Statische Sicherheitsanalyse mit automatisierten Code Audits

Sicherheitsanalyse von Java-Applikationen mit erweitertem Programm (WALA-)Slicing

Masterthesis

im Studiengang Informatik
(Fachbereich Mathematik und Informatik)

vorgelegt von:	Philip Phu Dang Hoan Nguyen
Matrikelnummer:	22 83 166
Erstgutachter:	Dr. Karsten Sohr
Zweitgutachter:	Prof. Dr. Martin Gogolla

© 2018

Dieses Werk einschließlich seiner Teile ist **urheberrechtlich geschützt**. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtgesetzes ist ohne Zustimmung des Autors unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen sowie die Einspeicherung und Verarbeitung in elektronischen Systemen.

Eidesstattliche Erklärung

Ich, Philip Phu Dang Hoan Nguyen, Matrikel-Nr. 22 83 166, versichere hiermit, dass ich meine Masterthesis mit dem Thema

> *Statische Sicherheitsanalyse mit automatisierten Code Audits - Sicherheitsanalyse von Java-Applikationen mit erweitertem Programm (WALA-)Slicing*

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Mir ist bekannt, dass ich meine Masterarbeit zusammen mit dieser Erklärung fristgemäß nach Vergabe des Themas in dreifacher Ausfertigung und gebunden im Prüfungsamt der Universität Bremen abzugeben oder spätestens mit dem Poststempel des Tages, an dem die Frist abläuft, zu senden habe.

Bremen, den 28. März 2018

Philip Phu Dang Hoan Nguyen

© Philip Phu Dang Hoan Nguyen

Inhaltsverzeichnis

 © Philip Phu Dang Hoan Nguyen

 © Philip Phu Dang Hoan Nguyen

1. Einleitung

Informationssicherheit spielt in unserem jetzigen Zeitalter nach wie vor eine große Rolle und rückt durch die Medien durchaus okkasionell ins Rampenlicht, sei es wegen interkontinentaler Datenspionage, Datenschutz-Verletzungen oder Software-Sicherheitslücken. Letztere erregen die Aufmerksamkeit der breiten Masse, da diese Probleme zwar IT-Unternehmen verantworten, letztendlich aber jeden persönlich betreffen, wie bei einem Datenleck, bei dem personenbezogene Daten von Dritten unrechtmäßig erlangt werden (beispielsweise beim Sony Datenleck im Jahre 2011 [datenleck.net 2011]).

Eine solide Basis der Informationssicherheit bildet die Kryptografie, welche nicht nur genutzt wird, um Nachrichten geheim zu halten, sondern Grundlage ist für Sicherheitsziele wie die Vertraulichkeit, Integrität und Authentizität. Dass mit Hilfe der richtigen Kryptosysteme in der Theorie eine hohe Sicherheit garantiert werden kann, zeigt die Entwicklung dieser in den letzten Jahren. So wurden seit den 70ern diverse Verschlüsselungsverfahren entwickelt, jahrelang auf Lücken und Belastbarkeit geprüft, optimiert und schließlich standardisiert [Paar und Pelzl 2009]. Unter anderem resultierte daraus eines der bekanntesten Verschlüsselungsverfahren AES, welches für zahlreiche Standards aus unserem Alltag angewendet wird, wie beispielsweise IP-Telefonie, Festplattenverschlüsselung, WiFi (IEEE 802.11i) oder TLS, welches für verschlüsselte, nicht abhörbare Übertragung von Internetseiten benötigt wird (HTTPS).

AES wird seit über 15 Jahren von Sicherheitsanalytikern auf Sicherheitslücken überprüft; bisher ohne existierenden praktikablen Angriff außer reines Bruteforce. Dieser Bruteforce-Angriff, also das unsystematische Durchprobieren eines beispielsweise 128-Bit langen AES Schlüssels (kürzeste mögliche Schlüssellänge), würde jedoch laut Abschätzungen ca. 1,02 Trillionen ($1,02 * 10^{18}$) Jahre dauern [EETimes 2012]. Dass hierbei selbst das Parallelisieren dieser Aufgabe mit Hilfe von mehreren Computern keine großartige Hilfe ist, zeigt eine Beispielrechnung des Datenträger-Herstellers Seagate: Dabei wird hypothetisch davon ausgegangen, wenn jede Person auf der Erde (7 Milliarden) zehn Computer besäße und jeder dieser Rechner 1 Milliarde Schlüsselkombinationen pro Sekunde prüfen kann (äußerst optimistisch), man dennoch über 77.000.000.000 Jahre bräuchte [SEAGATE 2008]. Da abschließend davon ausgegangen werden kann, dass die Funktionsweise von AES keine Sicherheitslücken

hat und der Bau von Quantencomputern ebenso nicht möglich ist, um die der Kryptografie zugrunde liegenden mathematischen Probleme schneller lösen zu können, gilt AES nicht nur als sicher, sondern auch als zukunftssicher.

Zwar bieten heutige Verschlüsselungs-Standards in der Theorie zwar einen nahezu perfekten Schutz, jedoch setzt dieses in der Praxis auch eine korrekte Verwendung voraus. Dass dies oft nicht der Fall ist, zeigt ein Sicherheitsbericht der Software-Sicherheitsfirma Veracode aus dem Jahre 2016 [Veracode 2016]. Diese bezieht Daten aus über 200.000 Applikationen, welche in einem Zeitraum von 18 Monaten analysiert und in diesem Bericht zusammengefasst wurden. Herauszustellen ist hierbei, dass Kryptografie-Probleme wesentlich öfter Probleme bereiten als die üblichen Software-Schwachstellen wie Cross-Site-Scripting oder CRLF- und SQL-Injection. Insgesamt 39 % aller Applikationen zeigten laut Bericht sicherheitskritische Implementierung von Kryptosystemen, wobei dazu u.a. die Verwendung unsicherer oder geknackter Verschlüsselungsverfahren, unsaubere Validierung von TLS-Zertifikaten, hardcodierte Schlüssel im Klartext oder zu schwache Verschlüsselung (etwa durch falsche Parameter oder zu kurze Schlüssellänge) zählen. Entgegen der Erwartungen schneiden bei den Sicherheitstests kommerziell-entwickelte sogar generell schlechter ab als intern selbst-entwickelte Programme, was vergangene Implementierungsfehler großer Firmen in der Vergangenheit ebenfalls unterstreicht. So wurde beispielsweise 2005 bei Microsoft in der Verschlüsselungsfunktion einiger Office Produkte entdeckt, dass die RC4-Initialisierungsvektoren mehrfach genutzt wurden, was schließlich zur Sicherheitslücke führte [Wu 2005].

Gründe für diese mangelnde Umsetzung in der Praxis sehen Experten u.a. im Wandel der Software-Entwicklung, in der Verschlüsselung durch das steigende Sicherheitsbewusstsein der Menschen immer mehr als selbstverständlich angesehen und mehr gefordert wird. Während in der Vergangenheit Softwareentwickler und Kryptografie-Experten getrennte Tätigkeitsfelder waren, gehört heute die Verwendung von Kryptografie gezwungenermaßen zum Repertoire der Entwickler dazu, welche in der Regel zu wenig Expertise hinsichtlich Kryptografie haben [Van Houtven 2013; Crypteron 2016]. Neben dem mangelnden tiefen Verständnis der Kryptosysteme, welche zu den oben genannten typischen Programmierfehlern führen, sind zudem auch viele Programmierschnittstellen (API) unnötig komplex, schlecht dokumentiert, nicht intuitiv, welches letztendlich oft zu einer falschen Benutzung (z.B. unsinnige Parametrisierung) dieser Entwickler führt [Contini 2017]. So thematisieren und beschrieben auch Green und Smith diese Problematik in ihrem Artikel: "Developers are Not the Enemy - The Need for Usable Security APIs"[Green und Smith 2016], welcher 2016 in der Zeitschrift „IEEE Security & Privacy“ erschien. Dieser beunruhigende Umstand gibt daher genügend Anlass, sich näher mit Möglichkeiten auseinander zu setzen, Sicherheits-Audits für Programme zu vereinfachen und letztendlich Fehler

 © Philip Phu Dang Hoan Nguyen

dieser Art zu finden und zu beheben. Hauptaugenmerk soll in dieser Arbeit auf Open-Source-Programmen in Java gerichtet werden. Die Wahl fällt hierbei auf Java, da es laut TIOBE Index seit Jahren die populärste Programmiersprache ist [TIOBE 2017] und somit in vielen Open-Source-Repositories vertreten ist; die Arbeit behandelt explizit Open-Source-Anwendungen, da diese auf Grund des offenen Quellcodes anfälliger für Angriffe sind und bessere Quellen für Analysen mit White-Box-Test-Verfahren darstellen, wie die in der Arbeit genutzte statische Code-Analyse.

Ziel dieser empirischen Arbeit ist es, ein Auditor-Werkzeug zu entwickeln, welches mit Hilfe von Software-Engineering Techniken wie Programm-Slicing aus großen Applikationen die sicherheitsrelevanten Programmteile extrahieren kann, was Experten für weiterführende Analyse nutzen könnten. Programm-Slicing, oder Slicing wurde 1981 erstmals von Mark Weiser eingeführt und wurde derzeit als Hilfsmittel genutzt Programme besser zu verstehen und zu debuggen. Er fasst es wie folgt zusammen:

> „Program slicing is a method for automatically decomposing programs by analyzing their data flow and control flow. Starting from a subset of a program's behavior, slicing reduces that program to a minimal form which still produces that behavior. The reduced program, called a „slice“ is an independent program guaranteed to represent faithfully the original program within the domain of the specified subset of behavior.“[Weiser 1981]

Somit ergibt sich die Möglichkeit, ein Programm über Daten- und Kontrollflussabhängigkeiten automatisch über definierte Kriterien zu kürzen, ohne das ursprüngliche Verhalten zu verlieren. Mit Hilfe der Slicing-Kriterien können für das Sicherheits-Audit ausgewählte sicherheitsrelevante Methodenaufrufe, wie beispielsweise *Signature.sign()* oder *Cipher.doFinal()* ausgewählt und untersucht werden. Über sogenanntes Backward-Slicing werden diese Slicing-Kriterien als Startpunkte gesetzt und alle Anweisungen anschließend automatisch berechnet, welche diese (transitiv) beeinflussen.

Dieser Arbeit geht ein technischer Bericht über Sicherheits-Audits mit Hilfe von Design by Contract (DBC) und Slicing von Sohr, Mustafa, Hirch und Gulmann voraus [Sohr u. a. 2015b], welche die Idee dieser Arbeit in Grundsätzen darlegen, jedoch neben Slicing Konzepte wie DBC und Extended Static Checking (ESC) aufgreifen.

Ferner baut die Arbeit ebenfalls auf den Erkenntnissen der Evaluation des WALA-Slicers bzgl. der Anwendbarkeit auf sicherheitskritische Java-Programme von Detmers [Detmers 2016] auf, welche zum einen die Nutzbarkeit des Slicers *WALA* von IBM auf Java-Programme exemplarisch darstellt und zum anderen sinnvolle Erkenntnisse wie die optimale Parameterwahl liefert. *WALA* erwies sich als das bevor-

zugte Werkzeug für die Implementierung des Auditors. Aufbauend auf diesen und anderen Erkenntnissen gilt es, das Toolverhalten tiefergehender zu evaluieren, die Anwendung von *WALA* zu optimieren und untersuchte Erkenntnisse zu dokumentieren.

WALA wies beispielsweise bei bestimmter Parametrisierung nicht komplett korrekte Slices auf, welches im Verlauf der Evaluation auf die gewählte Kontrollabhängigkeitsoption zurückzuführen war. Außerdem gab es diesbezüglich weitere Seiteneffekte, welche mit Hilfe von Optimierungen gelöst werden konnten. Generell gilt es bei der Parameterwahl und den Optimierungen, einen passenden Trade-off zwischen einem detaillierten, jedoch zu großen und teuren und einem groben, aber kleinen und schnellen Slice zu finden.

Aus der vorhergehenden Zielausrichtung ergibt sich folgende Gliederung der vorliegenden Arbeit in acht Teile:

Zunächst werden in Kapitel 2 die Grundlagen der Kryptografie erläutert, um ein grundsätzliches Verständnis der verschiedenen Verfahren, wie beispielsweise Verschlüsselung, Signaturen und Hashes, zu vermitteln. Anschließend wird in Kapitel 3 eine kleine Einführung in die Programmanalyse gegeben. Die Erläuterungen der statischen Analyse bilden hierbei die Grundlage für das später genutzte Programm-Slicing, das ebenfalls grob beschrieben wird. In Kapitel 4 wird zunächst die Methodik der Arbeit erläutert, in der nicht nur auf die empirische Vorgehensweise und die bestehenden Forschungsergebnisse, sondern auch auf das Vorgehensmodell zur Software-Entwicklung eingegangen werden soll. Das Kapitel 5 leitet den praktischen Teil der Arbeit ein und beschreibt zunächst die definierten Anforderungen, bevor auf die Analysetechnik des Auditors eingegangen wird. Neben der technischen Beschreibung des Slicing-Werkzeugs *WALA* und der groben Darstellung der API, wird ebenso auf die allgemeine Funktionsweise und die Architektur des Auditors Bezug genommen. Im darauf folgenden Abschnitt wird die Implementierung inklusive einer Auflistung der benutzten Bibliotheken, sowie das Build-Verfahren beschrieben. Letztlich wird eine detaillierte Erläuterung zu den resultierenden Optimierungen und Erweiterungen des „Auditors“ gegeben. Mit Kapitel 6 wird auf die Evaluierung des Auditors eingegangen, welche sich in zwei Teile untergliedert. Der erste Abschnitt beschäftigt sich mit kleineren Testfällen, welche bestehende Probleme erläutern und die dazu implementierte Erweiterung verifizieren soll. Es folgt der zweite Teil der Evaluierung, in welcher der Auditor auf zwei größere Anwendungen mit Sicherheitsmechanismen angewendet und abschließend bewertet wird. Das letzte Kapitel schließt die Arbeit mit einer Zusammenfassung ab, welche Gesamtaussagen über die Effizienz und Nutzbarkeit des Auditors geben soll; sie soll ebenfalls einen Ausblick geben, welche Forschungsarbeiten zu dieser Thematik folgen könnten.

2. Grundlagen der Kryptografie

Da die vorliegende Arbeit Sicherheitsanalysen von Java-Applikationen behandelt, ist es grundlegend ein Grundverständnis über Technologien zu haben, welche genutzt werden, um letztendlich „Sicherheit“ zu gewährleisten. Folgender Abschnitt soll einen groben Einstieg in die Kryptografie und deren Verfahren schaffen.

Informationssicherheit beschreibt zunächst unterschiedliche Eigenschaften, die ein System bzw. Programm aufweisen soll, als sogenannte *Sicherheitsziele*. Typische *Sicherheitsziele* [Bormann u. a. 2010] sind:

„Datenschutz“ Der Begriff Datenschutz beschreibt das Recht auf Schutz eigener (persönlicher) Informationen. Das bedeutet, dass der Benutzer eines Systems die informationelle Selbstbestimmung über seine Daten hat und Applikationen diese bei der Verarbeitung zunächst vertraulich vor unbefugtem Zugang und missbräuchlicher Weiterverarbeitung schützen muss, bis der Benutzer dieses berechtigt. Dies schützt die Privatsphäre des Menschen und erlaubt es ihm, selbst zu entscheiden, wieviel er über sich preisgeben möchte.

„Integrität“ Integrität bedeutet Schutz vor unautorisierter und unbemerkter Veränderung von Daten. Als Beispiel können, bezogen auf Bankanwendungen, Bankkonten und Geld-Transaktionen genommen werden. So sollten Kontostände nicht unautorisiert verändert werden können, sondern ausschließlich kontrolliert über Transaktionen abgehandelt werden, welche die Änderungen des Kontostandes zudem noch nachweisen.

„Authentizität“ Das Sicherheitsziel Authentizität beinhaltet im Allgemeinen, dass Informationen integer und frisch sind und eindeutig einer Identität zuzuordnen sind. Vor allem der Datenursprung muss mittels Authentizität nachgewiesen werden können.

„Vertraulichkeit“ Vertraulichkeit soll verhindern, dass Daten abgehört werden. So sollen Daten, die über eine Kommunikationverbindung gehen, nicht für Dritte lesbar sein.

„Verbindlichkeit/Nichtabstreitbarkeit“ Nichtabstreitbarkeit erfordert, dass durchgeführte Handlungen nicht abstreitbar sind. Erreichbar ist dieses Ziel bspw. mit digitalen Signaturen.

„Zurechenbarkeit“ Zurechenbarkeit stellt sicher, dass durchgeführte Handlungen einem Kommunikationspartner eindeutig zugeordnet werden. Dies ist bspw. bei Bankanwendungen bedeutsam, da jedes Konto einem Inhaber gehört und Transaktionen eindeutig zugeordnet werden müssen.

Die grundlegendste Technologie, um Informationssicherheit zu gewährleisten, bildet die Kryptografie, welche unter anderem mit Hilfe von Chiffren und Algorithmen, Nachrichten verschlüsselt und sicher transportierbar macht. Kryptografie lässt sich in drei verschiedene Gebiete aufteilen: Symmetrische Algorithmen, asymmetrische Algorithmen und kryptografische Protokolle, welche in diesem Kapitel einführend vorgestellt werden. Nach der Einführung in die grundlegenden Konzepte der Verschlüsselung beschreibt das vorliegende Kapitel ferner grob kryptografische Verfahren, die in der heutigen Zeit verwendet werden, um Datenintegrität zu gewährleisten.

Da eine detaillierte Erläuterung aller Kryptosysteme den Rahmen der Arbeit überschreiten würden, werden lediglich die groben Konzepte eingeführt. Erläuterungen der Einzelschritte oder mathematischen Hintergründe sind aus weiterführender Literatur zu entnehmen (bspw. [Paar und Pelzl 2009] oder [Martin 2012]).

Im Folgenden soll als erstes im Abschnitt 2.1 auf die symmetrische Kryptografie eingegangen werden, welche sich in Strom- und Block-Chiffren unterteilt. Im Abschnitt 2.2 wird die asymmetrische Kryptografie (auch Public-Key Kryptografie genannt) mit einer kurzen Erläuterung zu Einweg-Funktionen beschrieben. Darauf wird grob auf die Anwendung von Verfahren wie Hashes, MACs, digitale Signaturen und Zertifikate eingegangen, welche alle unterschiedlichen Ebenen von Datenintegrität abdecken.

2.1. Symmetrische Kryptografie

Um die vorhergehenden Sicherheitsziele zu erreichen, benötigt es grundsätzlich Systeme, die Nachrichten verschlüsseln und entschlüsseln, damit Dritte diese nicht abgreifen und lesen können. Die Problematik wird in Abbildung 2.1 verdeutlicht, in der eine dritte Person `Oscar` die Kommunikation zwischen `Alice` und `Bob` klar abhören kann, da der Kanal zwischen ihnen keinerlei Kryptosysteme einsetzt. Für Erklärungen auf dem Gebiet der Kryptografie und Transportprotokollen werden die Synonyme `Alice` und `Bob` als Sender und Empfänger einer Nachricht verwendet.

Symmetrische Verschlüsselung beschreibt eine einfache Form von Kryptografie, wie sie von der Antike bis zu den ersten offiziellen Standards genutzt wurde: Zwei Parteien teilen sich einen geheimen Schlüssel, welcher genutzt wird, um eine Nachricht auf

 © Philip Phu Dang Hoan Nguyen

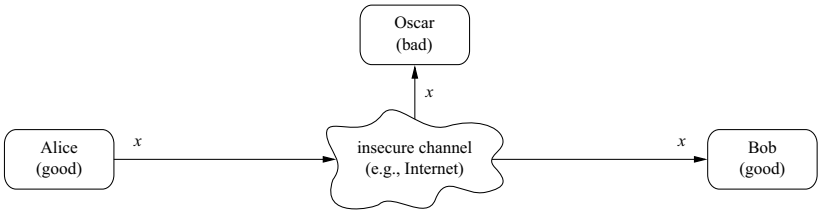

Abbildung 2.1.: Kommunikation über einen ungesicherten Kanal (Quelle: [Paar und Pelzl 2009])

der einen Seite zu verschlüsseln und auf der anderen Seite wieder zu entschlüsseln [Paar und Pelzl 2009].

Die Abbildung 2.2 verdeutlicht ein symmetrisches Kryptosystem mit seinen in der Kryptografie wiederkehrenden Komponenten:

- `x` als Klartext (engl. plaintext),
- `y` als Geheimtext (engl. ciphertext),
- `k` als Schlüssel (engl. key)

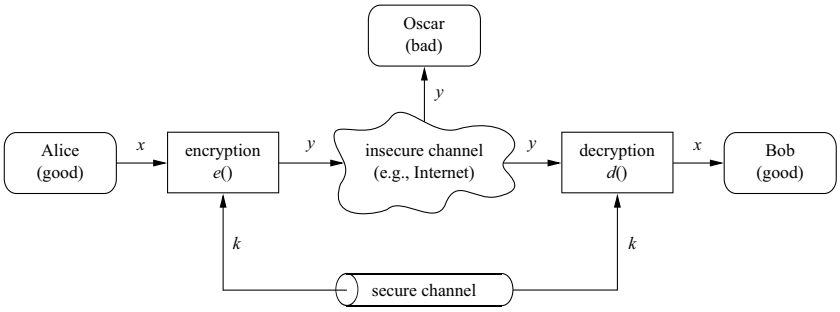

Abbildung 2.2.: Symmetrisches Kryptosystem (Quelle: [Paar und Pelzl 2009])

Symmetrische Verschlüsselung lässt sich in *Strom-Chiffren* und *Block-Chiffren* aufteilen, welche sich in der Art der Abarbeitung der Nachrichten unterscheiden. Abbildung 2.3 zeigt bildlich die Unterschiede zwischen den beiden Typen, in welchen b Bits verschlüsselt werden. Da digitale Daten aus binären Daten bestehen, macht es Sinn, dass Verschlüsselungen auf dieser Ebene durchgeführt werden.

2.1.1. Strom-Chiffren

Strom-Chiffren (engl. stream ciphers) verschlüsseln jedes Bit des Klartextes einzeln, indem es mit einem Bit des Schlüsselstroms verrechnet wird (Vgl. Abbildung 2.3a).

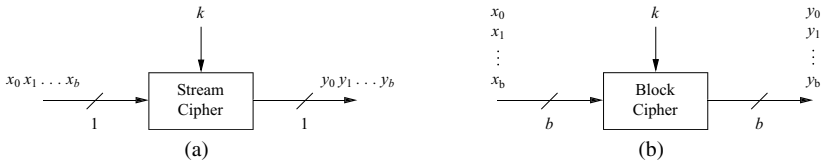

Abbildung 2.3.: Prinzipien der Verschlüsselungen mit Strom-Chiffre (a) und Block-Chiffre (b) (Quelle: [Paar und Pelzl 2009])

Als logische Operation wird in vielen Fällen die Operation XOR verwendet, da diese gleichzeitig zum Verschlüsseln und zum Entschlüsseln genutzt werden kann (das zweifache Anwenden von XOR auf einen Klartext mit einem Schlüssel ergibt erneut den Klartext).

Entscheidend für die Sicherheit und Effektivität der Strom-Chiffren ist die Generierung des Schlüsselstroms. Für sog. *bedingungslose Sicherheit*, demzufolge perfektsichere Verschlüsselung, könnten One-Time Pads (OTP) genutzt werden, welche mit Hilfe von einem True Random Number Generator (TRNG) Schlüssel generieren können, die die selbe Länge haben wie die Nachricht. Praktikabel sind allerdings eher Schlüsselstrom-Generatoren, welche einen initialen (kurzen) Schlüssel als Input (Seed) nehmen und somit als Pseudorandom Number Generator (PRNG) fungieren.

Ferner gibt es neben den vorgestellten synchronen Strom-Chiffren, bei denen der Schlüsselstrom nur abhängig vom Schlüssel ist, auch asynchrone, bei welchen der resultierende Geheimtext zusätzlich als zweiter Input für den Schlüsselstrom Generator genutzt wird und somit den Schlüsselstrom beeinflusst.

2.1.2. Block-Chiffren

Block-Chiffren (engl. block ciphers) verarbeiten im Gegensatz zu Strom-Chiffren jeweils einen kompletten Block des Klartextes (vgl. Abbildung 2.3b). In anderen Worten bedeutet das, dass ein Block mit einer bestimmten *Blocklänge* (*block-size*) (und somit einer festgelegten Anzahl an Bits) mit dem gleichen Schlüssel verarbeitet wird und somit ein Geheimtext-Block derselben Größe ausgegeben wird. Zu den bekanntesten Block-Chiffren gehören unter anderem die beiden Standards Data Encryption Standard (DES) und Advanced Encryption Standard (AES). DES entstand aus einer Ausschreibung im Jahre 1974 von dem US National Bureau of Standards (NBS) und wurde 1977 modifiziert publiziert und weitverbreitet als Standard genutzt, bis Schwächen in der Schlüssellänge entdeckt wurden (mit Hilfe von spezieller Hardware war es möglich alle möglichen Schlüssel in angemessener Zeit

© Philip Phu Dang Hoan Nguyen

mit Hilfe von *Bruteforce* durchzutesten). DES verschlüsselt Blöcke der Länge 64 Bits mit einem Schlüssel, der 56 Bits umfasst und basiert auf einem sogenannten *Feistel-Netzwerk* oder *Chiffre*. Während die ursprüngliche Form als „geknackt“ oder „broken“ gilt (wegen seiner nun zu kleinen Schlüssellänge), wird der Algorithmus neben dem Nachfolger AES noch in seiner weiterentwickelten Form, bekannt als *Triple DES*, in vielen Anwendungen genutzt.

Der Nachfolger AES entstand ebenfalls aus einer Ausschreibung im Jahre 2001, in dem der *Rijndael Algorithmus* als Sieger hervorging und später modifiziert als AES publiziert wurde. Im Unterschied zu DES sollte das Chiffre die Blockgröße 128 statt 64 Bits nutzen, um zukunftssicher zu bleiben, unterschiedliche Schlüssellängen unterstützen (128, 192 und 256 Bits) und zusätzlich komplett öffentlich zugänglich sein, um breites Vertrauen zu gewinnen und zum anderen die komplette Expertise aller Analysten in die Entwicklung mit einzubringen. AES gilt ebenfalls nicht nur als zukunftssicher, sondern ist, anders als DES, auch noch effizient in Software zu implementieren.

Der Algorithmus basiert nicht auf einem *Feistel Netzwerk*, sondern auf einem *Substitutions-Permutations Netzwerk*, welches vereinfacht aus einer Serie von verknüpften Operationen besteht, welche zum einen die Eingaben mit speziellen Ausgaben austauscht (*Substitution*) und zum anderen die Bits durcheinander mischt (*Permutation*).

Die internen Strukturen von DES und AES werden u. a. von Paar und Pelzl im Werk *„Understanding Cryptography“* [Paar und Pelzl 2009] detailliert erläutert und können dort weiterführend nachvollzogen werden.

Betriebsarten

Für Block-Chiffren werden verschiedene Betriebsarten (auch Betriebsmodus genannt) eingeführt, mit denen festgelegt wird, wie die Verschlüsselung mehrerer Klartext-Blöcke vollzogen werden soll. Die Betriebsarten variieren in deren Sicherheit und Fehleranfälligkeit, bringen je nach Anwendung verschiedene Vor- und Nachteile mit sich. Bekannte Betriebsarten sind hierbei:

- Electronic Code Book mode (ECB),
- Cipher Block Chaining mode (CBC),
- Cipher Feedback mode (CFB),
- Output Feedback mode (OFB)
- Counter mode (CTR)

Während der Modus ECB, wie im klassischen Gedanken jeweils alle Klartextblöcke ohne Rückkopplung mit dem Schlüssel in Chiffretextblöcke umwandelt, arbeiten andere Modi bspw. mit Rückkopplung, indem wie bei CBC oder CFB der erstellte Chiffretextblock direkt mit dem nächsten Klartextblock verrechnet wird und erst dann wieder mit dem Schlüssel chiffriert wird. Dies hat den Vorteil, dass gleiche Klartextblöcke innerhalb der Nachricht immer unterschiedliche Chiffretexte ergeben, jedoch auch den Nachteil der Fehlerfortpflanzung (engl. error propagation), da im Fehlerfall alle darauffolgenden Blöcke falsch verschlüsselt werden. Andere Modi, wie OFB oder CTR nutzen die Verschlüsselungsfunktion zur Berechnung einer Ausgabe, die dann mit dem Klartextblock und der Operation XOR verknüpft wird. Bei OFB wird diese Ausgabe als Rückkopplung an den nächsten Block weitergegeben, wo diese wieder mit dem Schlüssel verschlüsselt wird. Für den ersten Block wird dabei ein sogenannter *Initialisierungsvektor (IV)* genutzt. Dieser bezeichnet einen zufällig ausgewählten Block, welcher im Regelfall für jede Nachricht neu ausgewählt werden muss und der bewirkt, dass bei gleichen Klartextblöcken unterschiedliche Chiffretextblöcke entstehen. CTR hingegen nutzt keine Rückkopplung und keinen zufällig ausgewählten IV, sondern arbeitet mit einer einmalig genutzten Zufallszahl (*Nonce*) und einem Zähler, welcher mit jedem weiteren Klartextblock hochgezählt wird. Diese Blöcke werden mit Hilfe des Schlüssels chiffriert und wie bei OFB mit dem Klartext über XOR verknüpft.

