPackage.*
org.omg.PortableServer.POAPackage.*
org.omg.PortableServer.portable.*
org.omg.PortableServer.ServantLocatorPackage.*
org.omg.SendingContext.*
org.omg.stub.java.rmi.*
org.w3c.dom.*
org.w3c.dom.bootstrap.*
org.w3c.dom.events.*
org.w3c.dom.ls.*
org.xml.sax.*
org.xml.sax.ext.*
org.xml.sax.helpers.*

java.lang.Character
```

```
java.lang.Character.Subset
java.lang.Character.UnicodeBlock
java.lang.Compiler
java.lang.Double
java.lang.Enum
java.lang.Float
java.lang.Integer
java.lang.Long
java.lang.Number
java.lang.Package
java.lang.Process
java.lang.ProcessBuilder
java.lang.ProcessBuilder.Redirect
java.lang.Runtime
java.lang.RuntimePermission
java.lang.SecurityManager
java.lang.Short
java.lang.StackTraceElement
java.lang.StrictMath
java.lang.StringBuffer
java.lang.StringBuilder
java.lang.System
java.lang.Thread
java.lang.ThreadGroup
java.lang.ThreadLocal

java.security.AccessControlContext
java.security.AccessController
java.security.AlgorithmParameterGenerator
java.security.AlgorithmParameterGeneratorSpi
java.security.AlgorithmParameters
java.security.AlgorithmParametersSpi
java.security.AllPermission
java.security.AuthProvider
java.security.BasicPermission
java.security.CodeSigner
java.security.CodeSource
java.security.DigestInputStream
```

```
java.security.DigestOutputStream
java.security.GuardedObject
java.security.Identity
java.security.IdentityScope
java.security.KeyFactory
java.security.KeyFactorySpi
java.security.KeyPair
java.security.KeyPairGenerator
java.security.KeyPairGeneratorSpi
java.security.KeyRep
java.security.KeyStore.Builder
java.security.KeyStore.CallbackHandlerProtection
java.security.KeyStore.PasswordProtection
java.security.KeyStore.PrivateKeyEntry
java.security.KeyStore.SecretKeyEntry
java.security.KeyStore.TrustedCertificateEntry
java.security.KeyStoreSpi
java.security.MessageDigest
java.security.MessageDigestSpi
java.security.Permission
java.security.PermissionCollection
java.security.Permissions
java.security.Policy
java.security.PolicySpi
java.security.ProtectionDomain
java.security.Provider.Service
java.security.SecureClassLoader
java.security.SecureRandom
java.security.SecureRandomSpi
java.security.Security
java.security.SecurityPermission
java.security.SignatureSpi
java.security.SignedObject
java.security.Signer
java.security.Timestamp
java.security.UnresolvedPermission
java.security.URIParameterCryptoPrimitive
java.security.KeyRep.TypeAccessControlException
```

```
java.security.DigestException

java.security.InvalidAlgorithmParameterException
java.security.InvalidParameterException
java.security.KeyException
java.security.KeyManagementException
java.security.PrivilegedActionException
java.security.ProviderException
java.security.UnrecoverableEntryException

javax.net.ssl.CertPathTrustManagerParameters
javax.net.ssl.ExtendedSSLSession
javax.net.ssl.HandshakeCompletedEvent
javax.net.ssl.KeyManagerFactory
javax.net.ssl.KeyManagerFactorySpi
javax.net.ssl.KeyStoreBuilderParameters
javax.net.ssl.SSLContextSpi
javax.net.ssl.SSLEngine
javax.net.ssl.SSLEngineResult
javax.net.ssl.SSLParameters
javax.net.ssl.SSLPermission
javax.net.ssl.SSLServerSocket
javax.net.ssl.SSLServerSocketFactory
javax.net.ssl.SSLSessionBindingEvent
javax.net.ssl.SSLSocket
javax.net.ssl.SSLSocketFactory
javax.net.ssl.TrustManagerFactorySpi
javax.net.ssl.X509ExtendedKeyManager
javax.net.ssl.X509ExtendedTrustManajavax.net.ssl.
javax.net.ssl.SSLEngineResult.HandshakeStatus
javax.net.ssl.SSLEngineResult.Stajavax.net.ssl.
javax.net.ssl.SSLException
javax.net.ssl.SSLHandshakeException
javax.net.ssl.SSLKeyException
javax.net.ssl.SSLProtocolException
javax.net.ssl.SSLPeerUnverifiedException