2.2. Asymmetrische Kryptografie

Asymmetrische Kryptosysteme wurden ursprünglich erfunden, um typische Probleme symmetrischer Systeme zu lösen. Wie bereits erläutert basieren symmetrische Kryptosysteme auf einem gemeinsamen Schlüssel, welcher von zwei Parteien zum Verschlüsseln und Entschlüsseln benutzt wird. Dieses Konzept birgt Einschränkungen, die mit asymmetrischer Verschlüsselung aufgelöst werden können [Martin 2012; Paar und Pelzl 2009]:

Abbildung 2.4.: Analogie für symmetrische Verschlüsselung (Quelle: [Paar und Pelzl 2009])

Symmetrisches Vertrauen Da Sender und Empfänger den gleichen Schlüssel teilen, bedeutet dies gleichzeitig, dass das gegenseitige Vertrauen Beider vorausgesetzt ist. Dabei ist zu bedenken, dass der Empfänger mit Besitz des Schlüssels grundsätzlich die gleichen Operationen durchführen kann wie der Sender. Zusätzlich ist es unmöglich, das Sicherheitsziel *Verbindlichkeit* einzuhalten, welches in Kapitel 2 beschrieben wurde. Die Abbildung 2.4 verdeutlicht dieses Problem anhand eines Beispiels, in dem `Alice` und `Bob` einen identischen Schlüssel für einen Safe haben.

Schlüssel-Aushandlung Um symmetrische Kryptosysteme zu nutzen, müssen sich Sender und Empfänger zunächst auf einen Schlüssel einigen. Dafür müssen beide einen zusätzlichen sicheren Mechanismus besitzen.

Anzahl an Schlüsseln Es ist zudem notwendig, für jeden Kommunikationspartner einen separaten Schlüssel festzulegen, was schnell zu einer unverwaltbar hohen Anzahl an Schlüsseln führen kann (Schlüsselverteilungsproblem).

Um das Prinzip von asymmetrischer Kryptografie (oder auch oft bekannt als *Public-Key Kryptografie*) zu erläutern, wird auf die Analogie mit dem Safe zurückgegriffen. Die Abbildung 2.5 zeigt den Safe mit zwei unterschiedlichen Schlüsseln, dem öffentlichen Schlüssel zum Verschlüsseln einer Nachricht (hier zum Einwerfen eines Briefes) und dem privaten (geheimen) Schlüssel zum Entschlüsseln dieser (hier zum Beschaffen des Briefes aus dem Safe).

Abbildung 2.5.: Analogie für asymmetrische Verschlüsselung (Quelle: [Paar und Pelzl 2009])

Um diesen Mechanismus zu realisieren, benötigt `Bob` ein Schlüsselpaar bestehend aus einem *public key* und einem passenden *private key*. Dadurch, dass der *public key* öffentlich ist, wird kein komplexer sicherheitsrelevanter Schlüsselaustausch mehr zwischen beiden (oder mehreren) Parteien benötigt. `Alice`, der Sender bekommt den Schlüssel frei lesbar bspw. via Mail zugesendet oder holt sich diesen etwa aus dem Internet. Alle Nachrichten, welche mit dem *public key* von `Bob` verschlüsselt werden, können mit dem geheimen *private key* von `Bob` entschlüsselt werden. Technisch wird das Schlüsselpaar über ein bekanntes Prinzip generiert, der *Einweg-Funktionen*, welches im folgenden Abschnitt 2.2 kurz erläutert wird. Durch dieses Prinzip ist es

zwar möglich aus dem privaten Schlüssel einen eindeutigen öffentlichen Schlüssel zu generieren, wohingegen dies umgekehrt nicht möglich ist.

Das Prinzip der asymmetrischen Kryptografie lässt sich nicht ausschließlich für die Chiffrierung und Dechiffrierung nutzen, sondern wird ebenfalls für andere Sicherheitsmechanismen genutzt, wie mit der Vorstellung der Einschränkungen von symmetrischer Kryptografie vorweggenommen wurde. Die Hauptanwendungen, die mit asymmetrischen Kryptografie umgesetzt werden können, sind:

Schlüsselaushandlung Die bekanntesten Protokolle, die geheime Schlüssel aushandeln, sind der Diffie-Hellman Key Exchange (DHKE) und der Rivest-Shamir-Adleman Algorithm (RSA) key transport.

Nichtabstreitbarkeit Die Sicherstellung von Integrität und Verbindlichkeit kann über Algorithmen digitaler Signaturen erreicht werden. Beispiele sind RSA, Digital Signature Algorithm (DSA) oder Elliptic Curve Digital Signature Algorithm (ECDSA).

Identifikation Mit Anwendung von *Challenge-and-response* Protokollen und digitalen Signaturen können Kommunikationspartner eindeutig identifiziert werden.

Verschlüsselung Nachrichten werden heutzutage über Algorithmen wie RSA oder Elgamal verschlüsselt.

Abschließend ist zu erwähnen, dass asymmetrische Systeme wesentlich langsamer sind als symmetrische, weswegen viele aktuelle Protokolle hybride sind und Konzepte beider nutzen. So wird beispielsweise asymmetrische Kryptografie zum signierten Schlüsselaustausch genutzt und symmetrisch mit dem entsprechenden Schlüssel verschlüsselt. Beispiele sind das bekannte SSL/TLS Protokoll für verschlüsselte Web-Verbindungen oder IPsec, welches sichere Kommunikation über IP-Netze ermöglicht.

Einweg-Funktionen

Eine Funktion $f()$ ist eine Einweg-Funktionen (engl. *One-way function*), wenn:

1. $y = f(x)$ leicht zu berechnen ist und die Umkehrfunktion
2. $x = f^{-1}(y)$ zu berechnen unausführbar ist

Während „leicht“ mit schnell und in polynomialer Zeit gleichzusetzen ist, soll „unausführbar“ bedeuten, dass das Lösen rechenintensiv und lang-andauernd ist, sodass selbst mit dem besten Algorithmus und stärkster Rechenleistung, Ergebnisse erst in

 © Philip Phu Dang Hoan Nguyen

unzumutbarer Zeit verfügbar wären [Paar und Pelzl 2009]. Zwei beliebte Einwegfunktionen, welche in standardisierten Public-Key-Verfahren genutzt werden, sind zum einen das Problem der *Primfaktorzerlegung*, auf dem RSA basiert, und zum anderen das *Diskreter Logarithmus-Problem (DLP)*, auf dem bspw. DSA oder der Diffie-Hellman Schlüsselaustausch basiert.

Bei dem Problem der *Primfaktorzerlegung* geht es um die *Faktorisierung* von Primzahlen. Es ist zunächst mathematisch leicht, das Produkt aus zwei Primzahlen zu errechnen, da dies jeder Taschenrechner beherrscht und in polynomialer Zeit durchgeführt werden kann. Geht es um den umgekehrten Weg, bei dem alleinig das Produkt bekannt ist und die sogenannte *Faktorisierung* errechnet werden soll, wird das Problem spätestens bei großen Zahlen komplex und rechenintensiv. Das Paar dieser Prim-Faktorisierung ist hierbei eindeutig (injektiv), weswegen die *Primfaktorzerlegung* als Einweg-Funktionen gut geeignet ist. Bei dem *DLP* geht es um modulare Exponentiation/Potenzierung. Dabei wird eine Zahl a mit einer anderen Zahl b potenziert und anschließend ein Modulo mit einer Primzahl p berechnet, welches ebenfalls in polynomialer Zeit berechenbar ist. Die Rückrichtung erweist sich im Gegensatz dazu als schwieriges Problem, wenn einzig das Ergebnis der modularen Exponentiation mit den Nummern a und n gegeben ist und b berechnet werden muss. Die *Einweg-Funktionen* würde somit lauten:

$$f(b) = a^b \mod n \tag{2.1}$$

2.3. Datenintegrität

Im vorherigen Abschnitt wurden grundlegende Kryptografie-Mechanismen vorgestellt, die besonders Sicherheitsziele wie *Datenschutz* und *Vertraulichkeit* behandelten. Bei der Datenintegrität handelt es sich, wie bereits beschrieben um den Schutz vor unautorisierter und unbemerkter Veränderung von Daten. Die Datenintegrität lässt sich nach Martin [Martin 2012] in verschiedene Ebenen aufteilen, welche u.a. durch unterschiedliche kryptografische Verfahren gelöst werden können:

Zufallsbedingte Fehler In dieser Ebene müssen Mechanismen bestehen, die zufallsbedingte Fehler, etwa durch Rauschen im Kommunikationskanal oder „Bitkipper“ erkennbar machen. Die Integrität wird hierbei über fehlerkorrigierenden Code oder *Checksummen* gewährt, welche mit Hilfe der ursprünglichen Daten berechnet und den Daten angehängt werden (bekannt als Prüfziffern oder Paritätsbits).

Einfache Manipulationen In dieser Ebene müssen einfache Änderungen bemerkt werden, was durch *Checksummen* nicht gewährleistet ist, da diese leicht von einem Angreifer vorhergesagt und gesetzt werden können. Besser eignen sich dafür *Kryptografische Hashfunktionen*, die Hinweise liefern können, dass Daten manipuliert worden sind.

Aktive Angriffe Gegen aktive Angriffe sind *kryptografische Hashfunktionen* nur bedingt hilfreich, da ein Angreifer mit Hilfe einer Nachricht und dem *Hash* leicht nachvollziehen kann, welcher Algorithmus genutzt wurde, und somit jede beliebige Nachricht ohne großen Aufwand selbst mit einem gültigen Hash versehen kann. Diese Art von Datenintegrität benötigt zusätzlich eine gewisse *Authentifizierung*, welche durch den kryptografischen Mechanismus *Message Authentication Code (MAC)* gewährleistet werden kann.

Angriffe bezüglich Nichtabstreitbarkeit Auf dieser letzten Ebene muss die Datenintegrität zusätzlich noch auf *Nichtabstreitbarkeit/Verbindlichkeit* geprüft werden, da bei den bisher vorgestellten Ebenen beide Parteien die Daten mit einem *Hash/MAC* versehen können. Da *MACs* mit einem symmetrischen Schlüssel generiert werden, ist der Datenursprung nicht eindeutig, da beide Parteien fähig sind, Nachrichten zu erstellen und zu verschicken. Folglich wird ein Mechanismus benötigt, der eindeutige Verbindlichkeit garantieren kann, auch gegenüber einer dritten Partei. *Digitale Signaturen* decken alle Ebenen der Datenintegrität ab und werden in Verbindung mit *digitalen Zertifikaten* genutzt.

2.3.1. Kryptografische Hashfunktionen

Kryptografische Hashfunktionen (oder auch *krypt. Streufunktionen*) werden vielseitig für verschiedenste kryptografische Anwendungen genutzt. Eine Hashfunktion ist eine mathematische Funktion, welche beliebig lange Eingabewerte auf eine Zeichenfolge fester Länge abbildet. Genutzt wird eine starke Einwegfunktion, wobei diese anders als die in Abschnitt 2.2 beschriebenen Einwegfunktionen nicht *injektiv* ist und somit ebenso *kollisionsresistent* sein muss. Das bedeutet in einfachen Worten, dass unterschiedliche Eingaben den gleichen Hash ergeben können, was allerdings bei einer effizienten Hashfunktion mit geringer Wahrscheinlichkeit vorkommt und nicht gezielt beeinflussbar ist. Aus dem Hash lässt sich zumal der Eingabetext nicht zurückrechnen. Wie bereits erläutert können die Ausgaben, welche als Hashes bezeichnet werden, unter anderem gegen einfache Manipulationen eingesetzt werden und somit leichte Datenintegrität gewährleisten. Ein nützlicher Nebeneffekt von Hashes ist der sogenannte Lawineneffekt, der die Eigenschaft beschreibt, dass bei einer minimalen Änderung der Eingabe ein komplett andere Ausgabe entsteht.

Praktische Anwendung finden Hashes bspw. im Internet bei der Bereitstellung von Software, bei der zum Herunterladen zusätzlich ein passender Hash als Checksumme beigelegt ist. Diese kann genutzt werden, um zu prüfen, ob die Software korrekt und ohne Fehler herunter geladen werden konnte. Zum anderen kann ebenso mit Einschränkung sichergestellt werden, dass die Software nicht durch Angreifer schadhaft modifiziert wurde. Da bekannte Hashfunktionen wie bspw. Message-Digest Algorithm 5 (MD5) oder Secure Hash Algorithm (SHA) öffentlich und zusätzlich leicht und schnell zu berechnen sind, kann nach dem Herunterladen die selbe Hashfunktion (bspw. MD5, dessen Hash 128-Bit lang ist und als 32-Zeichen HEX-Wert dargestellt wird) auf die Datei angewendet und der Hash mit dem vom Hersteller bereitgestellten verglichen werden. In diesem Fall wird ein beliebig langer Eingabewert in eine feste, kurze und (fast) eindeutige Zeichenfolge „komprimiert“. Ein anderes Anwendungsbeispiel findet sich bei der Verwaltung von Passwörtern. Hier werden in den meisten Anwendungen Passwörter statt in Klartextform als Hashes in der Datenbank gespeichert, um diese zu verschleiern. Selbst mit Einsicht auf die Daten ist es somit nicht möglich Passwörter in Klartext wiederherzustellen, welches im Regelfall ohnehin nicht gewünscht ist. Gibt ein Benutzer beim Einloggen sein Passwort in das System ein, wird somit ausschließlich eine Hashfunktion auf dieses angewendet und mit dem Hash in der Datenbank verglichen.

2.3.2. Message Authentication Code (MAC)

Wie bereits kurz eingeführt wird ein MAC als kryptografische Checksumme genutzt und bietet gewisse *Datenintegrität* und *Authentifizierung* beim Versenden von Nachrichten. MACs sind grundlegend von den Eigenschaften ähnlich zu Hashfunktionen mit der Ausnahme, dass zusätzlich ein symmetrischer Schlüssel involviert wird. Dieser wird auf der Senderseite zum Generieren des sogenannten *„authentication tag“* und auf der Empfängerseite zum Verifizieren verwendet. Das Verfahren ähnelt der Verwendung von Hashes als Checksumme und wird in Abbildung 2.6 gezeigt.

Damit eine Manipulation einer Nachricht erkannt werden kann, teilen sich `Bob` und `Alice` einen Schlüssel k, welcher genutzt wird, um die Nachricht x mit einem *authentication tag* m zu versehen. Mit Hilfe des MAC unter Eingabe der Nachricht und dem Schlüssel wird dieser *authentication tag* generiert und zusammen mit der Nachricht an `Alice` geschickt. Da `Alice` den Schlüssel mit `Bob` teilt, kann sie in der Konsequenz ebenfalls das passende *authentication tag* zu der Nachricht berechnen und vergleichen.

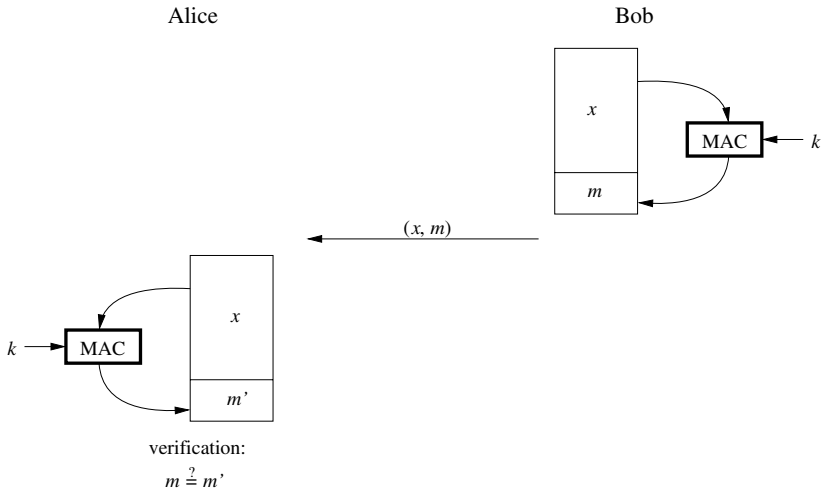

Abbildung 2.6.: Prinzip von Message Authentication Codes (MACs) (Quelle: [Paar und Pelzl 2009])

2.3.3. Digitale Signaturen

Ein MAC bietet zwar die Möglichkeit, beim Nachrichtenaustausch *Datenintegrität* mit *Authentifizierung* zu gewährleisten, jedoch adressiert es nicht die Probleme des *symmetrischen Vertrauens*, siehe Abschnitt 2.2. Vorherige Verfahren schützen zwar die Kommunikationspartner vor Außenstehenden, doch nicht vor deren Kommunikationspartner selbst. `Alice` hat durch das symmetrische Verfahren bspw. die Möglichkeit, eine gültige Nachricht mit einem authentication tag zu erstellen, und kann behaupten, dass `Bob` diese an sie geschickt hat.

Mit Hilfe von digitalen Signaturen ist es Dritten unabhängig möglich, Nachrichten auf *nichtabstreitbare Urheberschaft* zu prüfen. Digitale Signaturen sind heutzutage weit verbreitet, sie reichen von digitalen Zertifikaten für den sicheren Identitätsnachweis bis zu „elektronischen Unterschriften“ im elektronischen Handel (E-Commerce).

Digitale Signaturen basieren auf einem asymmetrischen Verfahren und somit ebenfalls auf *Public-Key Kryptografie* (vgl. Abschnitt 2.2). Die Grundidee ist, dass die Person, die eine Nachricht signiert (bspw. bei einer Bestellung) einen privaten Schlüssel und der Empfänger oder die dritte Partei den öffentlichen Schlüssel dieser Person zum Verifizieren nutzt. Der Ablauf vom Signieren bis zum Verifizieren wird in Abbildung 2.7 veranschaulicht.

`Bob` generiert ein Schlüsselpaar mit einem *private key* k_{pr} und einem *public key* k_{pub}, wobei zweites an `Alice` weitergegeben wird. Beim Signieren der Nachricht x verarbeitet ein Algorithmus diese zusammen mit dem *private key* zu einer Signatur

 © Philip Phu Dang Hoan Nguyen

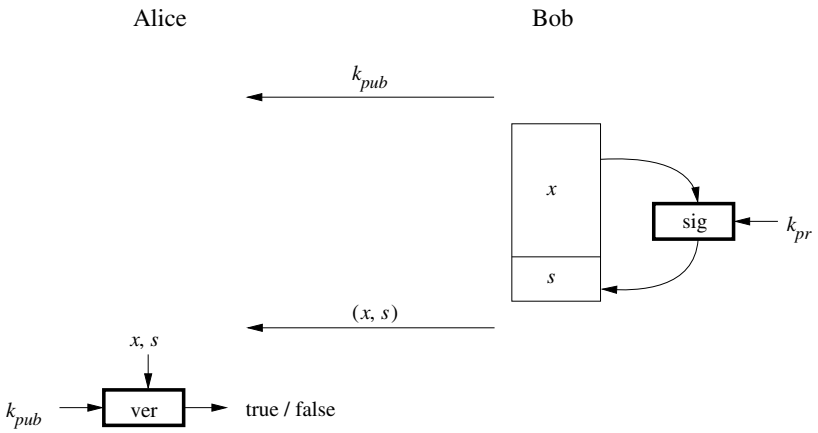

Abbildung 2.7.: Prinzip digitaler Signaturen (Quelle: [Paar und Pelzl 2009])

s, welche an die Nachricht angehängt wird und als Paar an `Alice` gesendet wird. Die digitale Signatur ist somit wechselnd und einzig im Zusammenhang mit einer Nachricht nutzbar. Die Signatur ist von der Repräsentation lediglich ein großer Zahlenwert (zB. ein String der Länge 2048 Bits).

`Alice` ist nun in der Lage, mit Hilfe einer Verifikationsfunktion und der Eingabe des *public keys* und der Signatur die Nachricht zu verifizieren.

Bekannteste Verfahren für digitalen Signatur sind unter anderem RSA, DSA und ECDSA.

2.3.4. Digitale Zertifikate

Anders als bei der symmetrischen Kryptografie wird bei der asymmetrischer Kryptografie kein Schlüsselaustausch benötigt, weil der öffentliche Schlüssel frei und unverschlüsselt an jeden Empfänger gegeben werden kann. Zwar gibt es keine effektive Möglichkeit aus dem öffentlichen Schlüssel den privaten Schlüssel zu errechnen, jedoch benötigt ein Angreifer diesen nicht zwingend. Bei einem sog. *„Man-in-the-Middle-Angriff“* stellt sich ein Angreifer (in der Analogie `Bob` und `Alice` ebenfalls `Oscar` genannt) zwischen die Kommunikationspartner und verfälscht die Kommunikation. Konkret fängt `Oscar` bspw. den öffentlichen Schlüssel von `Bob` ab, den `Bob` versucht an `Alice` zu schicken und ersetzt ihn durch seinen eigenen Schlüssel. Am Schlüssel ist für `Alice` nicht erkennbar, ob dieser tatsächlich von `Bob` kommt - Authentifizierung ist dementsprechend nicht gewährleistet. Da `Oscar` das Schlüsselpaar eigenständig generiert hat, ist er im Besitz des passenden, geheimen, privaten

Schlüssels und kann im Beispiel der Signierung alle Nachrichten selbständig signieren. *Man-in-the-Middle-Angriffe* sind für jede Art von asymmetrischer Kryptografie anwendbar, weswegen gegen diese Art von Angriffen ein Mechanismus vorliegen muss.

Um jeden öffentlichen Schlüssel mit Bestimmtheit authentifizieren zu können, wurden *digitale Zertifikate* eingeführt, welche die Identität von einem Benutzer oder Server an einen öffentlichen Schlüssel binden. Im Bereich der Netzwerk-Authentifizierung ist bspw. der Standard *X.509* essentiell für verschlüsselte Internet-Kommunikation und umfasst u.a. Informationen zum Zertifikatsinhaber (Name, DNS-Eintrag oder E-Mail), Aussteller, Signatur-Algorithmus und zur Gültigkeit. Kern dieser sog. *Public Key Infrastructure (PKI)* bilden vertrauenswürdige *Zertifizierungsstellen* (*certificate authority*, kurz *CA* genannt), welche Identitäten überprüfen und die Zertifikate mit Hilfe digitaler Signaturen beglaubigen und herausgeben. Die digitale Signatur dieser *CA* kann mit Hilfe des öffentlichen Schlüssels verifiziert werden (Funktionsweise siehe Abschnitt 2.3.4). Um nicht erneut *Man-in-the-Middle-Angriffen* zum Opfer zu fallen, muss der sicherer Transport des öffentlichen Schlüssels vom *CA* ebenso gewährleistet werden. Um dieses Problem zu lösen, liefern Software-Hersteller, wie bspw. Microsoft oder Browserhersteller, wie Mozilla oder Google diese Schlüssel heutzutage in ihrer Software mit.

Zertifikate, bspw. für gesicherte Webübertragungen (*SSL*), werden nicht von einer zentralen Zertifizierungsstelle herausgegeben, sondern von unzähligen *CAs* unterschiedlicher Länder, welche hierarchisch andere vertrauenswürdige *CAs* zertifizieren können. Am Ende der sog. *chain of trust* können sich Endbenutzer oder Firmen ihre Identität beglaubigen lassen.

 © Philip Phu Dang Hoan Nguyen

3. Grundlagen der Programmanalyse

Dieses Kapitel soll die Grundlagen und einen groben Überblick über Programmanalyse und deren Techniken geben. Da in der vorliegenden Arbeit die Analyse von Programm-Quelltexten im Vordergrund steht, wird der Fokus dieses Kapitels auf die statische Programmanalyse gerichtet. Diese findet sich generell nicht nur im Software Reverse-Engineering (oder auch Reengineering) Bereich wieder, sondern ist ebenso ein essentieller Bestandteil im Compilerbau. Koschke bestätigt dieses mit seinem Modell, welches in Abbildung 3.1 dargestellt wird, in welcher er statische Analysatoren von deren Struktur mit Compilern vergleicht (die tatsächliche Compiler-Struktur wird in Abschnitt 3.1 erläutert).

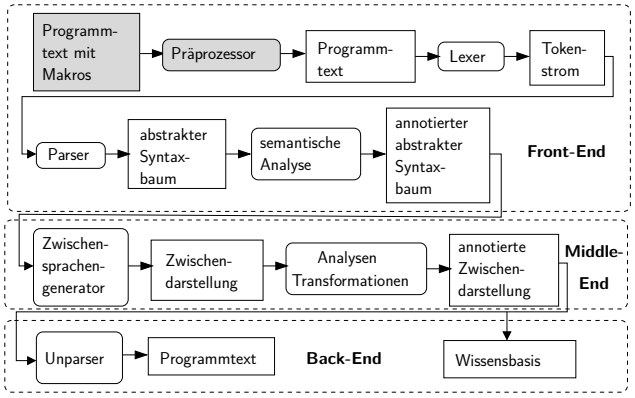

Abbildung 3.1.: Analysator- und Transformator-Struktur (Quelle: [Koschke 2014])

In der Abbildung 3.1 wird in der *Middle-End*-Ebene die annotierte Zwischendarstellung (Kapitel 3.2) erwähnt, welche sich als ideale Wissensbasis für statische Programmanalyse herausstellt. Koschke beschreibt zudem, dass bei Durchlauf der Compiler Phasen zunehmend mehr Wissen über das Programm gewonnen werden kann wie bspw.:

1. syntaktische Dekomposition
2. semantische Attributierung
3. Namensbindung
4. Kontrollflussinformation
5. Datenflussinformation
6. Abstrakte Syntaxbäume.

Aufgrund dessen wird im folgenden Abschnitt auf die verschiedenen Phasen und Konzepte des Compilers eingegangen, welche für die Programmanalyse und Programmoptimierung relevant sind. Das Verständnis dieser Grundbegriffe und Methoden bietet die nötigen technischen Grundlagen, um eine Überleitung zu den Analysemöglichkeiten herzustellen. Schlussendlich wird thematisch der Bogen über die verwendeten Zwischendarstellungen und Graphen zum Programm-Slicing gespannt, welches das Kernstück des Auditors und somit der vorliegenden Arbeit bildet.

Im Folgenden sollen im ersten Abschnitt 3.1 die Compiler-Struktur und explizit der Analyse-Teil des Compilervorgangs beschrieben werden. Es folgt eine Vorstellung der Zwischendarstellungen AST und SSA in Abschnitt 3.2 und anschließend eine Einführung in die Datenflussanalyse (Abschnitt 3.3). Die Erläuterungen zur Kontrollflussanalyse in Abschnitt 3.4 untergliedern sich in intra- und interprozedurale Abschnitte, in denen die wichtigsten Graphen-Formen vorgestellt werden. Diese bilden die Grundlage für den Abschnitt 3.5, in dem neben den Grundlagen des Slicings ebenfalls grob auf intra- und interprozedurale Techniken eingegangen wird.

3.1. Compiler-Struktur

Wie bereits in der Einleitung erwähnt, werden durch einen Einblick in den Compilerbau nützliche Erkenntnisse für die statische Programmanalyse gewonnen, da dort benutzte Konzepte im Reengineering wiederverwendet werden. Zunächst soll grob auf die Funktionsweise eines Compilers eingegangen werden.

Ein Compiler kann als Softwaresystem beschrieben werden, welches ein Programm (geschrieben in einer für Menschen lesbaren Programmiersprache) in eine Form übersetzen kann, in der es von Computern ausgeführt werden kann. Dieses resultierende Zielprogramm kann nun ein ausführbares Programm in Maschinensprache sein, das vom Benutzer aufgerufen werden kann, oder etwa wie bei Java Bytecode, welches eine Hardware-unabhängige Zwischenform darstellt, die später von einer virtuellen Maschine interpretiert wird. Ein Interpreter im Allgemeinen stellt hierbei eine andere Form von Sprachprozessor dar, welche die Operationen des Quellprogramms direkt mit den vom Benutzer vorgegebenen Eingaben ausführt.

Vorteile des Compilervorgangs sind nicht nur, dass das Zielprogramm in Maschinensprache ausführbar ist (und optimal an die jeweilige Zielplattform angepasst ist), sondern ebenso, dass das Quellprogramm auf syntaktische und semantische Wohlgeformtheit geprüft und zusätzlich noch durch ausführliche Analysen optimiert wird [Aho u. a. 2008].

© Philip Phu Dang Hoan Nguyen

Die Compiler-Struktur lässt sich, wie in Abbildung 3.2 veranschaulicht, zunächst grob in zwei Teilaufgaben zerlegen: Analyse und Synthese.

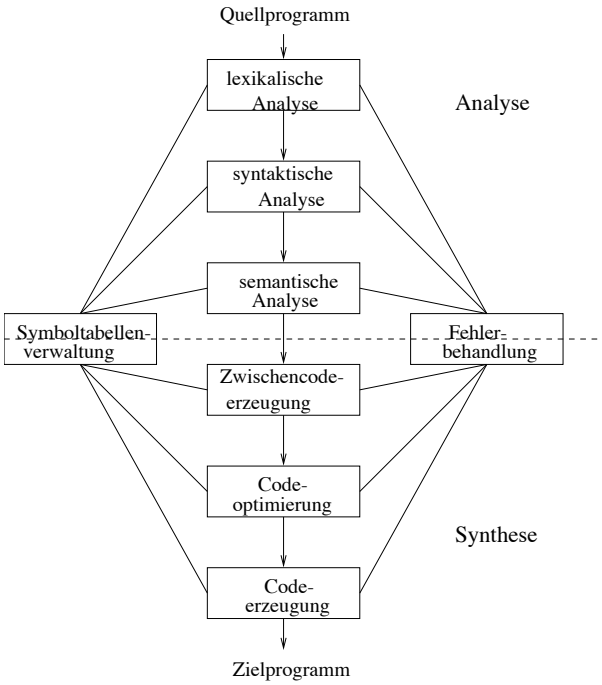

Abbildung 3.2.: Phasen eines Compilers (Quelle: [Goltz u. a. 2010] Abb. 1.2)

Aufgabe der Analyse ist es, das Quellprogramm in seine Bestandteile zu zerlegen, unter anderem auf Fehler zu analysieren und mit einer grammatischen Struktur zu versehen, welche in einem nächsten Schritt in eine Zwischendarstellung umgewandelt wird. Die Zwischendarstellung, Zwischencode oder im Englischen Intermediate Representation (IR) genannt, wird im zweiten Teil, der Synthese, weiter verarbeitet, bspw. um den Code zu optimieren und abschließend den Zielcode zu erzeugen. Im Kapitel 3.2 wird auf verschiedene Zwischendarstellungen eingegangen wie bspw. Abstract Syntax Tree (AST), Static Single Assignment Form (SSA) oder Three Address Code (TOC). Die Datenstrukturen, die Konstrukte zum Quellprogramm beinhalten, wie etwa Bezeichner, Typ oder Speicherplatz und werden später dazu verwendet, den Zielcode zu erzeugen [Aho u. a. 2008; Wilhelm u. a. 2013].

Im Folgenden wird auf die für die Arbeit relevanten Compiler-Phasen eingegangen: die lexikalischen Analyse, die syntaktische Analyse und die semantische Analyse, sowie Teile der Zwischencodeerzeugung.

Wie die restlichen Phasen der Synthese, demzufolge die Codeoptimierung und -erzeugung im Detail aussehen, beschreiben bspw. Aho, Lam, Sethi und Ullman [Aho u. a. 2008] und soll an dieser Stelle nicht weiter betrachtet werden.

3.1.1. Lexikalische Analyse

Die lexikalische Analyse liest den Zeichenstrom aus dem Quelltext und zerlegt diesen in bedeutungsvolle Zeichenfolgen, welche Lexeme genannt werden. Diese Lexeme sind die kleinste atomare Einheit der jeweiligen Programmiersprache und lassen sich in verschiedene Kategorien/Klassen gruppieren, wie bspw. Bezeichner, Zuweisungen, Operatoren, reservierte Schlüsselwörter, Zahlen oder Literale. Diese erhalten folgend abstrakte Symbole wie bspw. **Id** für Identifier (Bezeichner), **String** oder **Number**. Für jedes Lexem wird von einem sogenannten Scanner/Lexer ein Token instanziiert (die Abarbeitung der regulären Ausdrücke für die Mustererkennung kann hierbei über den Formalismus endlicher Automaten beschrieben werden), wodurch ein Tokenstream entsteht, welcher später an den Parser weitergegeben werden kann. Dabei müssen nicht nur Schlüsselwörter und Bezeichner voneinander unterschieden werden, sondern ebenfalls Konstanten, Kommentare und Leerzeichen richtig erkannt und als Token ausgegeben. Ein Token stellt hierbei eine Repräsentation des Lexems dar, bspw. in folgender Form

$$< Tokenname, Attributwert >,$$

welche die Kategorie der lexikalischen Einheit und optional einen Attributwert speichert. Die Variable *abc*, könnte somit als Paar (**Id**, „*abc*“) mit der Kategorie **Id** und dem gefundenen Lexem abgespeichert werden. In einem weiteren Schritt können diese Token noch verworfen werden, wenn sie für den Compilevorgang nicht relevant sind (bspw. Schlüsselwörter und Kommentare) oder noch in ihrer internen Form verbessert werden. Der als „Screening“ bezeichnete Vorgang wandelt den Bezeichner „*abc*“ in eine eindeutige Nummer, um diese mit einer Symboltabelle verknüpfen zu können. In dieser Datenstruktur können nicht nur weitere Informationen hinterlegt werden wie bspw. die Variablennamen oder -typen, sondern auch die Verweise auf die Stelle, an der diese Variable deklariert wurde. Dies löst unter anderem das Problem, dass Variablen und Methodennamen in einem Programm nicht eindeutig sind, trotz alledem richtig zugeordnet werden können. [Aho u. a. 2008; Wilhelm u. a. 2013].

3.1.2. Syntaktische und semantische Analyse

Die syntaktische Analyse wird von einem Parser durchgeführt, welcher den Tokenstream vom Lexer erhält (wie Abbildung 3.3 zeigt), diesen auf korrekte Grammatik der Programmiersprache prüft und wohlgeformte Programme in einen Parse-Baum wandelt. Somit wird die syntaktische Struktur eines Programmtextes überprüft und kann in einem weiteren Schritt in eine Zwischendarstellung gebracht werden.

© Philip Phu Dang Hoan Nguyen

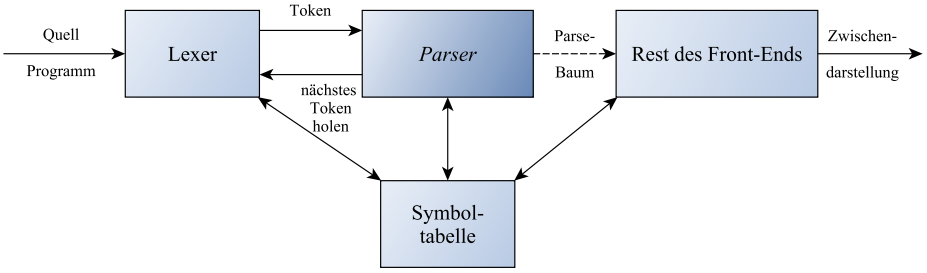

Abbildung 3.3.: Interaktion zwischen Lexer, Parser und restlichem Front-End (Quelle: Angelehnt an [Aho u. a. 2008] Abb. 4.1)

Um Tokenstreams auf lexikalische (Tippfehler), syntaktische (fehlende oder überflüssige Klammern) oder semantische Fehler (falsche Typisierung) untersuchen zu können, reichen Analysen mit endlichen Automaten über reguläre Ausdrücke nicht aus. Grund hierfür sind die komplexen syntaktischen Strukturen, die bspw. mit Hilfe von Klammerung mehrfache Verschachtlung von arithmetischen Ausdrücken bilden können (Code Nesting). Diese kann einzig über kontextfreie Grammatiken beschrieben werden. Anstatt über endliche Automaten konstruieren Parser sog. Kellerautomaten, welche eine definierte Sprache und demzufolge eine bestimmte Grammatik akzeptiert [Goltz u. a. 2010]. Die Konstruktion dieser Automaten über Top-Down oder Bottom-Up-Verfahren sowie tiefer gehende Erläuterungen der Grammatiken übersteigen den Rahmen dieser Arbeit, weswegen hier auf Literatur von Aho u.a. [Aho u. a. 2008] und Wilhelm u.a. [Wilhelm u. a. 2013] verwiesen wird.

Die Grammatik besteht dabei aus:

Terminale kann als Synonym für Tokens genommen werden und beschreibt bspw. alle Schlüsselwörter und Symbole wie Klammern.

Nichtterminale sind syntaktische Variablen, wie bspw. Anweisungen (*stmt*) und Ausdrücke (*expr*).

Startsymbol ist zumeist die linke Seite der ersten Produktion.

Produktionen beschreiben jeweils eine Methode, wie Terminale und Nichtterminale zu Strings zusammengesetzt werden können. Dabei steht auf der linken Seite ein Nichtterminal, welches zu ersetzen ist und auf der rechten Seite ein Rumpf, der aus null oder mehr Terminalen und Nichtterminalen besteht).

Das Definieren der kontextfreien Grammatik hat nicht nur den Vorteil, dass ein Programm auf Korrektheit überprüft werden kann, sondern auch, dass es in Form von Parserbäumen (auch Strukturbäume oder konkrete Syntaxbäume genannt) zur

Generierung der Zwischendarstellung des Programms genutzt werden kann [Aho u. a. 2008; Wilhelm u. a. 2013].

Die semantische Analyse des Compilers ergänzt den Syntaxbaum und die Symboltabelle mit Zusatzinformationen (attributierte Grammatiken) und ermöglicht die Prüfung der statischen Semantik des Programmes, ergänzend zu der lexikalischen und syntaktischen Struktur. Dazu zählen unter anderem Definiertheit, Sichtbarkeit, Gültigkeit (sind verwendete Bezeichner sichtbar deklariert?) und die richtige Typisierung von Variablen (entsprechen die zugeordneten Werte dem definierten Typen?) [Goltz u. a. 2010].

3.2. Zwischendarstellungen

Wie bereits im vorherigen Abschnitt 3.1 beschrieben, können Zwischendarstellung (IR) im Analyse-Synthese-Modell des Compilers verschiedene Formen annehmen. Je nach Compiler und Verwendungszweck können ebenso mehrere Zwischendarstellungen erzeugt werden. Man unterscheidet bei den Darstellungen zwischen höherer und niedriger Ebene, wobei höhere Ebenen näher an der Quellsprache sind und niedrigere Ebenen der Zielmaschine, demzufolge linear und dem Maschinencode näher sind. Unter die erste Kategorie fallen vor allem strukturelle graphen-orientierte Darstellungen, wie gerichtete azyklische Graphen oder die erwähnten Syntaxbäume. Als Beispiel für Zwischendarstellungen niedriger Ebenen könnten Bytecode oder Drei-Adress-Code (TOC) genannt werden. Im Folgenden sollen relevante Darstellungen erläutert werden und nach Einführung in die Datenflussanalyse ebenfalls auf Darstellungen wie Kontrollfluss-Graphen eingegangen werden, welche eine hybride Kombination aus Graph und linearem Code bilden.

3.2.1. Abstrakte Syntaxbäume

Wie im vorherigen Abschnitt beschrieben, können für die syntaktische Analyse konkrete Syntaxbäume genutzt werden, doch sind diese für die darauffolgenden Compiler-Phasen nicht weiter relevant. Mehrfach verschachtelte bzw. verkettete Einzelproduktionen mit nur Nichtterminalen dienen bspw. lediglich als Helfer, um Ausdrucksarten wie Terme oder Faktoren darzustellen und können theoretisch aus dem Baum entfernt werden, woraus eine vereinfachte Abstraktion entsteht.

Ein sogenannter abstrakter Syntaxbaum (engl. Abstract Syntax Tree, AST) repräsentiert die hierarchische Struktur des Quellprogramms in einer anderen Grammatik, in der es keine Nichtterminale mehr gibt. Die strukturerhaltende Abstraktion kann

© Philip Phu Dang Hoan Nguyen

hierbei entweder vom konkreten Syntaxbaums übersetzt oder wie in vielen Compilern direkt vom Parser generiert werden. Anders als bei den konkreten Syntaxbäumen beschreiben die inneren Knoten einen Operator, dessen Kinder den Operanden entsprechen. Dadurch ist es möglich mit den inneren Knoten ganze Programmierkonstrukte abzubilden und durch weitere Tiefe des Baumes Verschachtelungen und Sequenzen ideal darzustellen. Neben diesen syntaktischen Kanten, welche die syntaktische Dekomposition repräsentieren, können über den Parser auch zusätzliche Attributierungen vorgenommen werden. Diese können als semantische Kanten Verweise auf andere Knoten bilden und somit bspw. Namensverwendungen und Deklarationen verknüpfen.

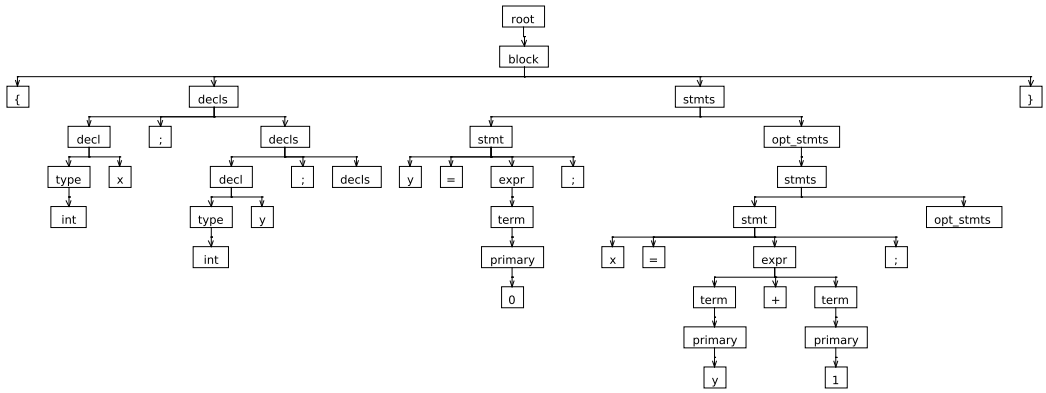

Abbildung 3.4.: Beispiel eines konkreten Syntaxbaums (Parserbaum) (Quelle: [Koschke 2014])

Abbildung 3.4 und Abbildung 3.5 sollen den Unterschied zwischen den beiden Syntaxbäumen zu folgendem Beispiel-Programmteil zeigen:

```
1 { int x;
2   int y;
3   y = 0;
4   x = y + 1;
5 }
```

3.2.2. Static Single Assignment Form (SSA)

Die Static Single Assignment Form, kurz SSA, ist eine weitere Zwischendarstellung, welche heutzutage bei zahlreichen Compilern benutzt wird, wie bspw. im JIT Compiler von Oracles HotSpot Java VM, bei der GNU Compiler Collection (GCC) oder in der Androids Dalvik Virtual Machine. SSA wurde erstmals von Cytron et

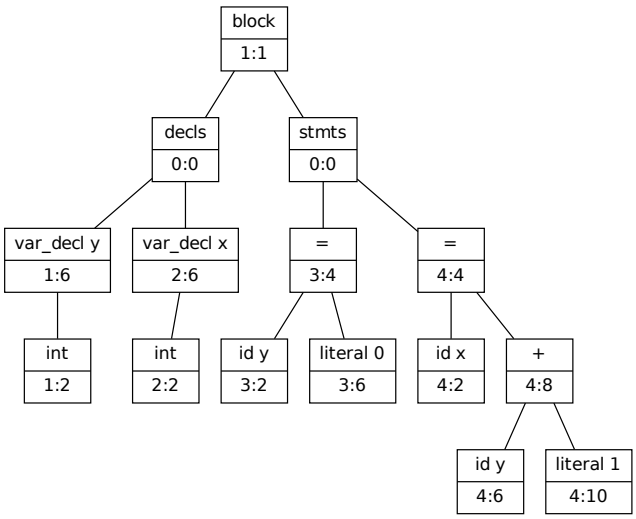

Abbildung 3.5.: Beispiel eines abstrakten Syntaxbaums (Quelle: [Koschke 2014])

al. vorgestellt [Cytron u. a. 1991] und bietet eine Datenstruktur, welche verschiedene Programmoptimierungen vereinfachen soll. Gerade in Bereichen Datenfluss und Kontrollfluss ist SSA effektiv, was ebenso für später angewandte Programmanalyse-Techniken wie das Slicing von großer Bedeutung ist und im Abschnitt 5.2.1 genauer erläutert wird. Optimierungen, welche von der Darstellung in SSA profitieren können, sind bspw.:

Codeverschiebung (Code motion): Zieht sich nicht verändernde Variablen aus Schleifen, um mehrfache Berechnung/Zuweisung zu verhindern.

Eliminierung von redundanter gemeinsamer Teilausdrücke (Common-subexpression elimination): Benutzt existierende Werte, die vorher errechnet wurden und Ausdrücken zugeordnet werden.

Eliminierung von totem Code (Dead-code elimination): Entfernt nicht erreichbaren Code bzw. Variablen.

Konstantenpropagation (Constant folding): Beschreibt, wie zur Laufzeit Variablen überprüft werden, ob diese vom Wert konstant bleiben. Sollte dies zutreffen werden diese im Datenfluss als Konstante behandelt.

Kopierpropagation (Copy propagation): Entfernt redundante Kopieranweisungen von Variablen ($z = y$), indem direkt der Wert der rechten Seite genutzt wird.

Die hier genannten Codeoptimierungen beruhen alle auf der Datenflussanalyse, welche im nächsten Abschnitt detailliert beschrieben wird.

Ein besonderes Merkmal der SSA ist, wie der Name andeutet, dass alle Zuweisungen zu Variablen eindeutige Namen bekommen und somit jede Variablenverwendung (*Use*) lediglich eine Definition (*Set*) hat. *Set* und *Use* sind hierbei bekannte Begriffe aus der *Datenabhängigkeitsanalyse*, welche die Grundlagen für für die Optimierungen und SSA bilden.

Über künstliche Zuweisungen, sogenannte ϕ-Knoten, können Variablen mehrfach in verschiedenen Kontrollflusspfaden genutzt werden. Dafür muss bspw. am Anfang der Grundblöcke, an denen Kontrollflüsse zusammenlaufen, ein ϕ-Knoten gesetzt werden [Aho u. a. 2008; Koschke 2014; Matthias Braun u. a. 2013].

Folgendes Beispielprogramm sei gegeben:

```
1  public int test(boolean w) {
2    int x = 1;
3    if (w == true) {
4      int y = x + 5;
5    } else {
6      int y = x - 2;
7    }
8    int z = y * 2;
9    return z;
10 }
```

Das Beispielprogramm soll hierbei eine Methode `test()` sein, welche als Parameter den Boolean `w` übergeben bekommt. Abhängig von dieser Variable teilt sich der Ablauf in zwei Kontrollflusspfade auf, in welchem jeweils die Variable `y` initialisiert und mit Hilfe von `x` berechnet wird. Zum Schluss wird die Variable `z` mit Hilfe von `y` errechnet und letztendlich zurückgegeben.

Den zugehörigen normalen Kontrollflussgraphen betrachtend (später näher ausgeführt in Abschnitt 3.4.2) (siehe linke Abbildung 3.6), werden die Variablen `x` und `z` jeweils in einem Knoten, jedoch `y` auf zwei verschiedenen Kontrollflusspfaden im Programm bestimmt. Der Knoten `z = y * 2` ist in diesem Fall nicht eindeutig bestimmbar, da es aus einem der beiden Pfade stammen könnte.

Auf der rechten Seite der Abbildung 3.6 ist der in SSA-Form transformierte Kontrollflussgraph zu sehen, welcher für jede Zuweisung einer Variable eindeutige Namen hat. Da `y` jeweils in zwei verschiedenen Kontrollflusspfaden definiert wird, erstellt SSA einmalig zwei Variablen `y1` und `y2`. Anweisungen, welche Bezug auf die Variablen haben, bedienen sich der zuletzt definierten Version, während eine weitere Neuzuweisung eine neue Version `y3` erzeugt. Indizes unterscheiden in diesem Beispiel zwischen den Definitionen der Variablen `w, x, y` und `z`, während WALA bspw. die Variablen komplett über unterschiedliche Nummern identifiziert und erzeugt.

Durch den neu erzeugten ϕ-Knoten `y3 = ϕ(y1, y2)` wird nun der Wert des Arguments für den Kontrollflusspfad zurückgegeben, der eingeschlagen wurde, um diesen Knoten zu erreichen. Schließlich können alle Datenabhängigkeiten explizit dargestellt werden und die Variable `z1` bestimmt werden.

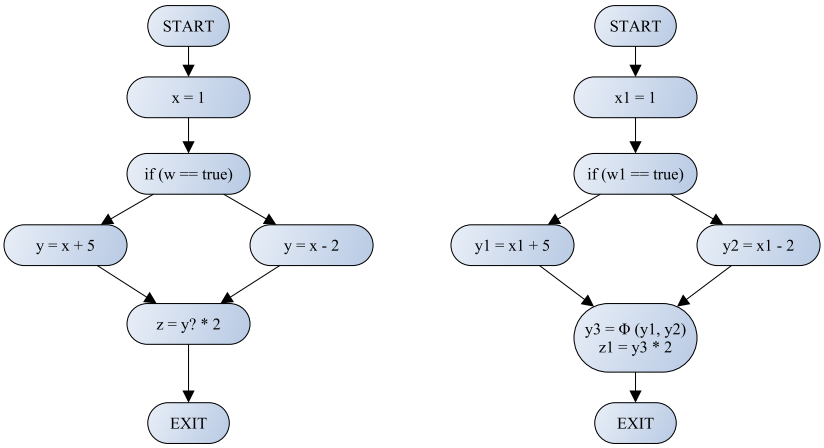

Abbildung 3.6.: KFG und KFG in SSA-Form zum Beispielprogramm

3.3. Datenflussanalyse

Die Datenflussanalyse beschäftigt sich mit der Ermittlung von Informationen über den Fluss der Daten entlang der Programmausführungspfade, demzufolge konkret zwischen zwei verschiedenen Programmpunkten. Wie bereits erwähnt, kann die Datenflussanalyse genutzt werden, um diverse Codeoptimierungen durchzuführen. Um toten Code zu finden, können bspw. nach Zuweisung einer Variable alle nachfolgenden Ausführungspfade untersucht werden (über Traversierung), ob diese Variable überhaupt genutzt wurde. Gleiches gilt für die im Abschnitt 3.2.2 genannten Optimierungen *code motion, common-subexpression elimination, constant folding* und *copy propagation* [Aho u. a. 2008].

Die Datenflussanalyse basiert auf *Erreichbarkeit* (eng. *reaching definitions*) in einem Flussgraphen G und beinhaltet als Knoten die Zuweisungen/Variablendefinitionen $d \in G$ zu dem jeweiligen Programmpunkt. Eine Definition d kann nun auf Erreichbarkeit an einem Programmpunkt p geprüft werden, wenn es einen Pfad $d \rightarrow *n \in G$ gibt, an dem d nicht überschrieben wurde (auch *kill* genannt). Es ist zu beachten, dass es bei der Abbildung vom Programmcode zu Datenflüssen keine eindeutige Lösung gibt, sondern nur Annäherungen (Approximation), da bspw. Transferfunktionen (bei denen Variablen umgeschrieben werden) oder der Kontrollfluss komple-

© Philip Phu Dang Hoan Nguyen

xere Einschränkungen hervorrufen kann (etwa wenn Kontrollflusskanten zwischen Grundblöcken bestehen) [Aho u. a. 2008; Krinke 2003; Robschink 2004].

3.4. Kontrollflussanalyse

Anweisungen wie *if-else* und *while* sind Ursache dafür, dass Programme selten sequentiell durchlaufen werden. Deren Auswertung wird mit booleschen Ausdrücken verknüpft und führt dazu, dass durch bestimmte definierte Bedingungen der Kontrollfluss geändert wird. Ein *Kontrollfluss* kann sich ebenfalls auf höherer Ebene (interprozedural) ändern, wenn das vollständige Programm mit all seinen Prozeduren/Funktionen betrachtet wird. Im Folgenden soll zwischen diesen beiden Analyse-Ebenen unterschieden und typische graphen-basierte Darstellungsformen vorgestellt werden [Aho u. a. 2008; Krinke 2003].

3.4.1. Intraprozeduraler Kontrollfluss

Der intraprozedurale Kontrollfluss wird für die meisten Compileroptimierungen genutzt. Solche Analysen sind *intraprozedural*, da sie jeweils innerhalb einer Prozedur zur gleichen Zeit gemacht werden. Die Knoten bestehen oftmals aus zusammengefassten Anweisungen (sog. Grundblöcken, engl. Basic-Blocks), während die Kanten bedingte oder unbedingte Kontrollflüsse repräsentieren. Solche Flussgraphen ergeben sich durch die syntaktische Struktur und Anweisungen wie *Goto*, *Exit* und *Continue* [Aho u. a. 2008; Krinke 2003].

3.4.2. Kontrollflussgraph

Die gängigste und meist genutzte Darstellungsform intraprozeduraler Kontrollflüsse ist der *Kontrollflussgraph*. Ein Kontrollflussgraph (engl. *Control Flow Graph (CFG)*) ist ein gerichteter attributierter Graph $G = (N, E, n^s, n^e, \nu)$, in dem die Anweisungen und Prädikate als Knoten ($n \in N$) und Kontrollflüsse als Kanten $(n, m) \in E$ dargestellt werden mit einem Startknoten $n^s \in N$ und einem Exitknoten $n^e \in N$ [Krinke 2003; Robschink 2004]. Diese an Knoten hängenden Prädikatskonstrukte (z.B. *if, while* oder *switch*) werden unterschiedlich logisch ausgewertet und unterscheiden somit, welche Anweisung als Nachfolger ausgeführt werden. Die Kanten dieser werden entsprechend durch $\nu : E \to true, false, \epsilon$ attributiert. Abbildung 3.7 zeigt ein Beispielprogramm und den dazu generierten Kontrollflussgraphen. Die Knoten sind mit der zugehörigen Programmzeile nummeriert, wobei ein Knoten ein Prädikat hat und somit zwei verschiedene Pfade einleitet.

```
1    a = read()
2    if (a>0) {
3      b = a + 1
4      c = a * 2
5    } else {
6      b = a - 1
7      c = a / 2
8    }
9    write(b)
10   write(c)
```

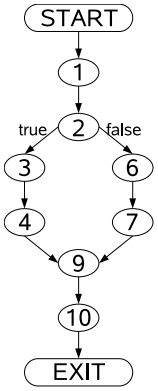

Abbildung 3.7.: Beispielprogramm mit abgeleiteten Kontrollflussgraphen (Quelle: [Krinke 2003])

3.4.3. Programmabhängigkeitsgraph

Ein *Programmabhängigkeitsgraph* (engl. *Program Dependence Graph (PDG)*) ist eine Abwandlung des Kontrollflussgraphen, bei dem die Kontrollfluss-Kanten mit anderen Kanten ausgetauscht werden, die zum einen Kontrollabhängigkeiten oder zum anderen die Datenabhängigkeiten beschreiben [Krinke 2003; Robschink 2004]. Dadurch werden die Auswirkungen der Kontrollflüsse und Erreichbarkeit etwas deutlicher dargestellt:

Kontrollabhängigkeiten existieren, wenn eine Anweisung die Ausführung einer anderen kontrolliert, während Datenabhängigkeiten existieren, wenn eine Variablendefinition eines Knotens einen anderen Knoten erreicht und dort genutzt wird.

Kontrollabhängigkeit

Die Definition von *Kontrollabhängigkeit* lässt sich folgendermaßen formulieren:

Zunächst wird ein Knoten M als *Postdominator* von einem anderen Knoten N bezeichnet, wenn alle Pfade von N zum Endknoten durch M verlaufen.

Ein Knoten m ist genau dann von einem Knoten n (direkt) *kontrollabhängig*, wenn

1. es einen nicht-leeren Pfad p von n nach m im CFG gibt,
2. m jeden Knoten auf dem Pfad p (außer n) *postdominiert* und
3. m kein *Postdominator* von n ist.

© Philip Phu Dang Hoan Nguyen

Das bedeutet grundsätzlich, dass der Knoten n mindestens zwei ausgehende Kanten besitzen muss, wobei alle Pfade der einen Kante durch m zum Endknoten verlaufen müssen (Voraussetzung 1) und alle Pfade der anderen Kante nicht durch m verlaufen (Voraussetzung 3). Somit *kontrolliert* der Knoten n die Ausführung vom Knoten m, da es auch einen anderen Pfad zum Endknoten gibt [Krinke 2003; Robschink 2004].

Für strukturierte Programme bedeutet das in anderer Form, dass in einem Kontrollflussgraph $G = (N, E, n^s, n^e, \nu)$ mit Kontrollprädikaten $n \in C$ (Schleifen oder bedingte Anweisungen wie *while* oder *if*) ein Knoten n' *kontrollabhängig* von n ist, wenn die Ausführung von n' abhängig von der Erfüllung von n ist. Die Abbildung 3.8 zeigt exemplarisch die Kontrollabhängigkeiten dreier Anweisungen innerhalb einer Schleife.

```
sum = 0;
mul = 1;
a = 1;
b = read();
while (a <= b) {
  sum = sum + a;
  mul = mul * a;
  a = a + 1;
}
write(sum);
write(mul);
```

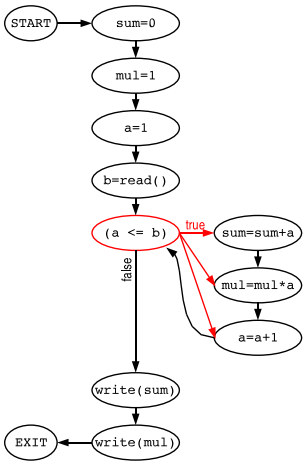

Abbildung 3.8.: Kontrollabhängigkeiten abgeleitet von einem Beispielprogramm (Quelle: [Robschink 2004])

Die Anweisungen `sum = sum + a`, `mul = mul * a` und `a = a + 1` sind kontrollabhängig von der Anweisung `(a <= b)` (rot markiert), da je nach Auswertung `true` oder `false` bestimmt wird, ob die drei Anweisungen überhaupt durchlaufen werden.