javax.crypto.*
```

```
java.net.*
java.net.Authenticator
java.net.CacheRequest
java.net.CacheResponse
java.net.ContentHandler
java.net.CookieHandler
java.net.CookieManager
java.net.DatagramPacket
java.net.DatagramSocket
java.net.DatagramSocketImpl
java.net.HttpCookie
java.net.IDN
java.net.Inet4Address
java.net.Inet6Address
java.net.InetAddress
java.net.InetSocketAddress
java.net.InterfaceAddress
java.net.JarURLConnection
java.net.MulticastSocket
java.net.NetPermission
java.net.NetworkInterface
java.net.PasswordAuthentication
java.net.Proxy
java.net.ProxySelector
java.net.ResponseCache
java.net.SecureCacheResponse
java.net.ServerSocket
java.net.Socket
java.net.SocketAddress
java.net.SocketImpl
java.net.SocketPermission
java.net.StandardSocketOptions
java.net.URI
java.net.URLClassLoader
java.net.URLDecoder
java.net.URLEncoder
java.net.URLStreamHandjava.net.
```

```
java.net.Authenticator.RequestorType
java.net.Proxy.Type
java.net.StandardProtocolFamjava.net.
java\/awt\/.*
javax\/swing\/.*
sun\/awt\/.*
sun\/swing\/.*
com\/sun\/.*
sun\/.*
```

Literatur

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[And94] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, 1994.

[BS18] Carsten Bormann and Karsten Sohr. isec: Informationssicherheit - Vorlesungsfolien, 2018.

[BSI] BSI. Auftrag des BSI. `https://www.bsi.bund.de/DE/Das-BSI/Auftrag/auftrag_node.html`.

[Cav08] Salvador Cavadini. Secure slices of insecure programs. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, ASIACCS '08, pages 112–122, New York, NY, USA, March 2008. Association for Computing Machinery.

[CJB10] Christof Paar, Jan Pelzl, and Bart Preneel. *Understanding Cryptography: A Textbook for Students and Practitioners*. Springer, Berlin, 2nd edition, 2010.

[Cona] Contributors of WALA. *WALA Analysis-Scope*. Abgerufen: 2022.

[Conb] Contributors of WALA. *WALA CallGraph*.

[Conc] Contributors of WALA. *WALA Class Hierarchy Basics*. Abgerufen: 2022.

[Cond] Contributors of WALA. *WALA Demand-Driven Pointer Analysis*.

[Cone] Contributors of WALA. *WALA Intermediate Representation*. Abgerufen: 2022.

[Conf] Contributors of WALA. *WALA Pointer Analysis*. Abgerufen: 2022.

[Cong] Contributors of WALA. *WALA Slicer*. Abgerufen: 2022.

[CW07] Brian Chess and Jacob West. Secure Programming with Static Analysis, 2007.

[Cyl19] Michael Cyl. Sicherheitsanalyse für Android-Systemdienste auf der Basis von Programm-Slicing. Master's thesis, Universität Bremen, 2019.

[DE82] Robling Denning and Dorothy Elizabeth. *Cryptography and data security*. Addison-Wesley Longman Publishing Co., Inc., USA, 1982.

[Det16] Mathias Detmers. Evaluation des WALA-Slicers bzgl. der Anwendbarkeit auf sicherheitskritische Java-Programme, 2016.

[DWWS18] Wolfgang Däubler, Peter Wedde, Thilo Weichert, and Imke Sommer. *EU-Datenschutz-Grundverordnung und BDSG-neu*. Kompaktkommentar. Bund-Verlag, Frankfurt am Main, 2018.

[EBFK13] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 73–84, New York, NY, USA, November 2013. Association for Computing Machinery.

[Eck18] Claudia Eckert. *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. De Gruyter, Inc., September 2018.

[FCKV10] Viktoria Felmetsger, Ludovico Cavedon, Christopher Kruegel, and Giovanni Vigna. Toward automated detection of logic vulnerabilities in web applications. In *Proceedings of the 19th USENIX conference on Security*, USENIX Security'10, page 10, USA, August 2010. USENIX Association.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[Ger15] Patrick Gerken. Statische Sicherheitsanalyse von Java Enterprise-Anwendungen mittels Program Slicing, 2015.

[Gra16] Jürgen Graf. *Information Flow Control with System Dependence Graphs*. PhD thesis, Karlsruher Institut für Technologie (KIT), 2016.

[Gul14] Markus Gulmann. Statische Sicherheitsanalyse der Android Systemservices, 2014.

[HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *SIGPLAN Not.*, 23(7):35–46, jun 1988.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[HS09] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8:399–422, October 2009.

[IDC21] IDC. Marktanteile der Betriebssysteme am Absatz vom Smartphones weltweit in den Jahren 2010 bis 2020 und Prognose bis 2025. `https://de.statista.com/statistik/daten/studie/182363/umfrage/prognostizierte-marktanteile-bei-smartphone-betriebssystemen/`, 2021.