Datenabhängigkeit

Die Definition von *Datenabhängigkeit* lässt sich folgendermaßen formulieren:

Sei $set() \in N$ eine Variablendefinition und $use() \in N$ eine Variablenverwendung. Ein Knoten m ist *datenabhängig* von einem Knoten n genau dann, wenn

1. es einen Pfad von n nach m im CFG gibt,

2. es eine Variable ν gibt mit $\nu \in set(n)$ und $\nu \in use(m)$ und
3. für alle Knoten $k \neq n$ auf diesem Pfad p gilt, dass $\nu \notin set(k)$

Dieses Problem ist analog zur Definition von Erreichbarkeit in Flussdiagrammen, welches in Abschnitt 3.3 vorgestellt wurde, und beschreibt grundsätzlich lediglich, dass zwei Knoten *datenabhängig* sind, wenn eine vorher definierte Variable in einem anderen Knoten genutzt wird. Die Abbildung 3.9 zeigt zum gleichen Beispielprogramm die Datenabhängigkeiten, welches in einer Analyse helfen kann, bestimmte Datenflüsse zu verfolgen.

```
sum = 0;
mul = 1;
a = 1;
b = read();
while (a <= b) {
  sum = sum + a;
  mul = mul * a;
  a = a + 1;
}
write(sum);
write(mul);
```

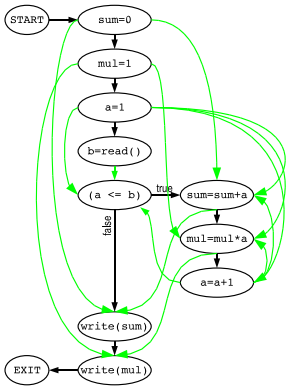

Abbildung 3.9.: Datenabhängigkeit abgeleitet vom selben Beispielprogramm (Quelle: [Robschink 2004])

Wie bereits erwähnt, verbindet ein *Programmabhängigkeitsgraph* beide Abhängigkeitsarten als Kanten in einem gerichteten Graphen, in dem die Knoten wie zuvor Anweisungen darstellen. Abbildung 3.10 zeigt einen vollständigen *Programmabhängigkeitsgraph*, in dem der Endknoten entfernt und ungeschachtelte Anweisungen zu unbedingten (*true*) kontrollabhängigen Knoten gewandelt wurden [Krinke 2003; Robschink 2004].

3.4.4. Interprozeduraler Kontrollfluss

Im letzten Abschnitt wurde die Analyse von intraprozeduralen Kontrollflüssen vorgestellt. Im echten Leben existieren allerdings keine Programme ohne Benutzung von Methodenaufrufen (Prozeduren) oder externer Bibliotheken. Generell ist die interprozedurale Analyse wesentlich komplexer, da bspw. Methodenaufrufe Variablen als Parameter an andere Prozeduren weitergeben können. Als Beispiel kann die Abbildung 3.12 dienen, welche einen interprozeduralen Kontrollflussgraphen (Interprocedural Control Flow Graph (ICFG)) zum Beispielprogramm in Abbildung 3.11 darstellt.

© Philip Phu Dang Hoan Nguyen

```
sum = 0;
mul = 1;
a = 1;
b = read();
while (a <= b) {
  sum = sum + a;
  mul = mul * a;
  a = a + 1;
}
write(sum);
write(mul);
```

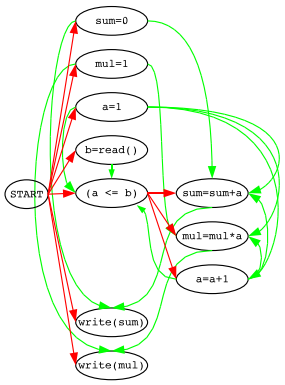

Abbildung 3.10.: Programmabhängigkeitsgraph (Quelle: [Robschink 2004])

```
1  int a, b, c;
2
3  void q () {
4    int z = 1;
5    a = 2;
6    b = 3;
7    p(4, z);
8    z = a;
9    c = 5;
10   p(6, c);
11 }
```

```
12 void p (int x, int& y) {
13   static int d = 6;
14   a = c;
15   if (x) {
16     d = 7;
17     p(8, x);
18   } else {
19     b = 9;
20   }
21   y = 0;
22 }
```

Abbildung 3.11.: Einfaches Beispielprogramm zum ICFG (Quelle: [Krinke 2003])

Das Programm beschreibt zwei Methoden `q()` und `p(int x, int y)`, wobei in `q()` die andere Methode aufgerufen wird, welche sich zusätzlich in bestimmten Zuständen ein weiteres Mal selbst aufruft. Der resultierende ICFG besteht aus mehreren Graphen, welche über interprozedurale Kanten in Verbindung gebracht werden [Krinke 2003].

Da gerade für das *Slicing* (näher erläutert in Kapitel 3.5 Slicing) die Abhängigkeiten verschiedener Prozeduren von großer Bedeutung sind, sollen zusätzlich noch andere, für die Arbeit relevante, Darstellungen der interprozeduralen Analyse vorgestellt werden.

3.4.5. Callgraph

Ein *Callgraph*, ebenfalls Aufrufgraph genannt, ist ein gerichteter Graph, welcher die Aufruf-Beziehungen zwischen den Prozeduren eines Programmes darstellt [Aho u. a. 2008; Koschke 2014]. Der Multigraph (erlaubt multiple Kanten mit gleichen Endknoten) definiert sich über seine Knoten, welche für Prozeduren stehen, und über

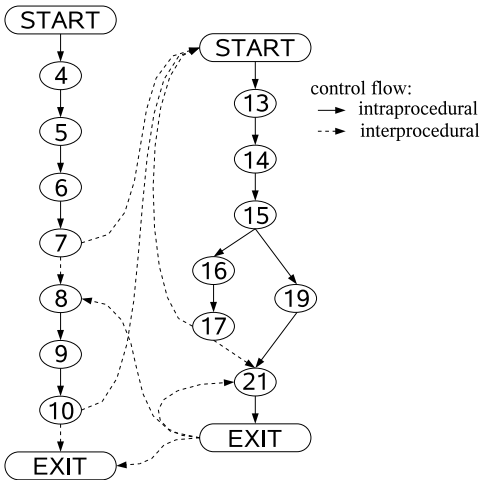

Abbildung 3.12.: Aus Abbildung 3.11 abgeleiteter ICFG (Quelle: [Krinke 2003])

die Kanten, welche Aufrufe darstellen. Jeder Knoten hat eine Menge an Aufrufstellen (engl. *call sites*), welche die Quelle mehrerer Kanten zu anderen Knoten bilden.

Der Graph ergibt sich aus expliziten Aufrufen im Programmcode oder aus Aufrufen über Funktionszeiger (engl. function pointer). Als Beispiel wird wiederholt das vorherige Programmbeispiel (Abbildung 3.11) als Grundlage genommen. Der daraus abgeleitete Callgraph wird in der Abbildung 3.13 verdeutlicht. Zu beachten ist, dass mehrfache Aufrufe (zweifacher Methodenaufruf von `p()` in `q()`) nur mit einer Kante repräsentiert werden.

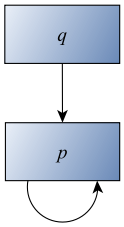

Abbildung 3.13.: Aus Abbildung 3.11 abgeleiteter Aufrufgraph

3.4.6. Systemabhängigkeitsgraph

Die in Abschnitt 3.4.3 eingeführten *Programmabhängigkeitsgraphen* deckten bislang lediglich intraprozedurale Abhängigkeiten ab. *Systemabhängigkeitsgraphen* (engl. *System Dependence Graph (SDG)*) bauen darauf auf und verbinden mehrere PDGs (in diesem Zusammenhang auch *Procedure Dependence Graph* genannt), um Programme auch interprozedural analysieren zu können.

Ein Systemabhängigkeitsgraph $G = (N, E)$ ist ein gerichteter Graph, in dem N die Anweisungen/Statements und Prädikate des Programms und E die Abhängigkeiten zwischen diesen darstellen. Neben den bekannten Kanten der PDGs zur Beschreibung der Kontroll- und Datenabhängigkeit werden die PDGs zusätzlich noch an sogenannten *call sites* miteinander verbunden. Die Verbindung besteht aus einem Aufruf-Knoten c (*call node*) und einer Aufruf-Kante (*call edge*) zum Funktionseintrittsknoten e (*entry node*) der nächsten Funktion/Methode. Da (multiple) Parametrisierung und Rückgabewerte des Funktionsaufrufes berücksichtigt werden müssen, führen SDGs zusätzlich ein:

- aktuelle Parameter: actual-in / actual-out-Knoten (copy-in / copy-out Parameterübergabe vorausgesetzt)
- formale Parameter: formal-in / formal-out-Knoten
- transitive Abhängigkeiten via PDG: sogenannte *Summary-Edges*

Die *Summary Edges* werden als Kanten zwischen actual-in and actual-out Knoten gesetzt, wenn diese kontrollabhängig vom Aufruf-Knoten c sind. Dadurch lassen sich Rückschlüsse über den transitiven Fluss von einem Parameter zum Rückgabewert der aufgerufenden Funktion ziehen. Besonders nützlich sind diese Informationen später beim Slicing, um unnötige Abstiege in aufgerufene Prozeduren zu vermeiden. [Graf 2010; Koschke 2014]

3.5. Programm-Slicing

Programm-Slicing, oder nur *Slicing* genannt, beschreibt eine Technologie, in der die Größe eines Programms über den Fokus auf einen bestimmten Programmteil reduziert wird. Es findet Verwendung in Phasen des Software-Entwicklungszyklus, unter anderem beim Debugging, bei Regressionstests und bei der Software-Wartung, spielt ebenfalls eine große Rolle beim Reverse Engineering [Binkley und Gallagher 1996], welches auch in der vorliegenden Arbeit beschrieben wird.

Weiser beschreibt das Konzept Slicing als Erster in seiner Publikation aus dem Jahr 1982 [Weiser 1982] zunächst als mentales Abstrahieren von wichtigen Programmzeilen, wie es Programmierer beim Debuggen vollziehen und publiziert später im Jahr 1984 [Weiser 1984], wie dieses (zunächst nur statisch) berechnet und formal definiert werden kann. Der *Slice* repräsentiert eine Teilmenge des kompletten Programms, welche in Abhängigkeit zu einem bestimmten Kriterium steht, dem sogenannten *Slice-Kriterium*. Mit Hilfe des Kriteriums, welches über Anweisungen und Variablen definiert wird, kann der Fokus auf einen bestimmten Programmpunkt gesetzt

werden. Ferner berechnet Weiser den *Slice* anhand eines Datenfluss-Algorithmus, welcher über Datenfluss-Graphen und *DEF/REF* Knoten (siehe Beschreibung der Reaching-Definitions in Abschnitt 3.3) die relevanten Variablen und Anweisungen bestimmt. Verbleibende Teile des Programms, die das Verhalten an dem Programmpunkt nicht beeinflussen, werden entfernt [Krinke 2003; Koschke 2014].

Anhand eines Beispiels (Programmcode siehe Abbildung 3.14) sollen die beiden unterschiedlichen Richtungen kurz verdeutlicht werden, in denen das Traversieren des Graphen an von einem Programmpunkt S aus durchlaufen werden kann.

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

Abbildung 3.14.: Beispielprogramm zur Analyse mit Slicing (Quelle: [Koschke 2014])

Ein *Forward Slice* enthält nur die Folge-Anweisungen eines Programms, die von der Ausführung vom S bzw. dem *Slice-Kriterium* beeinflusst werden. Im Teil (a) der Abbildung 3.15 wird somit über diese Slicing-Methode ermittelt, welche Variablen und Anweisungen die Zuweisung `sum := 0` beeinflussen. Da die Operationen `read(n)`, die Manipulationen von `product` und Schleifenausführung nicht davon abhängig sind, werden diese gestrichen (in der Abbildung ausgegraut und durchgestrichen) und sind somit nicht Teil des resultierenden *Slices*.

Beim *Backward Slice* werden Anweisungen und Variablen identifiziert, welche Einfluss auf das Programmverhalten an S haben. Teil (b) der Abbildung 3.15 verdeutlicht, dass die Anweisungen mit der Variable `sum` keinerlei Einfluss auf die Berechnungen der Variable `product` haben.

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

(a) Forward Slice (sum := 0)

(b) Backward Slice (write(product))

Abbildung 3.15.: Zwei Beispiel-Slices (Quelle: [Koschke 2014])

© Philip Phu Dang Hoan Nguyen

3.5.1. Intraprozedurales Slicing mit PDG

Moderne Slicing-Techniken bauen auf den Grundlagen von Weiser auf und basieren bspw. auf Programmabhängigkeitsgraphen. Ottenstein und Ottenstein [Ottenstein und Ottenstein 1984] stellten als Erste das *statische Slicing* mit Hilfe von PDGs vor.

Dabei werden sequentielle Programme wieder mit dem Prinzip *Erreichbarkeit* von Graphen mit Hilfe rückwärts gerichteter Traversierung, sog. *Backward-Slicing* über die PDG Kanten reduziert. Wie bereits in Abschnitt 3.4.3 beschrieben besteht ein PDG grundsätzlich aus den Knoten eines CFG (Anweisungen oder Prädikate) und den Kontroll- und Datenabhängigkeiten als Kanten.

Der Backward-Slice $S(n)$ eines PDG am Knoten n besteht letztlich aus allen Knoten, die von n transitive abhängig sind, wobei n das Slicing-Kriterium ist. Die Abbildung 3.16 bezieht sich erneut auf das Beispiel des PDGs vom Abschnitt 3.4.3 und setzt die Anweisung `write(mul)` als Kriterium für den Slice. Die Abbildung zeigt alle traversierten Knoten und somit einen Slice mit den Zeilen $S(11) = \{2, 3, 4, 5, 7, 8, 11\}$ und verdeutlicht, dass die Berechnung der Variable `sum` keinen Einfluss auf die Berechnung der Variable `mul` hat.

```
1  sum = 0
2  mul = 1
3  a = 1
4  b = read()
5  while (a <= b) {
6    sum = sum + a
7    mul = mul * a
8    a = a + 1
9  }
10 write(sum)
11 write(mul)
```

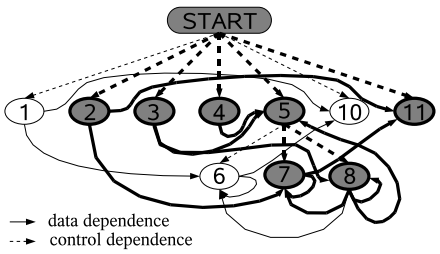

Abbildung 3.16.: Beispielprogramm und der Slice als PDG (Quelle: [Krinke 2003])

3.5.2. Interprozedurales Slicing mit SDG

Während vorherige Ansätze *interprozedurale* Anwendungsfälle abecken, beschäftigten sich Horwitz et al. [Horwitz u. a. 1988] mit der Erweiterung des auf PDG basierenden Algorithmus auf *interprozedurales Slicing*. Über *Systemabhängigkeitsgraphen* können Slices von kompletten Programmen ebenfalls prozedurübergreifend mit Hilfe eines Algorithmus generiert werden. Wie bereits in Abschnitt 3.4.6 beschrieben, stellen die PDGs die verschiedenen Unterprogramme dar, während der SDG die globalen Abhängigkeiten zwischen diesen, mit ihren interprozeduralen Kontroll- und

Datenflusskanten darstellt. Die grob erläuterten *Summary Edges* spielen beim *interprozeduralen Slicing* eine besondere Rolle, da diese Rückschlüsse über den transitiven Fluss von einem Parameter zum Rückgabewert einer aufgerufenden Funktion liefern kann, ohne diese Prozeduren/Funktionen selbst zu analysieren.

Die Traversierung beim interprozeduralen Slicing wird in zwei Phasen unterteilt, wobei in der ersten Phase vom Kriterium ausgehend einzig aufsteigend alle aufrufenden Prozeduren bestimmt werden (rückwärts über Parameter-In-Kanten) und in einer zweiten Phase für alle identifizierten Knoten aus Phase 1 alle aufgerufenen Prozeduren identifiziert werden (rückwärts über die Parameter-Out-Kanten). Für weiterführende detailliertere Ausführungen wird an dieser Stelle auf Literatur wie [Horwitz u. a. 1988] und [Koschke 2014] verwiesen.

3.6. Abgrenzung zur dynamischen Programmanalyse

Im vorherigen Kapitel wurde die Programmanalyse auf statischer Ebene und demzufolge auf den Quellcode bezogen beschrieben. Der Vollständigkeit halber muss erwähnt werden, dass auch dynamische Ansätze in der Praxis gängig sind, bei der das Programm zur Laufzeit analysiert wird. Dazu werden Programme mit konkreten Eingaben ausgeführt und während der Ausführung bspw. die Variablenwerte beobachtet. Dynamische Programmanalyse wird auf Grund dessen oftmals mit Testen und Debuggen in Verbindung gesetzt [Koschke 2014].

Intra- und interprozedurales Slicing wurde in dieser Arbeit bereits in seiner statischen Form vorgestellt, ist jedoch ebenfalls mit einem dynamischen Ansatz möglich [Korel und Laski 1988]. *Dynamisches Slicing* identifiziert ausschließlich Anweisungen innerhalb eines Programmausführungs-Traces (zu einem Slice Kriterium) und bringt zusätzlich Vorteile bei der Behandlung von Arrays und Pointer-Variablen. Hierbei werden im Gegensatz zum statischen Slicing alleinig die tatsächlich definierten und genutzten Variablen und Pointer einbezogen, anstatt etwa das ganze Array oder alle Pointer.

Dies führt dazu, dass *dynamische Slices* wesentlich kleiner sein können als statische und dementsprechend konkret sind, dafür schlechter verallgemeinerbar sind. Dieser Umstand sowie andere Vor- und Nachteile lassen sich generell auf statische und dynamische Programmanalyse übertragen.

Da in der vorliegenden Arbeit die Vorgehensweise der Sicherheits-Audits auf statischer Analyse basiert, bedarf es keiner weiteren Erläuterung *dynamischer Programmanalyse*.

© Philip Phu Dang Hoan Nguyen

4. Methodik

Das folgende Kapitel soll die methodischen Ansätze der vorliegenden Arbeit darlegen. Zunächst wird in Abschnitt 4.1 die generelle Vorgehensweise erläutert, die zum Erreichen des Forschungsziels gewählt wurde. Im weiteren Verlauf wird die empirische Methodik dieser Arbeit und das Vorgehensmodell zur Entwicklung des Auditors beschrieben. Anschließend werden vorhergehende Arbeiten des Technologie-Zentrum Informatik und Informationstechnik (TZI) vorgestellt und deren, für diese Arbeit relevanten, Erkenntnisse dargelegt.

4.1. Vorgehensweise

Das Ziel dieser Arbeit ist es, wie bereits in der Einleitung definiert, ein Auditor-Werkzeug zu entwickeln, welches später von Sicherheits-Analysten genutzt werden kann, um sicherheitsrelevante Programmteile aus Applikationen zu extrahieren und einfacher analysieren zu können.

Da in der wissenschaftlichen Einrichtung TZI der Universität Bremen unterschiedliche empirische Arbeiten zum Thema Programmanalyse mit Hilfe von Slicing geschrieben wurden, lag es nahe auf die bestehenden Technologien, Implementierungen und Erkenntnissen zurückzugreifen.

So zeigten Gulmann, mit der Masterthesis „Statische Sicherheitsanalyse der Android Systemservices“ im Jahr 2014 [Gulmann 2014] und Gerken [Gerken 2015] mit der Bachelorthesis „Statische Sicherheitsanalyse von Java Enterprise-Anwendungen mittels Program-Slicing“ im Jahr 2015 erste einfache Implementierungen (mit Hilfe des Frameworks WALA) zur statischen Programmanalyse im Kontext zu Android Systemdiensten und Java Enterprise-Anwendungen. Die weiterführende Arbeit „Evaluation des WALA-Slicers bzgl. der Anwendbarkeit auf sicherheitskritische Java-Programme“ von Detmers [Detmers 2016] im Folgejahr griff zudem offene Fragen und bereits bestehende Probleme wie bspw. effiziente Parametrisierung von WALA auf und evaluierte generell das Framework bezüglich der auf Anwendbarkeit auf sicherheitskritische Java-Programme.

Auch wenn im Bereich Programm-Slicing bereits über 30 Jahre Forschung betrieben wird, gibt es durch die Komplexität noch keine renommierten Programme auf dem Markt. Es ist allerdings bekannt, dass Slicing im Speziellen bei großen Programmen nicht sonderlich gut skaliert, sofern keine Optimierungen vorgenommen werden [Griswold 2001]. Daher bilden die Evaluationen und Erkenntnisse aus den verwandten Arbeiten eine wertvolle Grundlage für diese Arbeit, um zielführende Optimierungen für den Slicing-Prozess zu entwickeln.

Zunächst wurde auf Basis dieser Arbeiten eine einfache, ähnliche Implementierung nachkonstruiert, um die relevanten Resultate und Probleme nachzuvollziehen. Die Implementierung des finalen „Auditors“ begrenzt sich hierbei ausschließlich auf eine ausführbare Java-Anwendung ohne GUI, da diese im vorliegenden Anwendungsfall keinen Mehrwert hat. Zum einen sind die Ausgaben des Programmes Informationen zum Slicing-Prozess und eine Sammlung von Java Dateien, für die keine GUI benötigt wird. Die komplexen Einstellungsmöglichkeiten von WALA lassen sich zudem wesentlich einfacher mit einer strukturierten Konfigurationsdatei abbilden, welche im Abschnitt 5.3.3.3 erläutert ist. Zuletzt fiel die Entscheidung auch auf ein Kommandozeilen-Programm, da es die Möglichkeit offen hält, ein und dieselbe Applikation mit unterschiedlicher Parametrisierung automatisiert über Batchskripte laufen zu lassen. Die genaue Funktionsweise des Auditors wird im Abschnitt 5.2 dargestellt.

Anschließend wurden die bekannten WALA-Probleme vorheriger Arbeiten, die von Relevanz sind, mit Hilfe von verschiedenen kleineren Testfällen nachgestellt, analysiert und adressiert. Aus den Erkenntnissen vorheriger Arbeiten entstand letztendlich eine Teilmenge der Anforderungen an den Auditor (weitere detaillierte Erläuterungen zum Entwicklungsverfahren folgen im nächsten Abschnitt). Der gesamte Anforderungskatalog, woraus die implementierten Optimierungen entstanden sind, wird im Abschnitt 5.1 beschrieben.

Nach Abschluss aller Optimierungen wurden zusätzliche Testfälle aufgestellt, welche alle Anpassungen validieren sollten. Abschließend wurde der Auditor auf größere Java-Anwendungen angewendet, um dessen Verhalten zu evaluieren und ggf. Skalierbarkeit gewährleisten zu können. Der durchgeführte Benchmark soll hiermit gleichzeitig die Grenzen des Auditors aufzeigen und somit Einschränkungen der Benutzung dokumentieren.

© Philip Phu Dang Hoan Nguyen

Empirische Methodik

Die Vorgehensweise lässt sich wissenschaftlich einem empirischen Ansatz zuordnen, welcher sich unter anderem der qualitativen deskriptiven Forschungsmethode der *Fallstudien* bedient. Diese in der Softwareforschung häufig genutzte Methodik unterstützt nicht nur prinzipielle Funktionsfähigkeiten eines Werkzeugs zu zeigen, sondern hilft auch Vorkommnisse oder bestimmte Phänomene (hier falsches oder unzureichendes Verhalten beim Slicen) zu erfassen, verstehen und zu evaluieren [Padberg und Tichy 2007].

Dies unterstützte zusätzlich das Vorgehen bezüglich der Software-Entwicklung, in der iterativ aus Erkenntnissen bestehender Probleme, Anforderungen an den Auditor abgeleitet wurden. Zusätzlich steht, mit Hilfe der bestehenden empirischen Arbeiten, in verkürzter Zeit ein lauffähiges Software-Grundgerüst zur Verfügung, welches eine Grundlage für die Fallstudien bildet.

Vorgehensmodell zur Software-Entwicklung

Bei der Software-Entwicklung selbst wurde ein agiles *inkrementelles iteratives Verfahren* [Cockburn 2008] gewählt, da dies für die Forschungsmethode und Entwicklungsweise am zweckentsprechendsten war. Modelle wie das Wasserfall-Modell sind hierbei zu starr, zumal die Anforderungen am Anfang der Evaluation nicht deutlich waren oder das Spiralmodell beispielsweise zu groß und detailliert bezüglich der Zyklen.

Inkrementell heißt in diesem Kontext, dass die Entwicklung zwar zeitlich und auch eventuell funktional getrennt ist, aber aufeinander aufbauend ist und somit bei Fertigstellung zusammengeführt wird. *Iterative* Entwicklung verfolgt eine Strategie der Überarbeitungsplanung, in welcher das aktuelle System überprüft und weitere Verbesserungen geplant werden. In der gemischten Form ist es somit möglich, schrittweise inkrementell ein System zu entwickeln, das direkt getestet und analysiert werden kann, ohne dass alle Systemanforderungen (in diesem Falle Optimierungen) in frühen Phasen bekannt sind. Neue Erkenntnisse, Einschränkungen und Fehler können in die Planung des nächsten Iterationsschrittes einfließen und weitere Überarbeitungen bzw. Optimierungen einleiten.

Folglich werden die relevanten Arbeiten analysiert und in der initialen Phase in eine einfache und ausbaufähige Implementierung und eine Untermenge der zu diesem Zeitpunkt bekannten Systemanforderungen gewandelt. Mit Hilfe von verschiedenen neuen Testfällen, ergeben sich schrittweise weitere Anforderungen, welche implementiert, getestet und für die weitere Evaluierung des Auditors genutzt werden. Die

Iterationen können dadurch jederzeit in einer finalen Bereitstellung (engl. Deployment) enden. Der beschriebene Entwicklungsablauf wird bildlich in Abbildung 4.1 dargestellt.

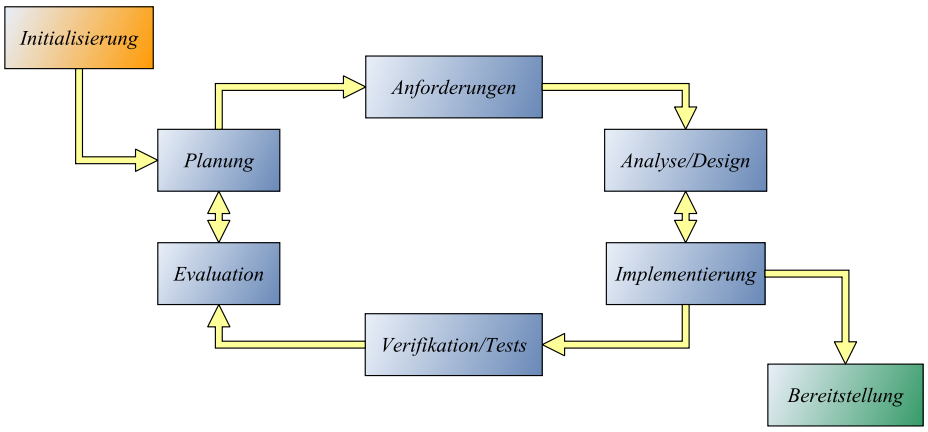

Abbildung 4.1.: Inkrementelles iteratives Entwicklungsmodell

Mit jeder Iteration wird das Werkzeug somit nicht nur optimiert und verbessert, sondern auch mit Testfällen validiert, woraus schlussendlich ein einsatztauglicher Auditor resultiert.

4.2. Bezug zu bestehenden Forschungsergebnissen

4.2.1. Adaption Implementierung und Testfälle

Implementierung

In Bezug auf Implementierungen wurden einfache grundlegende Slicing-Funktionalitäten (mit der Bibliothek von WALA) aus den Arbeiten von Gerken und Detmers [Gerken 2015; Detmers 2016] nachvollzogen und übernommen. Die Umsetzung dieser Arbeiten war in keinem der Fälle ein fertiges Werkzeug, sondern eher, wie sie es beschreiben, prototypische Implementierung als Machbarkeitsbeweis. Viele Slicing-Abläufe ähneln daher den offiziellen WALA-Beispielen und halten damit noch Möglichkeiten der Optimierung offen. Die einzige Ausnahme bildet die Generierung der Einstiegspunkte, welche auf unterschiedliche Weise implementiert werden kann und im Abschnitt 5.2.2 genauer erläutert wird. Ferner wurde der von Detmers implementierte Java Parser übernommen, welcher auf einer Implementierung von Gulmann basiert und nach Berechnung der Slice-Zeilen den entsprechenden Java-Quelltext rekonstruieren kann. Im Laufe der Arbeit zeigten sich weitere Schwächen in der

 © Philip Phu Dang Hoan Nguyen

Funktionsweise des Parsers auf, weswegen diverse Optimierungen als Anforderungen aufgenommen und durchgeführt wurden (vgl. Abschnitt 5.3.3.2).

Testfälle

Da die vorliegende Arbeit Sicherheitsanalysen auf Java-Applikationen durchführen soll, lag es nahe, sicherheitsrelevante Testfälle von vorherigen Arbeiten zu übernehmen und zusätzlich weiter zu analysieren. Gerken und Detmers greifen eine Anwendung in Form einer Enterprise Java Bean (EJB) auf, welche beispielhaft Geschäftslogiken in einem fiktiven Klinik-Informationssystem darstellen soll. EJB beschreibt eine moderne verteilte Komponentenarchitektur in einer *Java Enterprise Edition (Java EE)* Client-Server-Konfiguration. Mittels dieser ist es u.a. möglich komponentenbasiert zu programmieren, Objekte über sogenannte Entity-Beans persistent an eine Entität in einer Datenbank zu binden und z.B. Geschäftsmethoden für Transaktions- und Sicherheitsdienste umzusetzen.