[Ker19] Alim Kerimov. Evaluation eines auf Slicing basierenden Codeanalyse-Werkzeugs in Bezug auf seine praktische Anwendbarkeit im Kontext der IT-Sicherheit, May 2019.

[KFS+13] Kuzman Katkalov, Peter Fischer, Kurt Stenzel, Nina Moebius, and Wolfgang Reif. Evaluation of Jif and Joana as Information Flow Analyzers in a Model-Driven Approach. volume 7731, pages 174–186. January 2013.

[Kri04] Jens Krinke. *Advanced Slicing of Sequential and Concurrent Programs*. PhD thesis, Universität Passau, 2004.

[KSBR13] Kuzman Katkalov, Kurt Stenzel, Marian Borek, and Wolfgang Reif. Model-Driven Development of Information Flow-Secure Systems with IFlow. In *2013 International Conference on Social Computing*, pages 51–56, September 2013.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, October 1973.

[Mar17] Keith M. Martin. *Everyday cryptography: fundamental principles and applications*. Oxford University Press, second edition edition, 2017.

[Mei21] Marcel Meissner. Evaluation und Weiterentwicklung eines Slicers zur Sicherheitsanalyse von Java Anwendungen. Master's thesis, Universität Bremen, 2021.

[MMK06] Durga Mohapatra, Rajib Mall, and Rajeev Kumar. An Overview of Slicing Techniques for Object-Oriented Programs. *Informatica (Slovenia)*, 30:253–277, June 2006.

[MS08] Benjamin Monate and Julien Signoles. Slicing for Security of Code. pages 133–142, March 2008.

[Mö20] Jan Möhlmann. Konzeption und Weiterentwicklung eines Java Slicing-Tools für Sicherheitsanalysen. Master's thesis, Universität Bremen, 2020.

[Ngu18] Philip Phu Dang Hoan Nguyen. Statische Sicherheitsanalyse mit automatisierten Code Audits. Master's thesis, Universität Bremen, 2018.

[NHR05] Flemming Nielson, Chris Hankin, and Hanne Riis Nielson. *Principles of program analysis*. Springer, Berlin, corr. 2. print. edition, 2005. XXI, 452 S. : graph. Darst.

[NNH99] Flemming Nielson, Hanne Nielson, and Chris Hankin. *Principles of Program Analysis*. January 1999.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. *SIGSOFT Softw. Eng. Notes*, 9(3):177–184, apr 1984.

[SB06] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. *ACM SIGPLAN Notices*, 41(6):387–400, June 2006.

[SFB07] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. Thin slicing. *SIGPLAN Not.*, 42(6):112–122, jun 2007.

[SKBR14] Kurt Stenzel, Kuzman Katkalov, Marian Borek, and Wolfgang Reif. A model-driven approach to noninterference. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications*, 5:30–43, September 2014.

[Sri07] Manu Sridharan. *Refinement-Based Program Analysis Tools*. PhD thesis, EECS Department, University of California, Berkeley, Oct 2007.

[SS14] Shailendra Narayan Singh and Leena Singh. Study of current program slicing techniques. In *2014 5th International Conference - Confluence The Next Generation Information Technology Summit (Confluence)*, pages 810–814, September 2014.

[SVKW07] Stefan Staiger, Gunther Vogel, Steffen Keul, and Eduard Wiebe. Interprocedural Static Single Assignment Form. In *Proceedings of the 14th Working Conference on Reverse Engineering*, pages 1–10, 2007.

[TIO] TIOBE Software BV. TIOBE Index. `https://www.tiobe.com/tiobe-index/`.

[Tra] Transforma Insights. Number of Internet of Things (IoT) connected devices worldwide from 2019 to 2030. `https://www.statista.com/statistics/1194682/iot-connected-devices-vertically/`.

[TSB15] Julian Thomé, Lwin Khin Shar, and Lionel Briand. Security slicing for auditing XML, XPath, and SQL injection vulnerabilities. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 553–564, November 2015.

[TSBB18] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. Security slicing for auditing common injection vulnerabilities. *Journal of Systems and Software*, 137:766–783, March 2018.

[Uni] United Nations: Population Devision. World Population Prospects 2022 - Demographic Indicators - Total Population. `https://population.un.org/wpp/Download/Standard/MostUsed/`.

[W3T22] W3Techs. Usage statistics of operating systems for websites. `https://w3techs.com/technologies/overview/operating_system`, 2022.

[Wei84] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[WLG11] Yi Wang, Zhoujun Li, and Tao Guo. Program Slicing Stored XSS Bugs in Web Application. In *2011 Fifth International Conference on Theoretical Aspects of Software Engineering*, pages 191–194, August 2011.

[Zha] Chen Zhaojun. CVE-2021-44228. `https://www.cve.org/CVERecord?id=CVE-2021-44228`.