Die Klinik-Anwendung bietet somit nicht nur Operationen an, um bspw. rollenbasiert Patientenakten einzusehen, Rezepte zu lesen, auszustellen oder Diagnosen zu senden, sondern wendet ebenfalls Verschlüsselungsfunktionen und Signaturen sinngemäß an. Detaillierte Beschreibungen zu der Testanwendung wie bspw. die Sicherheits-Anforderungen, sind aus dem Paper „Towards Security Program Comprehension with Design by Contract and Slicing“ [Sohr u. a. 2015a] oder aus der detaillierten Evaluation in der Bachelorarbeit von Gerken [Gerken 2015] zu entnehmen.

4.2.2. Relevante Erkenntnisse

Aus der Arbeit von Gerken [Gerken 2015] geht zunächst hervor, dass es möglich ist, mit Hilfe von WALA, ein Werkzeug zu entwickeln, welches ein Slicing nach bestimmten, sicherheitsrelevanten Methodenaufrufen durchführen kann. Unterstrichen wird diese Erkenntnis u.a. mit unterschiedlichen Testfällen zu der zuvor beschriebenen EJB-Anwendung im Kontext Zugriffsberechtigung und Kryptografie (Digitale Signatur, SSL/TSL und symmetrische Verschlüsselung), wobei die Anwendung mit grob 250 Zeilen im Vergleich zu richtigen Anwendungen noch klein ist, aber eine große Menge an Fremdbibliotheken nutzt.

Bei der Durchführung der Testfälle stellt Gerken in seiner Arbeit fest, dass das Slicing ohne Berücksichtigung der Datenabhängigkeiten wichtige Aufrufe nicht berücksichtigt, jedoch mit explizit aktivierten Datenabhängigkeiten, selbst bei kleinen Anwendungen, zu speicher- und rechenintensiv wird. Dies führte zu Berechnungszeiten von

über 45 Minuten oder im schlechtesten Fall dazu, dass der Arbeitsspeicher volllief und das Programm vorzeitig terminierte. Grund ist ein zu schwach definiertes *Exclusionsfile*, welches bei WALA als Konfigurationsparameter mitgegeben wird und definiert, welche Applikationen und Bibliotheken (sog. Analysescope Abschnitt 5.2.2) in der Analyse einbezogen werden sollen. Gerken optimiert den Slicing-Vorgang deutlich, indem er sämtliche Pakete und Klassen der Java-Standardbibliothek mit einem detaillierten *Exclusionsfile* ausschließt, welches auch für die Umsetzung des Auditors dieser Arbeit benutzt wird.

Ferner tritt, so Gerken, beim Testen ein unerklärliches Verhalten auf, bei dem nicht relevante arithmetrische Operationen im Slice auftauchen, die keinerlei Abhängigkeit zu dem Slice Kriterium haben. Dieses Problem wird im Abschnitt 5.2.4 beschrieben und aufgeschlüsselt.

Aus der Arbeit von Detmers [Detmers 2016] ergeben sich ebenfalls wichtige Erkenntnisse, die wichtige Grundlagen für die Entwicklung des Auditors bilden. Zunächst evaluiert Detmers im ersten Teil seiner Arbeit seine eigene Slicing-Applikation mit Hilfe verschiedener Testfälle um zu prüfen, ob WALA alle gängigen Konzepte der objektorientierten Sprache Java beim Slicen abdeckt. Überprüft werden dabei unter anderem unterschiedliche Objekte, primitive Datentypen, Feldzugriffe, Vererbung/Überschreiben, Exceptions, Konstruktoren und Verzweigungsanweisungen wie `switch-case, if-then-else, for-, while` und `do-while`-Schleifen. Zusätzlich werden bei den Testfällen alle möglichen Kombinationen bei der Wahl der Kontroll- und Datenabhängigkeitsoptionen angewendet und evaluiert. Daraus leiteten sich nicht nur Erkenntnisse ab, welche Auswirkungen bestimmte Optionen haben, sondern ebenfalls, welche Optionen für zukünftige Anwendungen interessant sind und welche keinen Nutzen haben. Die genaue Funktionsweise von WALA, mit Erläuterungen bezüglich der Parametrisierung und der Kontroll- und Datenabhängigkeitsoptionen, werden später im Abschnitt 5.2.3 genauer erläutert. Unter anderem werden zusätzlich die, aus der Evaluation dazu gewonnenen Erkenntnisse dargelegt.

Der zweite Teil der Evaluation von Detmers behandelt ebenfalls die EJB-Anwendung der Arbeit von Gerken, wendet jedoch nur noch vier verschiedene Kombinationen der Kontroll- und Datenabhängigkeitsoptionen von WALA an. Aus der Analyse ergibt sich unter anderem, dass für viele Anwendungen in Bezug auf Datenabhängigkeit statt der detailliertesten Datenabhängigkeitsoptionen `FULL` auch `NO_BASE_PTRS` und `NO_EXCEPTIONS` angewendet werden kann, um die Größe des SDG zu verringern. Für weiterführende Untersuchungen ist zu berücksichtigen, dass in Detmers' Arbeit nicht eindeutig geklärt werden konnte, wie sich die Kontrollabhängigkeitsoption `NO_EXCEPTIONAL_EDGES` genau verhält. Im späteren Abschnitt 5.2.4 wird erläutert,

 © Philip Phu Dang Hoan Nguyen

welche Auswirkungen die Option genau hat, und letzten Endes auch, welche Zusammenhänge diese Option mit den bisher ungeklärten Seiteneffekten beim Slicen hat.

Ferner bemängelt Detmers ebenfalls wie Gerken, dass bei einigen Testfällen unerwünschte Objekte in den Slice kommen, was ebenfalls in der vorliegenden Arbeit behandelt wird.

5. Konzept und Implementierung

In diesem Kapitel wird mit den Erläuterungen zu dem Konzept und der Implementierung auf den praktischen Teil dieser Masterarbeit eingegangen. Zunächst wird im Abschnitt 5.1 erläutert, wie aus der Idee und den bereits umrissenen Zielen konkrete Anforderungen definiert wurden. Im Folgenden wird in Abschnitt 5.2 detailliert beschrieben, wie der Auditor technisch mit Hilfe der Analysetechnik „Slicing“ eine Sicherheitsanalyse unterstützt. Dazu gehören technische Beschreibungen, zum einen zum Algorithmus (Abschnitt 5.2.2) und zum anderen zu den verschiedenen Parametrisierungsmöglichkeiten (Slicing Optionen im Abschnitt 5.2.3 und andere Konfigurationen im Abschnitt 5.3.3.3). Im zweiten Teil des Kapitels wird Bezug auf die Implementierung (Abschnitt 5.3) des Auditors genommen. Neben der Systemarchitektur des Auditors, werden alle genutzten Bibliotheken und Werkzeuge erläutert, die für die Entwicklung relevant waren. Abschließend werden im Abschnitt 5.3.3 ausgewählte Optimierungen bezüglich der Implementierung noch im Detail beschrieben.

5.1. Anforderungen

Die Zielsetzung (in Abschnitt 4.1 bereits aufgeführt), ein Auditor-Werkzeug zu entwickeln, welches mit Hilfe von Programm-Slicing Sicherheits-Analysten beim Programmverständnis helfen soll, ist zunächst nur grob festgesteckt. Die grundlegende Idee besteht darin, ein Werkzeug bereitzustellen, welches ohne großen Installations- und zudem zusätzlichen Implementierungsaufwand zum Slicen von Java-Anwendungen genutzt werden kann. Aufgrund der reichen Varianz beim Slicen, sei es wegen unterschiedlicher Komplexitätsgrade der Programme oder wegen des unterschiedlichen Bedarfs bezüglich der Tiefe oder Genauigkeit der Analysen, war es nicht möglich, das Werkzeug grundsätzlich für alle Anwendungszwecke optimiert zu implementieren. Stattdessen wird die Anforderung gestellt, dass dem Nutzer die Möglichkeit gegeben soll, das Werkzeug für jeden Anwendungsfall passend zu parametrisieren. Über die verschiedenen Konfigurationsmöglichkeiten ist es somit möglich, nach Bedarf, das Werkzeug derart zu justieren, dass der Slice die wichtigsten Informationen aus dem originalen Quellcode extrahiert. Mit steigender Größe und Komplexität eines zu analysierenden Programmes, ist es in einigen Fällen umso wichtiger, dass der

Slicer konfiguriert werden kann, dass die Ergebnisse zwar weniger detailliert/präzise sind, im Tausch jedoch die rechnerischen Kapazitäten nicht übersteigt.

Eine weitere Anforderung, dass der Auditor als Kommandozeilen-Programm genutzt wird, wurde ebenfalls bereits in Abschnitt 4.1 damit gerechtfertigt, dass dies die mehrfache Anwendung mit unterschiedlicher Parametrisierung zulässt. Dies geht ebenfalls mit der Anforderung einher, dass die Analyseergebnisse und somit die Slices und optional die Informationen zur Pointeranalyse, dem Callgraph oder dem SDG (in Textform oder in Graphenform als PDF) in separate Ausgabeordner gesichert werden müssen. Andere Implementierungen der vorhergehenden Arbeiten exportieren die Slices im Vergleich nicht, sondern markieren die Zeilen im Originalquelltext (bspw. in einem Graphical User Interface (GUI)).

Andere Anforderungen entstanden, wie in Abschnitt 4.1 beschrieben, aus einem inkrementellen iterativen Entwicklungsansatz sukzessiv, wurden daraufhin implementiert und durch Testfälle validiert.

5.1.1. Anforderungskatalog

Im folgenden Abschnitt sollen alle funktionalen und nichtfunktionalen Anforderungen in einem Katalog aufgestellt und erläutert werden. Die Anforderungen werden im Rahmen dieser Masterarbeit zwar an der empfohlenen Anforderungsspezifikation nach *ISO/IEC/IEEE 29148* [IEEE 2011] angelehnt definiert, beschränken sich jedoch nur grob auf die Produktfunktionen (funktionalen Anforderungen), Qualitätsanforderungen (nichtfunktionalen Anforderungen) und die Einschränkungen. Von einer vollständigen Spezifikation wird in dieser Arbeit abgesehen, da mit Fertigstellung dieser Arbeit nicht gewährleistet werden kann, dass die Implementierung ein finales Produkt ergibt und zum anderen dies auch den Rahmen der Arbeit übersteigen würde.

5.1.1.1. Funktionale Anforderungen

Die funktionalen Anforderungen werden zunächst in Systemfunktionen (engl. system features) gruppiert und mit einer kurzen eindeutigen ID versehen, welche aus dem Kenner „SF“ und einer fortlaufenden Nummer besteht. Neben einer Beschreibung der Systemfunktion und ggf. des Ursprungs der Anforderung werden darüber hinaus die Priorität, die betroffene Rolle und Testfälle ergänzt, die im Zusammenhang zu dieser Funktion stehen und im Kapitel 6 näher erläutert werden. Die Prioritäten

 © Philip Phu Dang Hoan Nguyen

werden unterteilt in „Muss-Anforderungen“, „Soll-Anforderungen“ und „Wunsch-Anforderungen“, welches bei der inkrementellen Entwicklung nützlich für die Festlegung des Inhalts und Umfangs einer Iteration ist. Bei der Rolle wird zwischen dem *System* und dem *Benutzer* unterschieden, wobei Anforderungen mit der Rolle *System* automatisch vom Auditor durchgeführt werden müssen.

Zuletzt werden für jede Systemfunktion alle funktionalen Anforderungen aufgelistet, die vom Auditor gewährleistet werden sollen. Zum besseren Verständnis einiger Testfälle bezüglich der Konfiguration/Parametrisierung des Auditors empfiehlt sich ggf. der Querverweis zum Abschnitt 5.2, welcher nach diesem Abschnitt folgt.

SF01: Internationalisierung

Beschreibung:	Englische Programmausgaben/Quellcode sind erforderlich, da im IT-Forschungsumfeld Englisch die dominierende Sprache ist und in Anbetracht einer potenziellen Weiterentwicklung sinnvoll ist.
Priorität:	Muss
Rolle:	System
Testfälle:	-

Funktionale Anforderungen:

- Hauptsprache des Auditors soll Englisch sein. Dies betrifft die Programmausgaben sowie den Quellcode mit seinen englischen Bezeichnernamen (Klassen, Methoden, Variablen und Kommentaren).

SF02: Analysedurchführung

Beschreibung:	Die korrekte Durchführung der Analyse muss gewährleistet werden.
Priorität:	Muss
Rolle:	Benutzer
Testfälle:	Alle

Funktionale Anforderungen:

- Mit Ausführung des Auditors soll mit Hilfe einer gültigen Konfigurations- und Exclusiondatei die Analyse durchgeführt werden und die Ergebnisse korrekt in einem Zielverzeichnis abgelegt werden.

- Während der Ausführung der Analyse sollen Informationen zu den Zwischenschritten (Entrypoints-Ermittlung, Caller/Callee-Suche, Backward-Slice, Quelltext-Rekonstruktion) in der Kommandozeile für den Benutzer ausgegeben werden.

SF03: Konfigurationsdatei

Beschreibung:	Eine einfache Justierung der Konfiguration über eine Konfigurationsdatei ist zwecks Übersichtlichkeit und Mehrfachverwendung (etwa mit einem Batchscript) vorteilhaft.
Priorität:	Muss
Rolle:	Benutzer/System
Testfälle:	Alle

Funktionale Anforderungen:

- Der Benutzer soll die komplette Konfiguration und Parametrisierung über eine Konfigurationsdatei (Textformat) steuern können.
- Die Auswertung der Konfigurationsdatei vom System soll sinnvolle Defaulteinstellungen haben und bei falscher Definition angemessene Fehlermeldungen ausgeben.

SF04: Einschränkung Scope

Beschreibung:	Eine Einschränkung des Analyse-Scopes (Analyseumfang) ist beim Slicing unverzichtbar, da eine Analyse mit Einbeziehen aller genutzter Java Bibliotheken kaum effizient möglich ist. Ein Vorgehen mit zunächst detaillierter Exclusionsliste und schrittweiser Annäherung ist gängig und erforderlich (vgl. [Gerken 2015]).
Priorität:	Muss
Rolle:	Benutzer
Testfälle:	Alle

Funktionale Anforderungen:

- Der Benutzer soll vor der Analyse die Möglichkeit haben, Java-Klassen und -Pakete aus dem Analyse-Scope auszuschließen.

SF05: Konfiguration Entry Points

Beschreibung:	Eine Beeinflussung in der Ermittlung der Programm-Einstiegspunkte wirkt sich ebenfalls auf den Analyse-Scope aus und kann die Analyse entweder grober fassen, aber dafür ineffizienter machen oder das Gegenteil bewirken.
Priorität:	Soll
Rolle:	Benutzer/System
Testfälle:	Alle, speziell TF10 und TF11

Funktionale Anforderungen:

- Der Benutzer soll vor der Analyse verschiedene Möglichkeiten haben, mit Angabe einer Hauptklasse die Ermittlung der Programm-Einstiegspunkte zu beeinflussen.

© Philip Phu Dang Hoan Nguyen

- Das System soll abhängig davon, ob der Benutzer eine explizite Hauptklasse angibt, verschiedene Methoden durchprobieren, um Einstiegspunkte für die Analyse zu finden, ansonsten diese Angabe ignorieren und global alle Einstiegspunkte ermitteln.

SF06: Auswahl Slicing-Kriterien

Beschreibung:	Mit der Auswahl der Caller und Callee Methode werden Slicing-Kriterien definiert.
Priorität:	Muss
Rolle:	Benutzer
Testfälle:	Alle

Funktionale Anforderungen:

- Der Benutzer soll die Analyse-Kriterien mit Angabe der Caller Methode und der Callee Methode in der Konfigurationsdatei bestimmen können.

SF07: Erweiterung Slicing-Kriterien - Multiple Callers

Beschreibung:	Da bei der Analyse der Anwendung der Name der Caller Methode nicht zwangsläufig eindeutig ist (z.B. wegen überladender oder gleich-heißender Methoden in unterschiedlichen Klassen) ist ein Konfigurationsschalter notwendig, der entscheidet, ob nur der erste gefundene oder alle Caller in die Analyse einbezogen werden sollen.
Priorität:	Muss
Rolle:	Benutzer
Testfälle:	TF03

Funktionale Anforderungen:

- Der Benutzer soll durch das Setzen eines Konfigurationsschalters zusätzlich spezifizieren können, ob mehrfach gefundene Caller in die Analyse einbezogen werden sollen.

SF08: Erweiterung Slicing-Kriterien - Multiple Callee und Callee Klasse

Beschreibung:	Analog zu SF07 ist bei der Analyse der Name der Callee Methode nicht eindeutig, da dieser ebenso in jeder anderen Klasse existieren kann. Daher wurde die Anforderung gestellt, dass zum einen mehrere Calles in das Slicing einfließen und über eine optionale Angabe der passenden Callee Klasse die Callees eingeschränkt werden können.
Priorität:	Muss
Rolle:	Benutzer/System
Testfälle:	TF02

Funktionale Anforderungen:

- Das System soll mehrfache Callee-Aufrufe erkennen und bearbeiten können.

- Der Benutzer kann den Callee über die zusätzliche Angabe einer Callee Klasse spezifizieren.
- Der Benutzer kann die Callee Klasse entweder mit oder ohne vollständigen Paketpfad angeben.

SF09: Auswahl Slicing Optionen

Beschreibung:	Die Tiefe und somit Komplexität des Slicings lässt sich bei WALA anhand unterschiedlicher Datenabhängigkeits- und Kontrollabhängigkeitsoptionen steuern (genaue Erläuterungen in Abschnitt 5.2).
Priorität:	Muss
Rolle:	Benutzer
Testfälle:	Alle

Funktionale Anforderungen:

- Der Benutzer soll vor der Analyse aus einer vordefinierten Liste in der Konfigurationsdatei eine Datenabhängigkeitsoption und Kontrollabhängigkeitsoption festlegen können.

SF10 Rekonstruktion Quelltext

Beschreibung:	Nach dem Slicen muss aus den berechneten Statements des Slices der lesbare Quelltext rekonstruiert werden. Da WALA bei der Zurückführung von der Binärform nur einzelne Programmenzeilen angibt, muss zum einen der Originalquelltext einbezogen werden und zum anderen die zugehörigen Klassen-, Methoden- und Kontrollanweisungsköpfe und -rümpfe ermittelt werden. Je nach Komplexität der Anwendungen und unterschiedlichem Programmierstil (Coding-Konventionen), ergaben sich beim korrekten Rekonstruieren unterschiedliche Probleme, welche in Abschnitt 5.3.3.2 behandelt werden.
Priorität:	Soll
Rolle:	System
Testfälle:	TF08, TF09, TF10, TF11

Funktionale Anforderungen:

- Der Auditor soll aus den errechneten Slices die ursprünglichen, aber reduzierten Quelltexte rekonstruieren und in Form einer Java-Datei ablegen.
- Beim Rekonstruieren sollen einzelne Slices vom System um ihre umgebenden Anweisungen ergänzt werden (inklusive Klassen-, Methoden- und Anweisungsköpfe und -rümpfe).
- Der Auditor soll beim Rekonstruieren des Quelltextes Java Konstrukte wie *if-else, while, for*, aber auch Konzepte wie *Konstruktoren* oder *Exceptions* beherrschen.
- Der Auditor soll ebenfalls Quelltexte mit unterschiedlichen Programmierstilen rekonstruieren können.
- Die rekonstruierten Java Dateien sollen möglichst syntaktisch, aber in jedem Fall semantisch vollständig korrekt sein.

- Der Auditor soll möglichst nur Programmstellen im Slice haben, welche dem Slice-Kriterium und den gewählten Einstellungen entsprechen.

SF11 Rekonstruktion Quelltext - Optionen

Beschreibung:	Zum Rekonstruieren des Quelltextes werden die originalen Java-Dateien benötigt. Bei mehrfachem Gebrauch des Auditors (bspw. über ein Batchscript) ist es ansonsten sinnvoll, für jede Ausführung unterschiedliche Ausgabepfade zu definieren. Zusätzlich ist es nützlich, die Slicing-Ergebnisse anstatt in einer Java-Datei auf verschiedene Dateien (mit ursprünglichen Quellnamen) zu verteilen.
Priorität:	Muss
Rolle:	Benutzer/System
Testfälle:	Alle

Funktionale Anforderungen:

- Der Benutzer soll über die Konfigurationsdatei einen Suchpfad angeben können, in dem sich der originale Quelltext zur analysierenden Anwendung befindet.
- Das System soll bei Angabe des Suchpfades während der Analyse diesen rekursiv nach den benötigten Java-Quelldateien durchsuchen und ansonsten im Pfad des Auditors suchen.
- Der Benutzer soll über die Konfigurationsdatei einen Pfad angeben können, in dem die Ergebnisse der Analyse gespeichert werden.
- Der Benutzer soll über einen Schalter in der Konfigurationsdatei angeben können, ob die rekonstruierten Slices in eine oder mehrere Java-Dateien entsprechend dem Quellnamen geschrieben werden sollen.

SF12 Generierung PDFs

Beschreibung:	Bei der Analyse soll der Auditor als Zwischenprodukt ebenfalls den SDG, den Callgraph und die Zwischendarstellungen im Graphenformat berechnen und bereitstellen können. Diese Ergebnisse können einem Sicherheitsanalysten u.a. für das Programmverständnis oder ebenso für eine detailliertere Bestimmung der Parametrisierung nützlich sein. Da die Erzeugung der Graphen mit steigender Programmkomplexität die Ausführungszeiten und Rechenkapazitäten überschreiten kann, muss der Benutzer die Erzeugung ggf. deaktivieren können.
Priorität:	Soll
Rolle:	Benutzer
Testfälle:	TF02

Funktionale Anforderungen:

- Der Benutzer soll über einen Schalter in der Konfigurationsdatei die Erzeugung des SDG, des Callgraph und der Zwischendarstellungen im Graphenformat aktivieren können, welche im PDF-Format in den Ausgabeordner gesichert werden.

- Da bei der Erzeugung des SDGs im PDF-Format bei komplexeren Programmen der Speicher überlaufen kann, soll der Benutzer für diesen Graphentyp über einen separaten Schalter die Generierung aktivieren und deaktivieren können.

SF13 Erweiterung Slicing - Callee-Objektverfolgung

Beschreibung:	Über die Datenabhängigkeits- und Kontrollabhängigkeitsoptionen gibt es zwar die Möglichkeit, von einem Slice-Kriterium bzw. vom Callee alle abhängigen Anweisungen nachzuverfolgen, doch funktioniert dies einzig auf Ebene der Variablen selbst durchgängig zuverlässig. Aus den Testfällen zu der EJB-Anwendung ergaben sich Erkenntnisse, dass grundsätzlich Methodenaufrufe auf dem Objekt des Calles nicht in den Slice gelangen, obwohl diese sehr relevant für den Callee-Aufruf sind (Vgl. Testfall TF06 und TF07 in Abschnitt 6.2.6). Zusätzlich ist die Instanzierung des Objekts ebenfalls für die Analyse von Bedeutung, da dort ggf. bereits Parameter für den Konstruktor erkennbar sind.
Priorität:	Soll
Rolle:	Benutzer
Testfälle:	TF05, TF06, TF07, TF10

Funktionale Anforderungen:

- Der Benutzer soll über einen Schalter in der Konfigurationsdatei aktivieren können, dass zu einer Callee-Methode zum einen das Objekt erkannt wird und zum anderen, dass seine Instanziierung und alle darauf angewendeten Methodenaufrufe in den Slice einbezogen werden.

SF14 Erweiterung Slicing - Individuelle Instanziierungsmethoden

Beschreibung:	Bei der Systemfunktion SF13 muss die Instanziierung des Objekts für die Callee-Objektverfolgung erkannt werden. In einer ersten Implementierung konnten New-Operatoren in der Zwischendarstellung SSA eindeutig identifiziert und weiterverarbeitet werden. Leider nutzen bspw. kryptografische Bibliotheken zum Instanziieren/Erzeugen der Objekte eigene Methoden wie bspw. `Cipher.getInstance()`. Daher ist es notwendig dem Auditor solche Methoden mitgeben zu können.
Priorität:	Soll
Rolle:	Benutzer
Testfälle:	TF06, TF07, TF10

Funktionale Anforderungen:

- Der Benutzer soll in der Konfigurationsdatei des Auditors eine Liste von Methoden mitgeben können, welche eine Instanziierung des zu analysierenden Objektes bewirkt und somit in den Slice gelangen soll.

 © Philip Phu Dang Hoan Nguyen

5.1.1.2. Nichtfunktionale Anforderungen

Im Folgenden werden andere, nichtfunktionale Anforderungen (auch Qualitätsanforderungen genannt) an den Auditor aufgelistet und kurz erläutert.

Benutzbarkeit

Zwar bietet ein Kommandozeilen-Programm generell begrenzte Möglichkeiten, was die Bedienbarkeit angeht, jedoch soll der Benutzer zunächst mit Hilfe einer wohldefinierten Konfigurationsdatei in der Lage sein alle Parameter zu verstehen und korrekt anzuwenden. Bei der Programmausführung sollte die Analyse genügend Informationen zu den Zwischenschritten bereitstellen, um nicht nur die Parametrisierung zu evaluieren, sondern auch um einen gewissen Fortschritt zu erkennen. Nach Abschluss der Analyse soll daneben bei Erfolg eine Erfolgsmeldung und bei Fehlern sprechende Fehlermeldungen ausgegeben werden.

Leistung / Performance

Bezüglich der Leistung des Auditors, soll dieser nach Möglichkeit ressourcensparend und effektiv implementiert sein. Um das Analyseverhalten nach Komplexität der zu analysierenden Anwendung den verfügbaren Ressourcen anzupassen, wird eine große Auswahl an Konfigurationsmöglichkeiten angeboten.

Skalierbarkeit

Das Anbieten der verschiedenen Konfigurationsmöglichkeiten soll in einem gewissen Maße auch die Skalierbarkeit der Anwendung gewährleisten. Im besten Fall kann bei großen komplexen Programmen die Genauigkeit des Slices mit richtiger Parametrisierung dahingehend reduziert werden, dass sich noch nutzbare Ergebnisse ergeben. Dadurch ist der Trade-off zwischen Genauigkeit und Skalierbarkeit für den Anwender selbständig justierbar.

Wartbarkeit / Erweiterbarkeit

Da im Rahmen dieser Masterarbeit nicht ausgeschlossen werden kann, dass das resultierende Werkzeug keinem finalen Zustand entspricht, ist in jedem Fall denkbar, dass der Funktionsumfang in Zukunft ausgedehnt werden soll. Mit Vorbedacht der Wartbarkeit und Erweiterbarkeit, soll bei der Implementierung englischer (siehe *SF01:*

Internationalisierung), möglichst dokumentierter (mit Kommentaren) und modular gegliederter Quellcode berücksichtigt werden.

Korrektheit

Wie bereits in *SF02 Analysedurchführung* und *SF11 Rekonstruktion Quelltext - Optionen* erwähnt, sollen die Analyseergebnisse nicht nur möglichst sinngemäß, sondern auch bei der Quelltext-Rekonstruktion syntaktisch korrekt sein.

5.1.1.3. Einschränkungen

Neben den aufgeführten Anforderungen müssen jedoch auch einige Einschränkungen hingenommen werden, welche im Folgenden kurz beschrieben werden.

Benutzbarkeit

Obwohl die Konfigurationsmöglichkeiten gut dokumentiert sind, verbleibt der Auditor ein Werkzeug für Sicherheitsanalysten, weswegen die Benutzbarkeit für Laien nur eingeschränkt gewährleistet werden kann. Verwendungsfähige Ergebnisse lassen sich nur durch sinnvolle Parametrisierung erzielen und setzen damit auch ein gewisses Verständnis für die statische Analyse und das zu analysierenden Programm voraus.

Korrektheit

Auch bei den Anforderungen der Korrektheit an den Auditor müssen gewisse Einschränkungen hingenommen werden. Zunächst nutzt die Anwendung für den Slicing-Prozess hauptsächlich die Bibliothek WALA, welches im Bereich der statischen Analysen bekannt ist und laut Detmers ebenfalls funktional für die Analyse von sicherheitskritischen Java-Anwendungen geeignet ist [Detmers 2016]. Die Tatsache, dass die statische Programmanalyse noch aktiv erforscht wird und WALA kontinuierlich weiterentwickelt wird, lassen erahnen, dass eine absolute Korrektheit nicht gewährleistet werden kann. Bereits bekannte, als auch noch nicht gefundene Schwächen oder Probleme von WALA können im Rahmen dieser Arbeit nicht in der Bibliothek selbst adressiert werden.

Generell stellt sich die statische Analyse von der Berechnung her in jedem Fall einem unentscheidbaren Problem, dem Halteproblem, weswegen es ohnehin keine hundertprozentig genaue Lösung geben kann. Das Halteproblem beschreibt, dass

 © Philip Phu Dang Hoan Nguyen

grundsätzlich nicht entschieden werden kann, ob ein Algorithmus terminiert und dementsprechend einen finalen Zustand erreicht. Um dies eindeutig bestimmen zu können, muss das Programm ausgeführt werden [Chess und West 2007].

Chess und West fügen ferner hinzu, dass durch diese Problematik statische Analysen in jedem Fall *false positives* oder *false negatives* erzeugen. *False positives*, ebenfalls bekannt als *falscher Alarm*, sind im Zusammenhang zum Slicing bspw. Anweisungen, die keine Abhängigkeiten zum Slice-Kriterium haben, jedoch ungeachtet dessen im Slice erscheinen. Ein *false negative* ist der umgekehrte und weniger tolerierbare Fall, in dem eine Abhängigkeit besteht (demzufolge wichtig für die Analyse ist), allerdings nicht im Ergebnis auftaucht.

Skalierbarkeit

Auch wenn Skalierbarkeit zu den Qualitätsanforderungen gezählt wird, ist zu erwarten, dass diese beim Programm-Slicing an Grenzen stoßen wird. Daher lassen sich über die verschiedenen Einstellungsmöglichkeiten zwar ebenfalls für größere Programme Slices erzeugen, doch können durch den Trade-off ebenso unbrauchbare ungenaue Slices resultieren.

5.2. Analysetechnik

In diesem Abschnitt wird die Analysetechnik beschrieben, die mit Hilfe vom Programm-Slicing Sicherheitsanalysen unterstützen soll, sicherheitsrelevante Programmteile zu extrahieren und analysieren. Von der Grundidee her soll dies über die Erreichbarkeit eines Slice-Kriteriums in einem SDG realisiert werden (vgl. Kapitel 3.5). Backward-Slicing ermittelt alle transitiv abhängigen Knoten und somit auch alle Programmanweisungen, die das Kriterium vor dem Aufruf beeinflussen. Bevor die genaue Funktionsweise des Slicers erläutert wird, soll zunächst im Abschnitt 5.2.1 auf die *WALA* Bibliothek eingegangen werden, welche die grundlegenden Funktionen für das Programm-Slicing bereitstellt. Daraufhin wird die technische Funktionsweise des Auditors in Teilschritten erläutert. Vertiefend wird in Abschnitt 5.2.3 und Abschnitt 5.3.3.3 auf die Konfigurationsmöglichkeiten des Auditors eingegangen; zunächst auf die Slicing Kriterien und darauffolgend auf die restlichen Einstellungen. Abschließend wird im Abschnitt 5.3.3.2 vertiefend auf die Rekonstruktion des Quelltextes eingegangen und erläutert, wie diese Funktionalität im Auditor implementiert und im Laufe der Arbeit verbessert wurde.

5.2.1. WALA Framework

Die T.J.Watson Libraries for Analysis (WALA) entstand ursprünglich aus einem DOMO Forschungsprojekt des IBM T.J. Watson Research Center und wurde 2006 unter der Eclipse Public License als Open Source Projekt der Community zur Verfügung gestellt [WALA f]. Zu den Kernfunktionen gehören u.a.:

- Analyse von Java-Typ-Systemen und Klassenhierarchien
- Analyse von Interprozeduralen Datenflüssen
- Context-sensitive tabulation-based Slicer
- Konstruktion Pointeranalyse und Callgraph/Aufrufgraph
- Bibliothek (Shrike) zum Lesen und Schreiben von Java-Bytecode

Context-sensitive bedeutet im Kontext von Programm-Slicing, dass beim Traversieren des PDG einzig Pfade vorkommen, die zum Aufruf-Kontext passen [Krinke 2003]. Slicer, welche dagegen *context-insensitiv* arbeiten, unterscheiden Objekte bspw. nach deren deklarierten Typen oder Klassen und sind wesentlich ungenauer, da die Resultate Knoten mit Anweisungen enthalten können, die keinerlei Einfluss auf das Slice-Kriterium haben. Krinke vergleicht in seiner Arbeit beide Verfahren und kommt zu dem Ergebnis, dass *context-insensitive* Verfahren nicht nur langsamer, sondern ebenso ungenauer sind. Zudem erläutert er, dass ein *context-sensitives* Verfahren effizient über die *summary edges* von einem SDG realisiert werden kann, welche transitive Abhängigkeiten zu den aufrufenden Prozeduren abbilden (siehe Abschnitt 3.5.2).

Tabulation-based Vorgehen dagegen beschreibt einen Algorithmus, vorgestellt von Reps et al. [Reps u. a. 1995], mit dem Probleme der interprozeduralen Datenfluss-Analyse mit Hilfe von Graphen-Erreichbarkeit gelöst werden können. Dieser Vorgehensweise geht vorher, dass zunächst ein Graph aufgebaut werden muss, der sich aus den Programmfluss-Graphen und den Funktionen der Datenflüsse ableitet.

Die Umsetzung des Analysewerkzeugs nutzt einen großen Teil der zuvor genannten WALA-Funktionen, um aus dem zu analysierenden Java Bytecode einen Slice des Programmes zu erhalten. Grob wird mit Hilfe der Analysemöglichkeiten die Klassenhierarchie und interprozeduralen Datenflüsse untersucht, um Programm *Entrypoints* (Einstiegspunkte) zu definieren, woraus sich ein *Callgraph* (Aufrufgraph) für einen begrenzen Programmteil berechnen lässt. Mit dem Aufrufgraphen und einer Pointeranalyse lässt sich schließlich ein spezieller *Systemabhängigkeitsgraph* (SDG) bilden, welcher zusammen mit den Slice-Kriterien vom WALA Slicer genutzt wird,

© Philip Phu Dang Hoan Nguyen

um ein *Backward-Slicing* auszuführen und letztendlich die Programmzeilen des resultierenden Slices zu erhalten (genaue Erläuterungen folgen in der Beschreibung der Funktionsweise im Abschnitt 5.2.2).

Der SDG von WALA zeichnet sich dadurch aus, dass die Knoten des Graphen, welche Anweisungen entsprechen, in einer besonderen *Zwischendarstellung (IR)* (siehe Abschnitt 3.2) vorzufinden sind. Die WALA IR ist eine zentrale Datenstruktur, welche die Anweisungen jeweiliger Methoden repräsentiert. Die Zwischendarstellung ähnelt von der Sprache her zwar dem JVM Bytecode, basiert jedoch auf der SSA-Form, welche nicht mit einer Stack-Abstraktion, sondern mit mehreren symbolischen virtuellen Registern arbeitet [WALA e, b]. Die generelle Funktionsweise der SSA-Form wurde in Abschnitt 3.2.2 eingeführt. Die WALA IR Klasse repräsentiert jeweils eine Methode als Kontrollflussgraph, bestehend aus sogenannten *basic blocks*, mitsamt aller Instruktionen. Die Abbildung 5.1 zeigt exemplarisch eine IR-Struktur mit *basic blocks* und dem zugehörigem CFG zu einem Programmteil mit einer If-Anweisung.

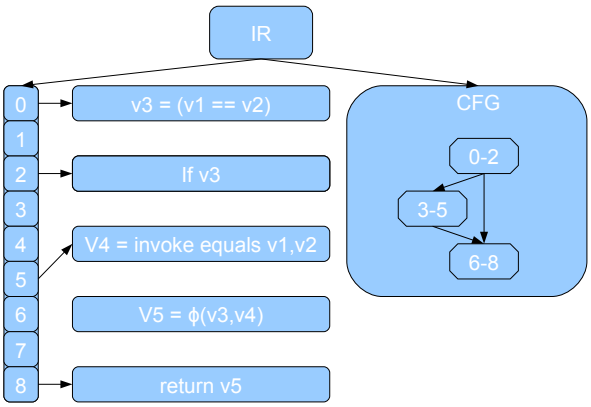

Abbildung 5.1.: WALA IR mit CFG (Quelle: [Dolby und Sridharan 2010])

Die Bibliothek bringt alle benötigten Methoden mit, um bspw. über die Knoten des SDGs oder CFGs zu iterieren und Methoden und Instruktionen zu analysieren. Die Implementierung der SSA-Instruktionen bietet zudem Möglichkeiten Variablendefinitionen und Verwendungen über Methoden wie `getDef()` und `getUses()` und verfügbare *Phi-Anweisungen* zu verfolgen. Die Abbildung 5.2 zeigt zum gleichen Beispiel, dass in der IR eines Knotens alle Definitionen und -verwendungen in einer `DefUse` Datenstruktur gesammelt und somit über `getDef(3)` die Definition der Variable `v3` in Block 0 und `getUses(3)` die beiden Verwendungen in Block 2 und 6 lokalisiert werden können.

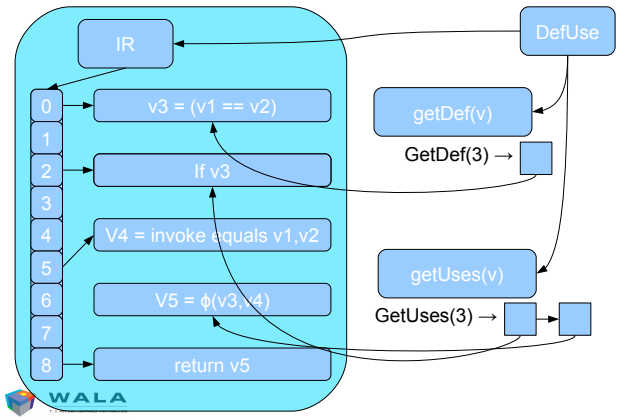

Abbildung 5.2.: WALA IR Funktionen: DefUse (Quelle: [Dolby und Sridharan 2010])

5.2.2. Funktionsweise

Die technische Funktionsweise des Auditors lässt sich in folgende Teilschritte unterteilen:

1. Aufbau des Analysescope
2. Erzeugen der Klassenhierarchie
3. Ermittlung der Entrypoints
4. Ermittlung der Pointeranalyse
5. Berechnung des Callgraphen
6. Ermittlung des Slicing-Kriterium
7. Erzeugen des Slice mit SDG
8. Rekonstruktion des Java-Quelltextes

Die Abbildung 5.3 zeigt die Datenflüsse zwischen den verschiedenen Prozessen vom Start des Auditors bis zum Bereitstellen der Analyseergebnisse. Im Folgenden sollen die Teilschritte kurz beschrieben werden, um die Implementierung mit Hilfe von WALA und die Zusammenhänge der jeweiligen Schritte aufzuzeigen.

Aufbau des Analysescope

Der Analysescope (Analyseumfang) spezifiziert das zu analysierende Programm und bestimmt ferner, welche benutzten Bibliotheken einbezogen werden sollen.
Das `AnalysisScope` Objekt kann definiert werden mit Hilfe der Klasse `AnalysisScopeReader` und der Methode `makeJavaBinaryAnalysisScope()`, welche einen Klassenpfad und eine sogenannte *exclusion file* benötigt [WALA a]. Die *exclusion file* ist eine Textdatei, welche im Programmverzeichnis liegen muss und bestimmt,

© Philip Phu Dang Hoan Nguyen

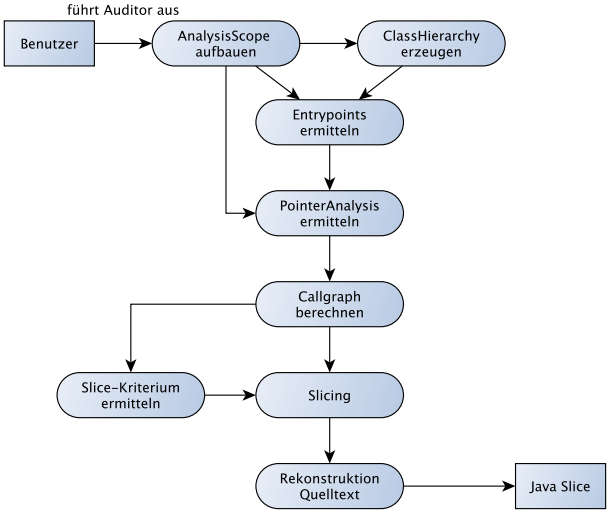

Abbildung 5.3.: Datenflussdiagramm zur Funktionsweise des Auditors

welche Bibliotheken bei der Analyse ignoriert werden sollen. Die Auflistung der auszuschließenden Bibliotheken kann dabei zeilenweise entweder mit Angabe der kompletten Java Klasse inklusive Paketnamen (bspw. `java.lang.Integer`) oder mit dem Wildcard Zeichen „*“ auf Pakete beschränkt geschehen (bspw. `javax.net.*`). Das gewählte Ausschließen von Java Standardbibliotheken kann verhindern, dass der Analyseumfang zu weit gefasst ist und Datenflüsse und Kontrollflüsse in den Slice gelangen, welche bspw. aufgrund der Tiefe der Java Standardbibliotheken nicht relevant oder von Interesse sind (bspw. GUI Bibliotheken). Zusätzlich verursachen vor allem stark verbreitete Bibliotheken oder fundamentale Imports wie `java.lang` (Standardbibliothek für Java Klassen, Objekte oder Typen) schnell enormen Overload, da für jede Programmzeile mehrere interprozedurale Kontrollflüsse in die jeweiligen Bibliotheken entstehen.

Daher empfiehlt es sich, für Analysten die *exclusion file* nach Anwendung anzupassen, um den Analysescope entweder feiner oder weiter zu fassen, da dieser, wie in Abbildung 5.3 zu sehen, Auswirkungen auf die Erzeugung der Klassenhierarchie, die Bestimmung der Entrypoints und die Erzeugung des Callgraphen hat.

Neben dem *exclusion file* wird für die Definition des Analysescopes noch der Klassenpfad des zu analysierenden Programmes benötigt. WALA unterstützt wie bereits erwähnt einzig das Slicing von Java Bytecode, weswegen der Pfad entweder Java .class Dateien oder einer .jar Datei (Java Archive) entsprechen muss. Die Konfigurationsdatei des Auditors erwartet das Programm fest definiert als eine .jar Datei inklusive Pfad.

Erzeugen der Klassenhierarchie

Mit Hilfe des `AnalysisScope` muss ein WALA `ClassHierarchy` Objekt aufgebaut werden, bevor im nächsten Schritt alle *Entrypoints* bestimmt werden können. `ClassHierarchy` ist eine zentrale Ansammlung von `IClass` Objekten, welche den jeweiligen Java-Klassen aus der Jar-Datei und den genutzten Bibliotheken entsprechen. Beim Erstellen der `ClassHierarchy` wird der Bytecode des Programmes in den Speicher geladen und grundlegende Informationen wie die Klassen und Typisierung geparst.

Ermittlung der Entrypoints

Sind `AnalysisScope` und `ClassHierarchy` erzeugt, folgt die Ermittlung der *Entrypoints* (Programm-Einstiegspunkte). *Entrypoints* definieren die Startknoten für den zu erzeugenden Callgraphen und somit, an welchen Stellen im Programm die Ausführung beginnen kann. Ferner bestimmen die Entrypoints im Zuge dessen auch umgekehrt diejenigen Callgraph-Knoten, welche bei einem Backward-Slicing vom Slicing-Kriterium beginnend erreicht werden sollen.

Für die Ermittlung der `Entrypoints` gibt es unterschiedliche Strategien. WALA bietet bspw. mit `makeMainEntrypoints()` die Möglichkeit, die *main*-Methode einer bestimmten oder aller Java-Klassen (im AnalysisScope) als Einstiegspunkt zu wählen. Weniger eingeschränkt sucht die Methode `AllApplicationEntrypoints()` nach allen möglichen Einstiegspunkten und ignoriert abstrakte Methoden.

Es ist ebenfalls möglich eigene Methoden zu implementieren, welche durch die zuvor erzeugte `ClassHierarchy` iterieren und gewünschte Entrypoints auswählen. Die Vorgehensweisen von Gulmann [Gulmann 2014] und Detmers [Detmers 2016] durchsuchen die `ClassHierarchy` nach einer zu untersuchenden Klasse, die vom Benutzer angegeben werden muss. Gulmann schränkt im Unterschied zu Detmers noch ein, dass einzig Klassen als Entrypoints gewählt werden, die öffentlich und dementsprechend in Java als `public` definiert sind.

Über die Angabe der `mainclass` kann in der Konfigurationsdatei die Klasse angegeben werden, dessen Entrypoints ermittelt werden sollen; sollten keine Entrypoints gefunden werden, versucht der Auditor automatisch alternative Methoden anzuwenden (vgl. SF05 in Abschnitt 5.1.1.1). Bezüglich der Reihenfolge werden hierbei zunächst die Methoden angewendet, die nach der expliziten Klasse suchen und erst bei Fehlschlag die alternative Methoden wie `makeMainEntrypoints()` und schließlich `AllApplicationEntrypoints()` verwendet, da diese grob sind und daraus ggf. ein zu großer Callgraph resultieren könnte.

© Philip Phu Dang Hoan Nguyen

Ermittlung der Pointeranalyse und Berechnung des Callgraph

Für den nächsten Schritt muss eine Pointeranalyse durchgeführt und der Callgraph berechnet werden. WALA erstellt bei der Pointeranalyse den Callgraphen als Nebenprodukt (on-the-fly) und löst damit die Zielknoten auf, welche über die dynamischen Aufrufe erreicht werden. Der Callgraph enthält alle traversierten Knoten (`CGNode` Objekte), welche im Ganzen die Struktur des möglichen Programmaufrufs darstellen [WALA g].

Pointeranalysen werden in statischer Programmanalyse genutzt, um Referenzen zwischen Pointern oder Heap-Referenzen und Variablen bzw. Speicherorte zu beobachten und nachzuvollziehen. Die sogenannte *alias analysis* bestimmt in diesem Zusammenhang ebenfalls, wenn mehrere Pointer den gleichen Speicher referenzieren und nutzen [Hind 2001]. *Aliasing* entsteht nicht ausschließlich bei Nutzung von Pointern, sondern ebenfalls bei einfacher Referenzierung von Objekten wie im folgenden Programmbeispiel 5.1 [Harvard 2011]:

```
1 //Example 1
2 MyClass obj1 = new MyClass(1,"test");
3 MyClass obj2 = obj1;
4
5 //Example 2
6 void m(Object a, Object b){...}
7 m(x,x); // a and b alias in body of m
```

Listing 5.1: Aliasing Beispiele

Im ersten Beispiel besteht ein Aliasing für das Objekt `obj1` und `obj2` durch eine direkte Referenzierung in Zeile 3, während im zweiten Beispiel eine sog. *call-by-reference* besteht, die durch die Verwendung des gleichen Objektes für mehrere Parameter in Zeile 7 verursacht wird. Innerhalb der Methode `m()` besteht ein Aliasing zwischen den Variablen `a` und `b`, da beide auf den gleichen Speicher von `x` zugreifen.

WALA bietet ein Framework für *flow-insensitive* Pointeranalyse nach Anderson, welche mit einigen Built-in Algorithmen ausgeliefert wird [WALA d]. *Flow-insensitive* unterscheidet sich von der *flow-sensitiven* Pointeranalyse, indem sie nicht für jeden Programmpunkt berechnet, welche Pointerausdrücke auf welchen Speicherort zeigen, sondern entweder über das ganze Programm oder eine einzelne Methode dieses Programms berechnet wird. *Flow-sensitive* Pointeranalyse ist jedoch üblicherweise zu ineffizient und wird im Kontext von WALA nicht benötigt, da der Programmpunkt mit Hilfe der Entrypoints fest definiert ist.

Mit Hilfe des WALA `CallGraphBuilder` lassen sich `PointerAnalysis` und `CallGraph` erzeugen, wobei bei der Initialisierung des `CallGraphBuilders` einer der Built-in Algorithmen für die Pointeranalyse ausgewählt werden muss (`ZeroCFA`, `ZeroOneCFA` oder `ZeroOneContainerCFA`).

Grob unterscheiden sich diese im Detailgrad des *context-sensitiven* Verfahrens. Während `ZeroCFA` *context-insensitiv* arbeitet und alle Objekte desselben Typs als ein einzelnes abstraktes Objekt behandelt, kommt `ZeroOneCFA` der *context-sensitiven* „Andersen Pointeranalyse“ am nächsten. `ZeroOneContainerCFA` erweitert das Verfahren einzig um die *context-sensitivity* für Collection bzw. Container Objekte [WALA d]. Aus den Erfahrungen der vorhergehenden Arbeiten wurde für die Implementierung die Strategie `ZeroOneCFA` festgesetzt, da dessen Präzision in allen Fällen ausreichte.

Aus dem `CallGraphBuilder` wird mit Hilfe der `Entrypoints` und dem `AnaysisScope` der `CallGraph` erzeugt, welcher als *context-sensitiver* Aufrufgraph alle Methoden zu einem Kontext als Knoten besitzt.

Ermittlung des Slicing-Kriteriums

Die Ermittlung des Slicing-Kriteriums ist der letzte verbleibende Schritt, der noch für das Backward-Slicing fehlt. Das Slicing-Kriterium definiert, wie in Kapitel 3.5 beschrieben, welche Anweisung oder Variable als Fokus für das Slicing gesetzt wird. Der Auditor erwartet in der Konfigurationsdatei eine sogenannte *Callee* Methode, welche als Slicing-Kriterium gesetzt und letztendlich analysiert werden soll. Der Slicer erwartet das Slicing-Kriterium als ein spezifisches `Statement` Objekt [WALA e], welches seine Begründung darin findet, dass die textuelle Angabe des `Callees` zu unspezifisch ist. Einerseits sind Methodennamen nicht eindeutig und können in unterschiedlichen Klassen mehrfach existieren. Zum anderen wäre es zu rechenintensiv, durch den kompletten Callgraph zu iterieren und in jeder Methode alle Anweisungen zu durchsuchen. Stattdessen soll durch den Anwender noch eine aufrufende *Caller* Methode angegeben werden, in welcher sich der *Callee* befindet.

Die Implementierung des Auditors definiert zum einen eine Methode `findMethods()` mit der der `CallGraph` traversiert wird und nach dem aufrufenden *Caller* Knoten (`CGNode`) gesucht wird. Da durch das Überladen von Methoden diese mehrfach mit demselben Namen in einer Klasse existieren können, wurde die Methode dahingehend erweitert, dass über einen weiteren Parameter gesteuert werden kann, ob mehrere *Caller* zurückgegeben werden sollen oder bei Fund direkt dieser für die Analyse verwendet werden soll (vgl. SF07 in Abschnitt 5.1.1.1).

 © Philip Phu Dang Hoan Nguyen

In einem weiteren Schritt iteriert die Methode `findCallsTo()` durch die *Caller* Knoten und sucht über die IR alle `SSAInstructions` nach dem *Callee* ab. Da ebenso die *Callee* Methode mehrfach auftauchen kann und zusätzlich unter verschiedenen Klassen nicht eindeutig sein muss, kann es erneut vorkommen, dass mehrere Anweisungen gefunden werden. Aus SF08 (siehe Abschnitt 5.1.1.1) entstand die Erweiterung, dass der Methode `findCallsTo()` über die Konfigurationsdatei noch eine Callee Klasse übergeben werden kann, um das Slice-Kriterium spezifischer definieren zu können.

Dieselbe Methode wurde ebenfalls um die Erweiterungen der „Callee-Objektverfolgung“ und der „Individuelle Instanziierungsmethoden“ (Vgl. SF13 und SF14 in Abschnitt 5.1.1.1) in einer späteren Entwicklungsiterationen ergänzt, welches in Abschnitt 5.3.3.1 beschrieben wird.

Erzeugen des Slice mit SDG

Wie eingangs erwähnt, nutzt die Implementierung Backward-Slicing, um einen Slice zu erzeugen. Die WALA Methode `Slicer.computeBackwardSlice()` erwartet neben dem Slice-Kriterium (`Statement`), ein `CallGraph`-Objekt, ein `PointerAnalysis`-Objekt und die beiden Slicing-Abhängigkeitsoptionen `DataDependenceOptions` und `ControlDependenceOptions`, welche im Abschnitt 5.2.3 näher erläutert werden. Die Abbildung 5.4 verdeutlicht nochmals das Zusammenspiel der Komponenten mit dem WALA `Slicer`.

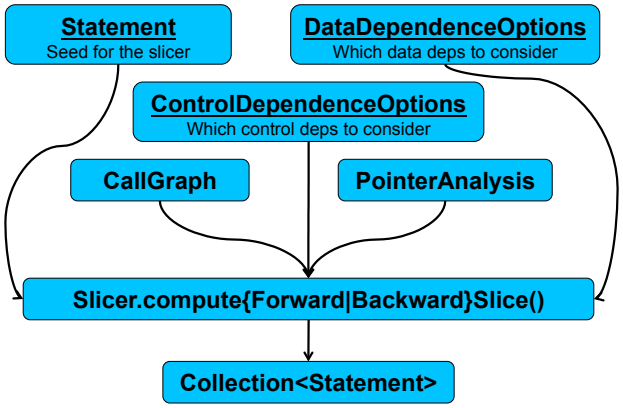

Abbildung 5.4.: Übersicht der Eingaben und Ausgaben des WALA Slicers (Quelle: [Dolby und Sridharan 2010])

Zur Laufzeit wird zunächst der SDG aus dem Callgraphen, der Pointeranalyse und den Abhängigkeitsoptionen aufgebaut. Mit Hilfe des SDG und dem Slice-Kriterium wird beim Slicing über das Prinzip *Erreichbarkeit* rückwärtsgerichtet traversiert und der Slice ermittelt, wie bereits in Abschnitt 3.5.2 beschrieben. Der Slice stellt hierbei

eine Collection von `Statements` dar, welche für die folgende Rekonstruktion des Java-Quelltextes von weiterer Bedeutung ist.

Rekonstruktion des Java-Quelltextes

Nachdem mit dem Slice alle `Statements` zur Verfügung stehen, die von Interesse sind, müssen diese im Folgeschritt in den Java Quellcode zurück übersetzt werden, um sie dem Anwender in lesbarer Form anbieten zu können. Anhand des Programmausschnitts 5.2 soll kurz erläutert werden, wie aus dem Slice die Zeilennummern des ursprünglichen Java-Quelltextes ermittelt werden.

```
1  public static Map<String, Set<Integer>> dumpSlices(final Collection<Statement> slice){
2
3    for(Statement s: slice)
4      int bcIndex, instructionIndex = ((NormalStatement) s).getInstructionIndex();
5      bcIndex = ((ShrikeBTMethod) s.getNode().getMethod()).getBytecodeIndex(instructionIndex);
6
7      int src_line_number = s.getNode().getMethod().getLineNumber(bcIndex);
8
9      String[] originalPath = s.getNode().getMethod().getDeclaringClass().getName().toString().
           split("/");
10     currentNumbers.add(src_line_number);
11     ...
```

Listing 5.2: Vereinfachter Programmausschnitt zur Zeilenermittlung

Mit der Methode `getInstructionIndex()` in Zeile 4 lässt sich aus einem Statement der Index der SSA Anweisung vom IR ausgeben, welcher aus dem Bytecode erzeugt wurde. Daraus erschließt sich der Bytecode Index (mit Hilfe der Teilbibliothek *Shrike*), der als Parameter in der Methode `getLineNumber()` genutzt werden kann, um die zugehörige Zeilennummer der entsprechenden Java-Datei zu ermitteln. Während der Entwicklung neuer Testfälle im Zuge der Evaluation fiel auf, dass die Rückgewinnung der Zeilennummern vom Bytecode nur funktioniert, wenn beim Kompilieren mit `javac` über die Debugging Option `-g` die Zeilennummer Debugging-Informationen generiert werden[1]. Zu welcher Datei die Anweisung gehört, kann hierbei über das Statement selbst erschlossen werden (siehe Zeile 9) [WALA c].

Da die Slices über diese Methodik teilweise nur einzelne Anweisungen ausgeben, müssen die zugehörigen Klassen, Methoden und Kontroll-Anweisungen inklusive Kopf und Rumpf zusätzlich ermittelt werden. Eine detaillierte Beschreibung dieser Erweiterung wird im Abschnitt 5.3.3.2 näher gegeben.

[1] Bestätigt durch Dolby in der WALA Mailingliste: `https://sourceforge.net/p/wala/mailman/message/36220880`, Aufruf: 15.02.2018

© Philip Phu Dang Hoan Nguyen

5.2.3. Slicing Parameter

Zuvor wurde beschrieben, dass beim Backward-Slicing neben dem Slice-Kriterium, dem Callgraphen und der Pointeranalyse zwei Slicing-Abhängigkeitsoptionen erwartet werden. Die Daten- und Kontrollabhängigkeitsoptionen haben einen maßgeblichen Einfluss auf den resultierenden SDG und sind am Ende in der Regel ausschlaggebend, ob ein Slice in annehmbarer Zeit berechenbar ist und ob der Slice für eine Analyse überhaupt nutzbare Resultate erzielt. Beide Optionen definieren, wie detailliert jeweilige Daten- und Kontrollabhängigkeiten im SDG verfolgt werden sollen und bestimmen, welche Kanten traversiert und im Slice enthalten sind.

Die verfügbaren Optionen sind von WALA lediglich grob dokumentiert und werden teilweise nicht vollumfänglich aufgelistet [WALA e] [Dolby und Sridharan 2010], was unterstreicht, dass die genauen Auswirkungen dieser Optionen genauer evaluiert werden müssen. Detmers führte bereits unterschiedliche Tests mit allen möglichen Kombinationen der Daten- und Kontrollabhängigkeitsoptionen durch [Detmers 2016], doch konnten mit vorliegender Arbeit in der Evaluation noch zusätzliche Erkenntnisse gewonnen werden, welche im Folgenden ebenfalls erläutert werden.

5.2.3.1. Datenabhängigkeitsoptionen

WALA bietet folgende Datenabhängigkeitensoptionen an:

FULL und NONE Verfolgt entweder alle oder ignoriert alle Datenabhängigkeiten. Während `NONE` keine Ergebnisse erzielt, bewirkt `FULL` das Gegenteil und konstruiert den detailliertesten, aber ebenso größten und rechen-intensivsten SDG.

NO_BASE_PTRS Verhält sich wie `FULL`, ignoriert jedoch Datenabhängigkeiten, welche indirekten Speicherzugriff durch *base pointer* definieren. Dadurch wird bspw. die Initialisierung von Arrays und deren Rückgaben ignoriert. WALA gibt als Beispiel an, dass die Abhängigkeiten von der Initialisierung der Variable `x` zu `y=x.f` ignoriert werden.

NO_HEAP Verhält sich wie `FULL`, ignoriert jedoch Datenabhängigkeiten von und zum *Heap* (dynamischer Zwischenspeicher). Dadurch fallen unter anderem Instanzvariablen, direkte Operationen auf diese Instanzvariablen und auch die Initialisierung dieser durch den Konstruktor weg.

NO_EXCEPTIONS Verhält sich wie `FULL`, ignoriert jedoch alle Datenabhängigkeiten zu `throw` und `catch` Knoten.

REFLECTION Verhält sich wie die Kombination `NO_BASE_PTRS` und `NO_HEAP` (`NO_BASE_NO_HEAP`), ergänzt jedoch noch Datenabhängigkeiten, die aus checkcast Anweisungen entstehen. Diese Option kann teilweise genutzt werden, um Java-Typumwandlungen (type casting) zu verfolgen.

Kombinationen Darüber hinaus lassen sich alle möglichen Kombinationen aus den vorgestellten Optionen zusammensetzen, wie bspw. `NO_BASE_NO_HEAP`, `NO_BASE_NO_HEAP_NO_EXCEPTIONS` oder `NO_HEAP_NO_EXCEPTIONS`.

5.2.3.2. Kontrollabhängigkeitsoptionen

WALA bietet folgende Kontrollabhängigkeitsoptionen an:

FULL Verfolgt alle Kontrollabhängigkeiten und konstruiert somit analog zu der Datenabhängigkeitsoption den größten und detalliertesten SDG und bewirkt damit, dass beim Slicing der größte Slice erzeugt wird.

NONE Ignoriert alle Kontrollabhängigkeiten beim Erzeugen des SDGs. Slicing mit der Option `NONE` erzielt laut Detmers in den meisten Fällen unbrauchbare Ergebnisse, da vom Einstiegsknoten alle Kanten ignoriert werden und somit im Slice einzig das Slicekriterium enthalten ist.

NO_EXCEPTIONAL_EDGES Ignoriert *Exception-Kontrollflüsse*. Die Option `NO_EXCEPTIONAL_EDGES` wird in der offiziellen Dokumentation von WALA nicht explizit benannt und nur vage im Javadoc und in einer Präsentation von zwei der drei Hauptentwickler Dolby und Sridharan erwähnt [Dolby und Sridharan 2010]. In der Entwicklung und der Evaluation im Rahmen dieser Arbeit konnte festgestellt werden, dass sie für die Analyse vorteilhaft ist und letztendlich ebenfalls das teilweise ungeklärte Verhalten in vorhergehender Arbeiten von Gerken und Detmers begründen kann. Die gewonnenen Erkenntnisse werden im nächsten Abschnitt gesondert erläutert.

5.2.4. Erkenntnisse Exception-Kontrollflüsse

Um den Einfluss der Kontrollabhängigkeitsoption `NO_EXCEPTIONAL_EDGES` zu erläutern, soll in Erinnerung gerufen werden, was Kontrollabhängigkeit bedeutet (siehe Abschnitt 3.4.3). Mit dem Einbeziehen der Kontrollabhängigkeiten werden grundsätzlich alle Anweisungen dem Slice ergänzt, welche die Ausführung des Slice Kriteriums (demzufolge der *Callee*-Methode) an einem Programmpunkt zwischen Anfangsknoten und dem *Callee* kontrollieren und somit auch auf einen anderen Pfad lenken können. Im Grundgedanke des intraprozeduralen Kontrollflusses sind dies Java-

 © Philip Phu Dang Hoan Nguyen

Kontrollstrukturen wie Verzweigungen (`if/else`, `switch`), Schleifen (`for`, `while/do`) oder Anweisungen, welche die Schleifen beenden können, wie `goto, break, continue` oder `return`. Aus den Testfällen TF123 und TG124 und Aussagen von Sridharan[2] ergibt sich die Erkenntnis, dass Kontrollabhängigkeiten auf Ebene interprozeduraler Kontrollflüsse, von daher zwischen verschiedenen Prozeduren auch auf eine andere Art und Weise bestehen können. Die hier genannten *Exception-Kontrollflüsse* (engl. *exceptional control flow*), welche durch die Kontrollabhängigkeitsoption `NO_EXCEPTIONAL_EDGES` explizit ignoriert werden, spielen hierbei eine entscheidende Rolle. Einfache Aufrufe wie bspw. das Definieren primitiver Objekte oder Methodenaufrufe führen grundsätzlich zu keiner Kontrollabhängigkeit, da im Regelfall nach Abschluss der Durchführung die darauf folgenden Anweisungen ausgeführt werden. Ausgenommen ist der Fall, in dem in der Prozedur (bspw. in der Fremdbibliothek oder in einer anderen Klasse bzw. Methode) potenziell Exceptions geworfen werden können. Dies bedeutet, dass bei einem Fehlerfall oder bei ungewolltem Verhalten der Kontrollfluss in dieser Prozedur sich verändern, einen anderen Pfad wählen oder sogar terminieren könnte. Wird somit die Kontrollabhängigkeitsoption `FULL` genutzt, werden solche Exception-Kontrollflüsse im SDG berücksichtigt und gelangen in den resultierenden Slice.

Um diese spezielle Art von Kontrollfluss zu erläutern, soll folgender Programmauszug 5.3 betrachtet werden:

```
1 try {
2     int a = 5;
3     check(a);
4     c = secretFct(a);
5     return c;
6
7 } catch (Exception e) {
8     return 0;
9 }
```

Listing 5.3: Beispielprogramm Exception-Kontrollflüsse

Im `try`-Block wird zunächst eine Variable `a` mit dem Wert 5 initialisiert und mit einer Methode `check()` in Zeile 3 überprüft, ohne näher darauf einzugehen, was diese Funktion macht. Unabhängig von dieser Methode wird eine zweite Methode `secretFct()` in Zeile 4 aufgerufen, dessen Ergebnis zurückgegeben werden soll. Im Grundgedanke würde bei einem Slicing mit dem Slice-Kriterium `secretFct()` die Zeile mit der Methode `check()` nicht in den Slice gelangen, da sichtbar weder Daten- noch Kontrollabhängigkeiten bestehen. Läuft die Methode jedoch auf einen

[2]Siehe WALA Mailingliste: `https://sourceforge.net/p/wala/mailman/message/35964881`, Aufruf: 15.02.2018

Fehler und es würde eine Exception geworfen werden, würde das Programm in den `catch`-Block springen und die Ausführung der Methode `secretFct()` überspringen. Das gewählte Beispielprogramm würde im Übrigen die Methode `check()` genauso als Exception-Kontrollflussknoten erkennen, wenn es keinen `try-catch` gäbe, da das Programm nach `check()` durch die Exception terminieren und somit das Slice-Kriterium ebenfalls nicht erreichen würde.

Problematisch ist hierbei die Tatsache, dass es schwierig ist, den Überblick über alle genutzten Klassen und Methoden der zu analysierenden Anwendung und zusätzlich der, für die Analyse inkludierten, Bibliotheken zu behalten und zudem noch zu wissen, wo Exceptions geworfen werden können.

Dies führt dazu, dass die Benutzung der Kontrollabhängigkeitsoption `FULL` in gewisser Weise intransparent sein kann und Anwender bei der Betrachtung der Slicing-Ergebnisse bestimmte Teile als *false positive* einordnen. So konnten Gerken und Detmers in ihren Arbeiten Teile des Slices nicht vollständig nachvollziehen und haben diese Anweisungen nach deren Auffassung als *false positive* eingeordnet. Eines der Probleme [Gerken 2015] [Kapitel 5.4] wurde in dieser Arbeit als Testfall erneut aufgenommen und im Testfall123 evaluiert und gelöst. Analog dazu lässt sich ebenso die Verhaltensweise im Testfall *ObjectSensitivity* von Detmers [Detmers 2016] [Kapitel 6.3.2] erklären. Bei beiden Autoren kamen jeweils bestimmte Variablen des Typs Integer im Slice vor, obwohl diese keine sichtlichen Abhängigkeiten zum Slice-Kriterium verfügen. Diese *false positives* lassen sich damit begründen, dass die Wrapperklasse `Integer` im Gegenzug zum primitiven Datentyp `int` eine Exception-Kontrollflusskante darstellt, da sie ebenfalls Exceptions werfen kann[3].

Während der iterativen Entwicklung und der Evaluation des Auditors fiel noch ein weiteres Problem auf, welches ebenfalls durch die mangelnde Transparenz der Kontrollabhängigkeitsoptionen verursacht wird. Während Gerken einzig mit der Option `FULL` arbeitet, testet Detmers zwar in seiner Evaluation alle Optionen, beobachtet jedoch bei `NO_EXCEPTIONAL_EDGES` ein unerwartetes Verhalten beim Slicen, welches ebenfalls in der WALA Mailingliste[4] diskutiert wurde, von Dolby und in dieser Evaluation jedoch nicht nachvollzogen werden konnte. Detmers empfiehlt aus diesem Grunde ebenfalls den Gebrauch von `FULL` und verzichtet auf eine nähere Betrachtung der Option `NO_EXCEPTIONAL_EDGES`.

Aus den gewonnen Erkenntnissen der Testfälle dieser Arbeit ließ sich zeigen, dass die Kontrollflussoption `NO_EXCEPTIONAL_EDGES` sinnvoll ist, da zum einen die Größe und Rechenzeit des SDG minimiert (siehe Ausführungszeiten in Kapitel 6), und

[3] Siehe WALA Mailingliste: `https://sourceforge.net/p/wala/mailman/message/35964881`, Aufruf: 01.10.2017
[4] `https://sourceforge.net/p/wala/mailman/message/35354463`, Aufruf: 15.09.2017

 © Philip Phu Dang Hoan Nguyen

zum anderen die `false positives` verringert und ggf. verhindert werden. Problematisch ist, dass ein Anwender beim Slicing mit `FULL` generell nicht differenzieren kann, ob eine Anweisung im Slice ist, weil sie Datenabhängigkeiten hat oder weil sie eine Exception-Kontrollflusskante ist. Tatsächlich fällt bei den Tests auf, dass viele relevante Anweisungen ausschließlich zufällig indirekt durch die Exception-Kontrollflusskanten im resultierenden Slice enthalten sind und somit der Slicer als Ganzes relevante Abhängigkeiten übersieht. Das bedeutet gleichzeitig, dass viele `false negatives` erst durch die Nutzung der Option aufgefallen sind.

Die Testfälle *TF05: Objektverfolgung* und *TF06: Clinic (Verschlüsselung)* (siehe Abschnitt 6.2.6) geben Beispiele für `false negatives`. Zum einen fällt auf, dass nicht alle Arten von Instanziierungen (des Callee-Objektes) von WALA erkannt werden; zum anderen werden ebenfalls Methodenaufrufe auf das Callee-Objekt auf dem Pfad zum Callee nicht von WALA erkannt, woraus die Anforderungen aus SF13 und SF14 (siehe Abschnitt 5.1.1.1) entstanden sind.

Die Beschreibung dieser und anderer Erweiterungen erfolgt im Abschnitt 5.3.3.1.

5.3. Implementierung

In diesem Kapitel wird neben den benutzten Dritt-Bibliotheken ebenso kurz das Build-Verfahren beschrieben (vgl. Abschnitt 5.3.1 und Abschnitt 5.3.2). Abschließend wird im Abschnitt 5.3.3.1 auf die Erweiterungen und Optimierungen eingegangen. Dabei werden die Implementierungen erläutert, sowie abschließend alle Konfigurationsmöglichkeiten zusammenfassend im Abschnitt 5.3.3.3 aufgelistet.

5.3.1. Bibliotheken

T.J. Watson Libraries for Analysis (WALA)

Die Java-Bibliothek *WALA* vom ursprünglichen DOMO Forschungsprojekt des IBM T.J. Watson Research Centers wurde im Abschnitt 5.2.1 detailliert vorgestellt und bildet die Kernfunktionen des Auditors. WALA ist ein Open Source Projekt und wird auf *GitHub* entwickelt sowie zur Verfügung gestellt.

Hauptkontributoren sind aktuell Julian Dolby und Manu Sridharan (ehemaliger Kernentwickler Stephen Fink ist inaktiv seit 2011)[5], welche ebenfalls aktiv die Community über eine Mailingliste[6] betreuen.

Die für die Implementierung des Auditors genutzte Version von WALA ist 1.3.9.

JavaParser

Die Java-Bibliothek *JavaParser* bietet unterschiedliche Werkzeuge, um Java-Quelltext zu generieren, analysieren und zu verarbeiten. Sie wird entweder unter einer GNU Lesser General Public License (LGPL) Lizenz oder einer Apache Lizenz angeboten und wird ebenfalls als Open Source Projekt auf GitHub entwickelt[7]. JavaParser erlaubt es, Quelltext in Form einer Objekt-Repräsentation in eine Java Umgebung zu bringen, um diese analysieren zu können. In der Form eines AST (siehe Abschnitt 3.2.1) kann gezielt und effizient der Quelltext systematisch über den Baum analysiert werden. Der grundsätzliche Einsatz der Bibliothek wurde bereits in Abschnitt 5.2.2 erläutert und wird im späteren Abschnitt 5.3.3.2 erneut aufgegriffen und detailliert behandelt.

Die für die Implementierung des Auditors genutzte Version von JavaParser ist 1.0.11.

[5] `https://github.com/wala/WALA/graphs/contributors`, Aufruf: 01.03.2018
[6] `https://sourceforge.net/p/wala/mailman/wala-wala`, Aufruf: 01.03.2018
[7] `https://github.com/javaparser/javaparser`, Aufruf: 01.03.2018

 © Philip Phu Dang Hoan Nguyen

Commons Lang

Die Java-Bibliothek *Commons Lang* bietet ergänzend zur Java Standardbibliothek diverse Methoden zur Manipulation der Kernklassen. Unter anderen zählen hierzu Hilfsmittel für java.lang, wie bspw. String Manipulationen, grundlegende numerische Methoden oder Objektreflexion. Ebenfalls beinhaltet es Erweiterungen zu der Klasse java.util.Date oder der Methoden hashCode, toString und equals[8].

Die für die Implementierung des Auditors genutzte Version von Commons Lang ist 2.3.

5.3.2. Build-Verfahren

Der Build-Vorgang des Auditors wird über das Build-Management-Werkzeug *Maven*[9] gesteuert, welches die drei genannten Bibliotheken mit in den Build einbezieht.

Mit Hilfe von Maven, lässt sich das „Bauen“ der Anwendung leichter verwalten und zusätzlich automatisieren.

Über den Aufruf `mvn compile` können alle Klassen zu Java *class files* (.class) kompiliert werden und abschließend über `mvn package` in einem Jar-Archiv (.jar) bereitgestellt werden. Maven kümmert sich hierbei um alle Abhängigkeiten (Bibliotheken) und kann diese ggf. nachinstallieren.

[8] `https://commons.apache.org/proper/commons-lang`, Aufruf: 01.03.2018
[9] `https://maven.apache.org`, Aufruf: 01.03.2018

5.3.3. Optimierungen & Erweiterungen

5.3.3.1. Slicing Erweiterungen

Wie bereits in Abschnitt 5.2.2 erläutert, wurden bei der Ermittlung des Slicing-Kriteriums mehrere Erweiterungen implementiert, die im Folgenden kurz erläutert werden. Hauptaugenmerk liegt hierbei auf den Methoden `findMethods()` und `findCallsTo()`, welche zum Ermitteln des Slicing-Kriteriums definiert wurden, um zunächst im Callgraph die *Caller* Knoten und dann in den jeweiligen *Caller* Knoten die *Callee* Anweisungen zu lokalisieren.

Callee und Caller Erweiterungen

Ursprüngliche Implementierungen von den vorausgegangenen Arbeiten lehnen sich an Implementierungsbeispiele von WALA an und unterstützen lediglich das Finden einzelner *Caller* und *Callee* Knoten, weswegen diese Erweiterung (vgl. SF07 und SF08 in Abschnitt 5.1.1.1 und Abschnitt 5.1.1.1) in dieser Arbeit dem Slicer ergänzt wurde.

Die Methode `findMethods()` unterscheidet sich von der ursprünglichen Form darin, dass sie den Callgraphen komplett traversiert und am Ende eine Liste aller *Caller* als `CGNodes` zurückgibt. `findCallsTo()` wiederum erwartet als Eingabeparameter dementsprechend eine Liste von `CGNodes` und iteriert über alle `SSAInstructions` der Knoten, um schließlich die *Callee* Knoten in Form einer `Statement`-Liste zu erhalten.

Eine weitere Erweiterung beschreibt Anforderungen der SF08 in Abschnitt 5.1.1.1, in der eine zusätzliche Angabe der *Callee Klasse* den *Callee* spezifischer festlegen kann. Die Klasse kann vom Benutzer über den Wert `src_callee_class` in der Konfigurationsdatei als reiner Klassenname (zum Beispiel `Cipher`) oder als vollständiger Java Paketpfad bspw. in der Form `Ljavax/crypto/Cipher` angegeben werden. `findCallsTo()` traversiert über alle Statements eines Callgraphknoten, bis eine Methode über folgende Abfrage gefunden wird:

```
1 if (call.getCallSite().getDeclaredTarget().getName().toString().equals(methodName))
```

Eine *call site* (`CallSiteReference`) beschreibt den Aufrufort, d.h. die invoke (Aufruf) Anweisung im Bytecode, und verfügt zudem über Informationen zum Programm-Counter und zum *declared target* (`MethodReference`), welcher der aufrufenden Methode entspricht. Die Erweiterung fügt eine weitere Abfrage hinzu, welche die Klas-

 © Philip Phu Dang Hoan Nguyen

se vom *declared target* mit der vom Anwender angegebenen vergleicht. Die Klasse `MethodReference` bietet für diesen Zweck die Methoden `getDeclaringClass()` und `getName()`, welche den kompletten Java Klassennamen mit Paketpfad ausgibt. Wird lediglich der Klassenname verglichen, kann auf das Objekt zusätzlich noch `getClassName()` aufgerufen werden.

Callee Objektverfolgung

Während der Evaluation wurden in einer Entwicklungsiteration zudem zwei weitere Anforderungen definiert (SF13 und SF14, siehe Abschnitt 5.1.1.1), um die `findCallsTo()` erweitert werden soll. Der Bedarf an einer Objektverfolgung für den Callee lässt sich, wie bereits im vorherigen Abschnitt 5.2.4 erläutert, durch die fehlenden Anweisungen beim Slicing mit der Kontrollabhängigkeitsoption `NO_EXCEPTIONAL_EDGES` rechtfertigen.

Wie bereits in der Systemfunktion SF13 kurz angedeutet, liegt die Ursache für das Fehlen der Methoden, die auf dem Objekt des Callees aufgerufen werden, vermutlich daran, dass WALA Datenabhängigkeiten lediglich auf Ebene primitiver Variablen präzise verfolgen kann. Spezifischer soll das bedeuten, dass WALA beim Analysieren der Datenflüsse von Variablen lediglich Modifikationen mit Hilfe von Pointeranalysen erkennt, die mit der direkten Referenzierung des Speichers zu tun haben. Somit werden bspw. Anweisungen erkannt, die mit arithmetischen Operationen eine Variable modifizieren (vgl. Testfall arith Referenz) oder bspw. die Variable direkt über eine Zuweisung auf einen anderen Speicherort umsetzen und somit modifizieren. Auf Ebene von Objekten scheint WALA nicht zu verfolgen, ob Objekte etwa durch Methodenaufrufe modifiziert werden.

```
1  ...
2  public static String entry_method(Integer anothervalue) {
3    SimpleTestObject obj = new SimpleTestObject();
4    obj.do_stuff();
5    Integer addvalue = SimpleTestObject.calc_param();
6    result = obj.important_method(addvalue, anothervalue);
7  ...
```

Listing 5.4: Sliceausschnitt des Testfalls 88

Der Testfall *TF05: Objektverfolgung* (siehe Abschnitt 6.2.5) zeigt ein einfaches Beispiel, welches die Einschränkung deutlich macht und später noch detaillierter erläutert wird. Der Ausschnitt 5.4 zeigt einen kleinen Teil des Slices, der aus der Kontrollabhängigkeitsoption `NO_EXCEPTIONAL_EDGES` und keinerlei Optimierung resultiert. Orange-markierte Zeilen sind jene, die im Slice enthalten sind, während die

Notation „...“ weitere Anweisungen andeutet, die aus Übersichtlichkeit in diesem Ausschnitt nicht in aller Deutlichkeit gezeigt werden.

Es ist zu erkennen, wie die Anweisung `obj.do_stuff()` in Zeile 4 nicht in den Slice gelangt, obwohl das Slice-Kriterium `obj.important_method` ist und die Klassen-Methode `do_stuff()` eine Klassenvariable des Objektes `obj` umschreibt und somit indirekten Einfluss auf das Slicing-Kriterium hat.

Das später im Testfall *TF06: Clinic (Verschlüsselung)* beschriebene Szenario (siehe Abschnitt 6.2.6) behandelt die gleiche Einschränkung, dass Methodenaufrufe auf dem Objekt des Slicing-Kriteriums im Slice nicht berücksichtigt werde, zeigt jedoch im Kontext einer Sicherheitsanalyse speziell nochmals den Nutzen, der mit einer Erweiterung erzielt wird.

Der Testfall untersucht als Slicing-Kriterium die Methode `doFinal()` der Bibliothek `javax/crypto/Cipher`, welche Funktionen zum Verschlüsseln/Entschlüsseln von Daten bereitstellt. Anhand des kurzen Beispielprogramms 5.5 sollen kurz die Vorteile der Objektverfolgung erläutert werden.

```
1  Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
2
3  Key key = ... // get / create symmetric encryption key
4  cipher.init(Cipher.ENCRYPT_MODE, key);
5
6  byte[] data1 = "abcdefghijklmnopqrstuvwxyz".getBytes("UTF-8");
7  byte[] data2 = "zyxwvutsrqponmlkjihgfedcba".getBytes("UTF-8");
8  byte[] data3 = "01234567890123456789012345".getBytes("UTF-8");
9
10 byte[] cipherText1 = cipher.update(data1);
11 byte[] cipherText2 = cipher.update(data2);
12 byte[] cipherText3 = cipher.doFinal(data3);
```

Listing 5.5: Beispielprogramm Cipher Verschlüsselung (Quelle: [Jenkov 2018])

Die Funktion `doFinal()` der Bibliothek `Cipher` (hier im Beispiel in Zeile 12) verschlüsselt zunächst eigentlich nur einen Block und signalisiert das Ende einer Verschlüsselung. Während `update()` bei mehreren Klartext-Blöcken ausgeführt wird, muss `doFinal()` abschließend auf den letzten Block angewendet werden, um mit Hilfe von Padding den Geheimtext-Block auf die festgesetzte Blockgröße aufzufüllen. Wichtige Informationen zu den Parametern wie Schlüssel oder Verschlüsselungsalgorithmus müssen jedoch vorher in einem konstruierten `Cipher`-Objekt gesetzt werden. Für die Analyse sind somit im Speziellen die Methodenaufrufe und Parameter in den Zeilen 1 und 4 interessant, da dort der Verschlüsselungsalgorithmus und Schlüssel übergeben werden. `Cipher.getInstance()` ist hierbei gleichzustellen

© Philip Phu Dang Hoan Nguyen

mit einem Java *New-Operator* und instanziiert ein neues `Cipher`-Objekt mit Angabe des Algorithmus. Die Methode `init()` definiert, welcher Verschlüsselungsmodus betrieben wird und welcher Schlüssel verwendet wurde.

Das Beispiel verdeutlicht, warum das Objekt (hier `cipher`), auf dem die *Callee* Methode (`doFinal()`) ausgeführt wird, für die Sicherheitsanalyse wichtig ist und warum es von der Erzeugung (`getInstance()`) und allen Methodenaufrufen (`init()` und `update()`) bis zum abschließenden Slicing-Kriterium verfolgt werden sollte.

Die Funktionsweise der Erweiterung lässt sich folgendermaßen in zwei Schritte zusammenfassen:

1. Iteriere durch die Caller Methode anhand des Callgraphen (`CGNode`), finde den Callee (optional Callees) und speichere die Klasse des Objektes, auf dem die Callee Methode angewendet wird.

2. Iteriere erneut durch den `CGNode`, suche nach Instanziierungen der Objektklasse und führe eine grobe intraprozedurale Pointeranalyse durch, um zusätzliche Anweisungen dem Slice hinzuzufügen.

Die Implementierung wird grob über einen Programmablaufplan (siehe Abbildung 5.5 und Abbildung 5.7) skizziert und im Folgenden anhand dessen erläutert.

Der erste Teil basiert auf der ursprünglichen Implementierung von `findCallsTo()` und speichert bei Fund des Callees einzig die Klasse des Aufruf-Objektes via `getDeclaringClass()` in einer Variable `call_class` (siehe Programmablauf Schritt 1).

Da die Pointeranalyse des zweiten Teils erneut die Caller Methode traversieren muss und somit zusätzliche Rechenzeit kostet, lässt sich diese Erweiterung zunächst über den Schalter `advanced_mode` in der Konfigurationsdatei mit `true` oder `false` ein-, bzw. ausschalten.

Des Weiteren iteriert der zweite Teil durch alle SSA-Anweisungen der Caller Methode und sucht zunächst nach den Instanziierungen der aus dem ersten Teil bestimmten Klasse `call_class`, um anschließend die Objekte in den Folgeanweisungen zu verfolgen. Die Unterscheidung der unterschiedlichen `SSAInstructions` wird in WALA über diverse Spezialisierungen dieser Klasse gemacht (vgl. Abbildung 5.6).

Die für die Implementierung relevanten Klassen sind:

SSANewInstruction (*An allocation instruction "new"*)
: Eine Anweisung mit Zuweisung mit Hilfe des New-Operators.

SSAInvokeInstruction (*A call instruction*)
: Eine Anweisung mit Aufruf einer Methode (engl. invoke).

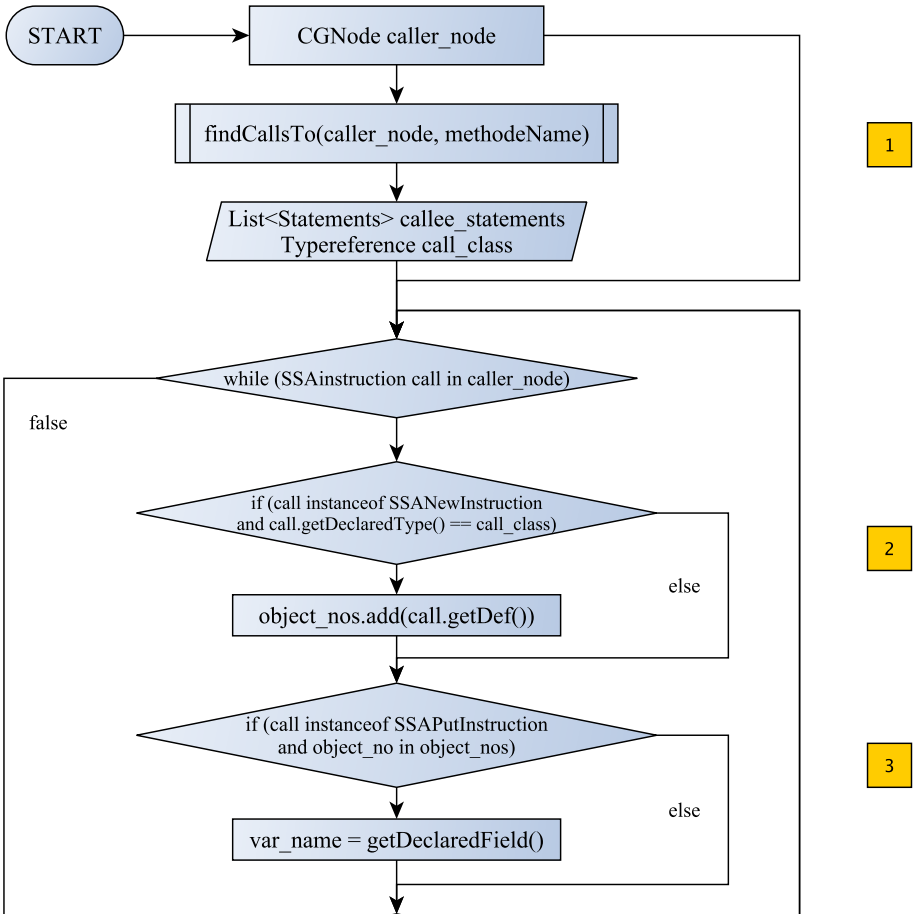

Abbildung 5.5.: Programmablaufplan für die Objektverfolgung (Teil 1)

SSAPutInstruction (*A putfield or putstatic instruction*)
Eine Anweisung, welche eine Variable zuweist.

SSAGetInstruction (*An instruction that reads a field (i.e. getstatic or getfield)*)
Eine Anweisung, die ein Feld liest.

SSAPhiInstruction (*A phi instruction*)
Ein ϕ-Knoten mit seinen Def und Uses.

Wird somit eine Anweisung der Klasse `SSANewInstruction` gefunden, wird zunächst überprüft, ob das erzeugte Objekt der gesuchten Klasse `call_class` entspricht, mit Hilfe der Methoden `getNewSite().getDeclaredType()` auf die SSA-Anweisung (siehe Programmablauf Schritt 2). Ist dies der Fall, kann mit der Methode `getDef()` die dazugehörige Variablennummer ausgegeben und in einer Liste `object_nos` gesichert werden.

© Philip Phu Dang Hoan Nguyen

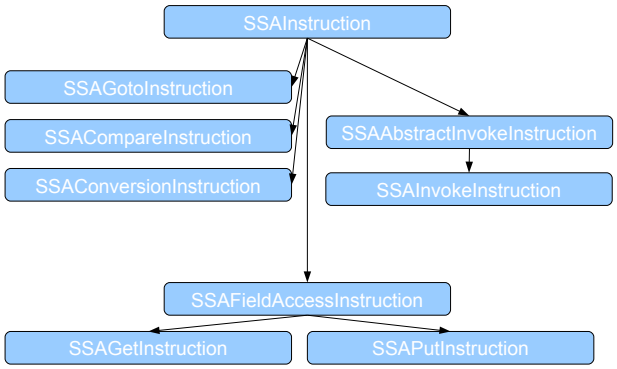

Abbildung 5.6.: Übersicht der SSA Instruktionstypen von WALA (Quelle: [Dolby und Sridharan 2010])

Da zu einer Variable bspw. durch Umdeklarieren oder unterschiedliche Kontrollflüsse mehrere Variablennummern bestehen können, muss die zuvor angesprochene Pointeranalyse durchgeführt werden.

Über die Analyse der `SSAGetInstructions` und `SSAPutInstructions` werden zum einen Programmflüsse erkannt, in denen Objekte gelesen und in einer weiteren Anweisung einer neuen Variable (und somit mit neuer Nummer) zugewiesen werden (siehe Programmablauf Schritt 3 und 4). Dafür In diesem Fall muss die Liste `object_nos` um diese Nummer ergänzt werden.

Zum anderen kann anhand der Analyse der ϕ-Knoten (`SSAPhiInstruction`) erkannt werden, wann durch Kontrollflüsse zwei verschiedene Variablennummern in einer neuen Variablennummer zusammenlaufen (siehe Erläuterungen zu SSA in Abschnitt 3.2.2). Der Testfall TF666 (phiknoten..doRSADecryption doFinal) zeigt exemplarisch, wie in einer `if-else`-Anweisung jeweils in beiden Verzweigungen die gleiche Variable `cipher` mit `getInstance()`) und unterschiedlichen Parametern definiert und in den Folgeanweisungen verwendet wird. Der ϕ-Knoten enthält zu allen Verzweigungen die entsprechenden Variablennummern und vereint diese mit einer neuen Nummer, welche in den Folgeanweisungen auch referenziert wird. In diesem Testfall wird so etwa `init()` auf dem Objekt mit der Variablennummer des ϕ-Knotens aufgerufen.

Die Implementierung berücksichtigt dies, indem die *„Uses“* der ϕ-Knoten mit den Variablennummern der Liste `object_nos` verglichen werden und bei Fund die Nummer des ϕ-Knotens mit `getDef()` ergänzt wird (siehe Programmablauf Schritt 5).

In den folgenden Iterationsschritten werden die Methodenaufrufe und somit die Anweisungen der Klasse `SSAInvokeInstruction` genauer untersucht. Da mit Hilfe von `getReceiver()` das Objekt über die Variablennummer identifiziert werden kann, auf dem diese Methode aufgerufen wird, können wir diese Nummer mit den vorher

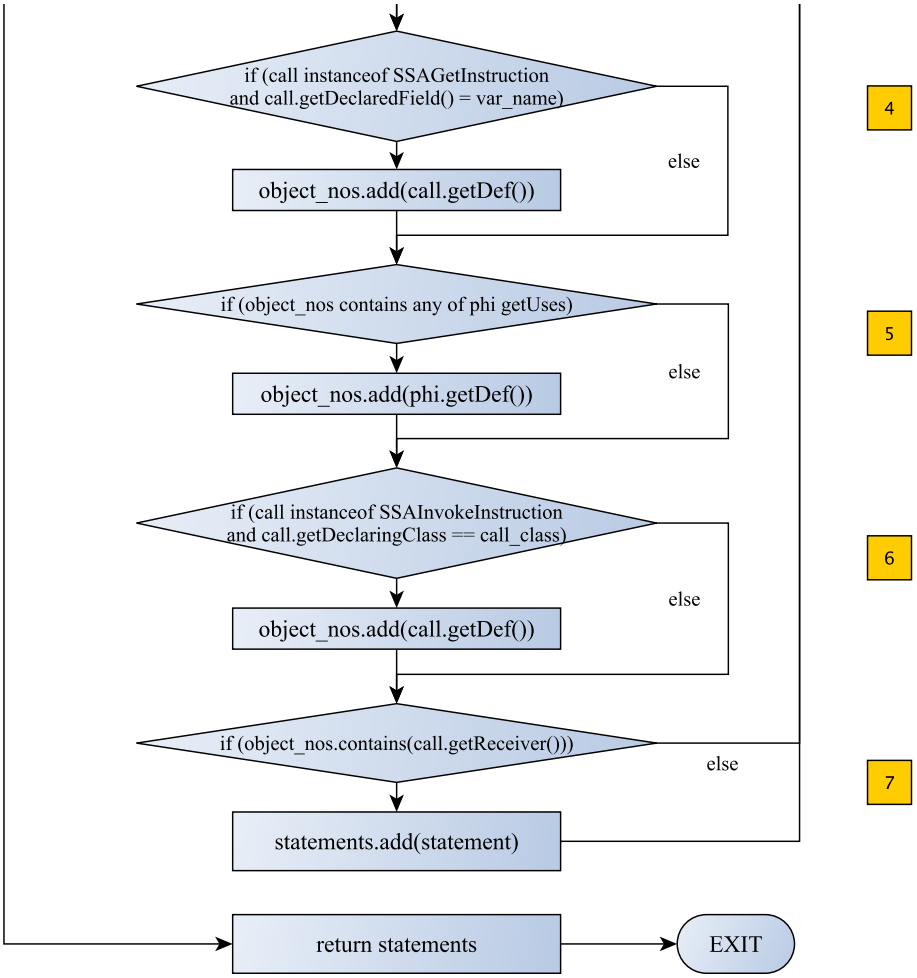

Abbildung 5.7.: Programmablaufplan für die Objektverfolgung (Teil 2)

gefundenen in `object_nos` vergleichen. Bei Fund ergänzt die Implementierung die `Statement` Liste `statements` mit dieser Anweisung (Programmablauf Schritt 7).

Sobald alle Anweisungen vom Caller Knoten durchlaufen sind, gibt die Methode die `Statement` Liste zurück, welche im Folgeschritt für das Backward-Slicing genutzt wird. Die Erläuterung des Algorithmus wurde hier zur Vereinfachung nur mit einem Caller Knoten ausgeführt, wird in der Implementierung jedoch durch die Option `multiple_caller` über eine Schleife berücksichtigt.

Individuelle Instanziierungsmethoden

Wie in der Systemfunktion SF14 beschrieben (siehe Abschnitt 5.1.1.1), nutzen Bibliotheken in einigen Fällen, bspw. `Ljavax/crypto/Cipher`, eigene Methoden zum Instanziieren/Erzeugen von Objekten. Die ursprüngliche Implementierung erkennt

© Philip Phu Dang Hoan Nguyen

zunächst Instanziierungen ausschließlich anhand der Anweisungen der Klasse `SSANew-Instruction`, weswegen Anweisungen wie `Cipher.getInstance()` nicht erkannt werden und nicht in die Pointeranalyse einbezogen werden. Eine Erweiterung gibt dem Anwender die Möglichkeit in der Konfigurationsdatei eine Liste mit Methoden zu übergeben, welche eine Instanziierung kennzeichnen.

Bei der Traversierung der Anweisungen in `findCallsTo()` kann bei `SSAInvoke-Instructions` zunächst geprüft werden, ob die Klasse, auf der eine Methode aufgerufen wird, mit der des Callee Objekts (`call_class`) übereinstimmt (siehe Abbildung 5.7 Programmablauf Schritt 6).

Ist dies der Fall, kann mit `getDeclaredTarget()` der Methodenname ausgegeben und mit den Instanziierungsmethoden der Konfigurationsdatei verglichen werden (vgl. Programmausschnitt 5.6, Zeile 1 - 6).

```
1  ...
2  Atom this_methodname = call.getCallSite().getDeclaredTarget().getName();
3  List<String> instantiation_list = Arrays.asList(prop.getProperty("object_inst_methods").split
       (","));
4  if (!instantiation_list.isEmpty()){
5    for (String inst_method : instantiation_list){
6      Atom inst_atom = Atom.findOrCreateUnicodeAtom(inst_method);
7      if  (this_methodname == inst_atom){
8        object_nos.add(call.getDef());
9        System.out.println("new object_no: " + object_no + " (" + inst_method + ") ");
10     }
11   }
12 }
```

Listing 5.6: Programmausschnitt findCallsTo()

Über die Methode `getDef()` wird abermals die neue Variablennummer ausgegeben und der Liste `object_nos` für die weitere Pointeranalyse hinterlegt (Zeile 7).

5.3.3.2. Erweiterungen Quelltext-Rekonstruktion

Wie in der Beschreibung zur Systemfunktion SF10 (Abschnitt 5.1.1.1) und zur Rekonstruktion des Quelltextes in Abschnitt 5.2.2 eingeführt, besteht der resultierende Slice zunächst ausschließlich aus einzelnen Anweisungen. Da, anders als bei den vorhergehenden Arbeiten, der Slice nicht im Originalquelltext markiert oder unterstrichen werden soll, sondern letztendlich als vollständige syntaktisch-korrekte Javadatei aus der Analyse errechnet werden soll, müssen Klassen-, Methoden- und Kontrollanweisungsköpfe, sowie -rümpfe ermittelt und bei der Quelltext-Rekonstruktion berücksichtigt werden.

Die Implementierung des *Parsers* basiert in den ersten Entwicklungsiterationen zunächst auf der Umsetzung von Detmers, nutzt wie Gulmann und Gerken die Bibliothek JavaParser und wurde aufgrund mangelnder Funktionalität später für diese Arbeit komplett neu entwickelt. Dem Parser werden der originale Quelltext und alle Zeilennummern vom Slice übergeben, damit dieser mit Hilfe von JavaParser den Quelltext zunächst in einen AST umwandeln (siehe Abschnitt 3.2.1) und anschließend über diesen iterieren kann. JavaParser bietet hierfür das Interface `Visitor`, mit dem über die Methode `visit()` durch alle Knoten eines bestimmten Typs iteriert werden kann[10].

Mit Hilfe eines Visitors wird der Quellcode geparst, indem durch das Traversieren über alle Methoden, die Slice-Liste der Zeilennummern durch die Zeilen der umgebenen Statements ergänzt und in einem letzten Schritt die reduzierte Javadatei rekonstruiert wird.

Der Algorithmus des ursprünglichen *Parsers* lässt sich folgendermaßen grob zusammenfassen:

1. Für jede Javadatei führe Parsen mit zugehöriger Slice-Liste (Zeilennummern) aus.
2. Füge erste und letzte Zeile der übergeordneten Klasse ein über `node.getParentNode().getBeginLine()` und `getEndLine()`.
3. Für jede Methode (Klasse `MethodDeclaration`) der Datei prüfe, ob irgendeine Zeilennummer aus der Slice-Liste zwischen der ersten und letzten Zeile der Methode liegt.
4. Falls ja, füge erste und letzte Zeile der Methode ein. Zusätzlich iteriere über den ganzen Körper (engl. *body*), welcher aus einem `BlockStmt` besteht, der alle Anweisungen der Methode zwischen { und } enthält. Prüfe, ob die Zeilennummer zwischen der ersten und letzten Zeile der Anweisung liegt.
5. Falls ja, füge erste und letzte Zeile der Anweisung ein und behandle diese je nach Anweisungs-Typ (`SwitchStmt, IfStmt, ForStmt, WhileStmt` oder `TryStmt`) weiter.

Durch das Einfügen der ersten und letzten Zeile der jeweiligen Anweisung soll bezweckt werden, dass der Methodenkopf inklusive Klammerung rekonstruiert wird.

Bei der Behandlung der verschiedenen Statement-Typen in Schritt 5, wird bei `ForStmt, WhileStmt` und `TryStmt` bisher die erste und letzte Zeilennummer dem

[10] `https://github.com/javaparser/javaparser/wiki/Manual`, Aufruf: 01.03.2018

Slice ergänzt, während `IfStmt` noch auf das `ElseStmt` und `ThenStmt` eingeht und bei `SwitchStmt` durch die Switch-Einträge iteriert wird.

Im Folgenden soll kurz auf die Probleme eingegangen werden, die während der Evaluation mit Einführung neuer Testfälle aufgetreten sind und iterativ behoben werden mussten.

Probleme des ursprünglichen Parsers

Um die Slice-Liste mit Klassen- oder Methodenköpfen zu ergänzen, wird auf den jeweiligen Knoten die Methode `getBeginLine()` aufgerufen. Jedoch ergaben sich bezüglich dieser Funktion bei mehreren Testfällen dieser Arbeit syntaktisch unkorrekte Slices, welchen unterschiedliche Ursachen zugrunde liegen. Zunächst wurden Konstruktoren nicht beim Parsen mit einbezogen, weswegen im Schritt 3 neben `MethodDeclaration` auch für `ConstructorDeclaration` eine zusätzliche `visit()`-Methode definiert werden muss. Ein weiteres Problem stellten Java-Annotationen dar, welche in allen Knoten an erster Zeile stehen und daher anstatt der Klassen- und Methodenköpfe im Slice angezeigt wurden, siehe Sliceausschnitt 5.7 aus dem Testfallxxx (2_clinic_sign).

```
1  @Stateful
2  public class ClinicExampleBean implements ClinicExample {
3    ...
4    private Clinician getClinician(String userID) {
5      return new Clinician(userID);
6    }
7    ...
8    @Override
9    public void sendDiagnosis(String userID, String ehrID, Key key) throws ... {
10     Clinician clinician = getClinician(userID);
11     ...
12     byte[] signedData = signData(ehr.getDiagnosis().getBytes(), clinician.getKeyName());
13     ...
14   }
15   @Override
16   public byte[] signData(byte[] data, String keyAlias) throws ... {
17     ...
18     mSignature = Signature.getInstance("SHA256withDSA", "SUN") ;
19     ...
20     return mSignature.sign();
21   }
22 }
```

Listing 5.7: Fehlerhafte Rekonstruktion aufgrund Annotationen

Der Ausschnitt zeigt, wie aus den interessanten Anweisungen wie Zeile 10, 12 oder 18 versucht wurde, die zugehörigen Methoden und die Klasse aus dem Quelltext zu parsen. In diesem Fall muss der Parser dahingehend so erweitert werden, dass zusätzlich geprüft wird, ob Annotationen (beginnend mit @) bestehen, um dann ggf. die darauffolgende Zeile ebenfalls in den Slice zu inkludieren.

Ferner berücksichtigt die Vorgehensweise nicht den Fall, dass Klassen und Methoden abhängig vom Programmierstil (Coding-Konvention) zwecks Übersichtlichkeit bspw. in mehrere Zeilen umgebrochen werden können. Der Sliceausschnitt 5.8 aus dem Testfall *TF11: OSCI* zeigt exemplarisch einen syntaktisch fehlerhaften Slice, in dem die Klassen- und Methodenköpfe unvollständig sind, da die Methode `getBeginLine()` des `JavaParsers` nur eine Zeile wiedergibt.

```
1  public class Crypto
2  {
3    public static byte[] doRSADecryption(Key key, byte[] data)
4    throws OSCICipherException, NoSuchAlgorithmException
5    {
6      return doRSADecryption(key, data, Constants.ASYMMETRIC_CIPHER_ALGORITHM_RSA_1_5,...);
7    }
8    public static byte[] doRSADecryption(Key key,
9    byte[] data,
10   String algorithm,
11   String mgfAlgorithm,
12   String digestAlgorithm,
13   byte[] oaepParams)
14   {
15     try
16     {
17       if (DialogHandler.getSecurityProvider() == null)
18       cipher = javax.crypto.Cipher.getInstance(Constants.JCA_JCE_MAP.get(algorithm));
19       else
20       cipher = javax.crypto.Cipher.getInstance(Constants.JCA_JCE_MAP.get(algorithm),
21       DialogHandler.getSecurityProvider());
22       ...
23       return cipher.doFinal(data);
24     }
25   }
26 }
```

Listing 5.8: Fehlerhafte Rekonstruktion aufgrund verschiedener Programmierstile

Hierbei fallen nicht nur Methoden wie `doRSADecryption()` in Zeile 3 und 8 auf, welche ihre Ergänzungen der `throws` bzw. Parameter aus Übersichtlichkeit einzeln in neuen Zeilen auflisten, sondern ebenfalls generell die fehlende blockeinweisenden Klammerung „{“ in Zeile 2, 5, 14 und 16, welche abhängig vom Programmierstil ebenfalls in eine neue Zeile umgebrochen werden kann. Auch hier liefert das Ergänzen des `getBeginLine()` der Knoten kein hinreichendes Ergebnis bei der Rekon-

© Philip Phu Dang Hoan Nguyen

struktion, zumal durch falsche Klammerung die Syntax von Java inkorrekt ist und somit bspw. bei Anzeigen über Editoren kein korrektes Syntax-Highlighting dargestellt werden kann. Letzte Auffälligkeit in diesem Sliceausschnitt betrifft Zeile 19, in welcher die `else`-Anweisung im Slice fehlt, obwohl die Zeile 20 Teil des Slices ist. Die Ursache hierfür ist auf eine inkorrekte Behandlung des `ElseStmts` zurückzuführen, wobei zudem die `if-else`-Verzweigung in einem „unkonventionellen“ Programmierstil ohne Klammerung vorliegt und ebenfalls gesondert behandelt werden muss.

Grundsätzlich fehlten bei der Behandlung der unterschiedlichen Anweisungen nicht nur bestimmte Typen (`IfStmn` ohne `ElseStmn`-Anweisungen führten zu Exceptions oder `CatchStmn` wurden komplett ignoriert), sondern ebenso eine gewisse Genauigkeit bzw. Tiefe beim Iterieren der ASTs. Dass innerhalb der Blöcke in den Methoden weitere Verschachtelungen bestehen können, wurde nicht berücksichtigt und stattdessen wurde lediglich eine Ebene tief geparst.

Die neue Implementierung des Parsers löst die aufgezeigten Probleme, funktioniert über einen rekursiven Aufruf und wird im Folgenden kurz erläutert.

Zur Evaluation dieser Problematiken, wurden die schlichteren Testfälle *TF08: Coding-Konventionen* und *TF09: Tief-verschachtelter Callee* erstellt, welche im Abschnitt 6.2.8 und 6.2.9 erläutert werden.

Implementierung Parser

Bevor die Funktionsweise des Parsers erläutert wird, soll zunächst kurz auf den AST und die Knotentypen des *JavaParsers* eingegangen werden.

Folgendes Beispielprogramm sei gegegeben:

```
1  package com.github.javaparser;
2
3  import java.time.LocalDateTime;
4
5  public class TimePrinter {
6
7    public static void main(String args[]){
8      System.out.print(LocalDateTime.now());
9    }
10 }
```

Listing 5.9: Beispielprogramm JavaParser (Quelle: [Smith u. a. 2017])

Das Umwandeln des Beispielprogramms 5.9 ergibt einen gerichteten AST, in dem alle Knoten von einem Wurzelknoten, der `CompilationUnit` ausgehen, welches einer Java Klasse entspricht. Nach der Definition eines *Baumes* besitzen alle Knoten (bis auf den Wurzelknoten) genau eine eingehende Kante vom Vaterknoten (engl. parent node), während die Knoten allen möglichen Anweisungen entsprechen. Mit Hilfe des AST lassen sich somit verschachtelte Programme gezielt traversieren.

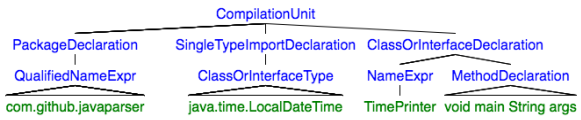

Abbildung 5.8.: Compilation Unit (Quelle: [Smith u. a. 2017])

Die Abbildung 5.8 zeigt den resultierenden Baum mit der `CompilationUnit` und den jeweiligen Kindern. Die `ClassOrInterfaceDeclaration` entspricht hierbei der Klasse, welche in diesem Beispiel eine Methode `main()` vom Typ `MethodDeclaration` besitzt. Die Notation mit dem Dreieck unterhalb des Knotens soll darauf hindeuten, dass der Kinderknoten nur grob dargestellt werden.

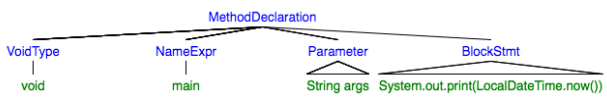

Abbildung 5.9.: Method Declaration (Quelle: [Smith u. a. 2017])

Abbildung 5.9 zeigt eine detailliertere Struktur der `MethodDeclaration`, in der die Kinderknoten nicht nur den Namen, die Parameter und den Typ, sondern auch den Methodenrumpf in Form eines `BlockStmts` beschreiben. Dieser Anweisungs-Typ enthält weitere Anweisungen zwischen der Klammerung „{“ und „}“, welche über `getStatements()` ausgegeben werden können.

Abhängig vom Typ, anders ausgedrückt welcher spezifizierten Klasse die Anweisung zugehört und welche Interfaces implementiert sind, verfügt diese über unterschiedliche Unterknoten, welche beim Parsen differenziert behandelt werden müssen. So besteht eine Anweisung der Klasse `IfStmt` in jedem Fall aus einer Bedingung (`Expression`) und den beiden Knoten `thenStmt` für den Fall, dass die Bedingung zutrifft, und `elseStmt` für den Fall, dass die Bedingung nicht zutrifft. In jedem Fall müssen die existenten Kanten weiter traversiert werden. Ähnlich wie bei `MethodDeclaration` verfügen auch Anweisungen wie `DoStmt, ForeachStmt, ForStmt` und `WhileStmt` durch das Nutzen des Interfaces `NodeWithBody` einen Körper, welcher (sofern mit Klammerung definiert) ebenfalls einem `BlockStmt` entspricht. In diesem Fall kann die Methode `getBody()` genutzt werden, welche für das weitere Traversieren und die korrekte Rekonstruktion der Anweisungsköpfe und der Klammerung eine große Rolle spielt.

Der Algorithmus des neuen Parsers lässt sich folgendermaßen zusammenfassen:

1. Für jede Javadatei führe Parsen mit zugehöriger Slice-Liste (Zeilennummern) aus.

2. Füge erste und letzte Zeile der übergeordneten Klasse ein über `n.getParentNode().getBeginLine()` und `getEndLine()` ein, beachte Annotationen und füge alle Zeilen von der ersten Zeile der Klasse bis zum ersten Kindknoten ein.

3. Für jede Methode (Klasse `MethodDeclaration`) der Datei prüfe, ob irgendeine Zeilennummer aus der Slice-Liste zwischen der ersten und letzten Zeile der Methode liegt.

4. Falls ja, füge erste und letzte Zeile der Methode ein. Füge zudem alle Zeilen von von der ersten Zeile der Methode bis zur ersten Zeile des `BlockStmts` ein. Rufe die Methode `getStatementBody()` mit dem Knoten (Methode), der Zeilennummer und Slice-Liste auf, welche rekursiv alle relevanten Knoten traversiert, nach Zeilenposition untersucht und die ergänzte Slice-Liste ausgibt.

5. In `getStatementBody()`: Iteriere über den ganzen Körper (`BlockStmt`) der Methode und prüfe, ob die Zeilennummer zwischen der ersten und letzten Zeile einer Anweisung liegt.

6. Falls ja, füge erste und letzte Zeile der Anweisung ein, behandle den Anweisungs-Typ und rufe ggf. die Methode `getStatementBody()` dementsprechend erneut auf die Kinderknoten auf.

Folgender Programmausschnitt 5.10 soll zur Erläuterung dienen und einen kurzen Einblick in die Implementierung des rekursiven Aufrufs gewähren:

```
1  public Set<Integer> getStatementBody(Node node, int line, Set<Integer> exeSlice){
2    ...
3    if(node.getBeginLine() <= line && node.getEndLine() >= line){
4      exeSlice.add(node.getBeginLine());
5      exeSlice.add(node.getEndLine());
6      if(node instanceof ForStmt){
7        ForStmt forStatement = (ForStmt) node;
8        exeSlice.add(forStatement.getBody().getBeginLine());
9        exeSlice.add(forStatement.getBody().getEndLine());
10       if(forStatement.getBody() instanceof BlockStmt){
11         exeSlice.addAll(getStatementBody(forStatement.getBody(),line,exeSlice));
12       }
13     }
14     if(node instanceof IfStmt){
15       ...
16     }
17     if(node instanceof TryStmt){
18       ...
19     }
20     if(node instanceof CatchStmt){
21       ...
22     }
23     if (node instanceof BlockStmt){
24       BlockStmt blockstmt = (BlockStmt) node;
25
26       for (Statement stmt:blockstmt.getStmts()){
27         Node blocknode = (Node) stmt;
28         exeSlice.addAll(getStatementBody(blocknode,line,exeSlice));
29       }
30     }
31   }
32   return exeSlice
33 }
```

Listing 5.10: Programmausschnitt vom Parser (getStatementBody())

Die Methode `getStatementBody()` in Zeile 1 erhält als Parameter einen Knoten, die Programmzeile, welche untersucht wird, und eine Integer-Liste des Datentyps `Set`, welche einer ungeordneten Liste entspricht, in der jedes Element einmalig vorkommt. Die Liste enthält anfangs alle Programmzeilen, die aus dem Backward-Slicing resultieren und wird über einen rekursiven Aufruf ergänzt und zurückgegeben. Wie bereits erläutert, müssen die Knoten entsprechend des Anweisungs-Typs unterschiedlich behandelt werden, da die weitere Verzweigungen nach Typ verschieden sind

© Philip Phu Dang Hoan Nguyen

(`IfStmt` entspricht `thenStmt` und `elseStmt` oder `ForStmt` entspricht `BlockStmt`). Dies geschieht über die `instanceof`-Abfrage und wird in diesem Beispiel für `IfStmt`, `TryStmt`, und `CatchStmt` lediglich angedeutet, allderdings exemplarisch für `ForStmt` in Zeile 6 vollständig angezeigt. Ist der Knoten eine `for`-Schleife, wird die erste und letzte Zeilennummer der Anweisung der Liste hinzugefügt und sofern die Schleife mit Klammerung definiert wurde, die Rekursionstiefe um eine Ebene erhöht. Dafür wird `getStatementBody()` erneut aufgerufen mit dem `BlockStmt` als Knoten (Vergleich Zeile 11). Die Behandlung der anderen Typen in den Zeilen 14, 17 und 20 verlaufen dementsprechend ähnlich.

Die Behandlung der `BlockStmt`-Knoten wird zwischen Zeile 23 und 30 verdeutlicht, in der über alle Anweisungen des Blocks über `getStmts()` iteriert und ebenfalls `getStatementBody()` aufgerufen wird.

Durch die Unteraufrufe innerhalb derselben Methode `getStatementBody()` errechnen sich durch alle Verschachtelungsebenen der Anweisungen rekursiv alle Zeilennummern, die einer Anweisung zugehörig sind.

Die Problematik der Erkennung der Klassen- und Methodenköpfe, welche durch mehrzeilige Klassenköpfe oder das Umbrechen der blockeinweisenden Klammerung „{“ in eine neue Zeile erschwert werden, wurde in Schritt 2 und 4 gelöst. Da bei Benutzung von JavaParser am Knoten nicht erkennbar ist, ob Umbrüche gemacht worden sind und `getBeginLine()` nur die allererste Zeilennummer der Anweisung ausgibt, wurde ein Workaround implementiert, welcher von dieser Zeile bis zur `BeginLine` des nächsten Knotens alle Zeilen in die Slice-Liste aufnimmt. Der Programmausschnitt 5.11 zeigt exemplarisch, wie der Klassenkopf inklusive „{“ geparst wird.

```
1  public void visit(MethodDeclaration n, Object arg) {
2    ...
3    if(n.getParentNode().toString().startsWith("@")){
4      first_line = n.getParentNode().getBeginLine()+1;
5    } else{
6      first_line = n.getParentNode().getBeginLine();
7    }
8    exeSlice.add(first_line);
9
10   // Add all lines between class and first node (Fix for backets in nextline)
11   List<Node> children = n.getParentNode().getChildrenNodes();
12   if (!children.isEmpty()){
13     List<Integer> lines = IntStream.rangeClosed(first_line, children.get(0).getBeginLine()-1).
           boxed().collect(Collectors.toList());
14     exeSlice.addAll(lines);
15   }
16   ...
```

Listing 5.11: Programmausschnitt vom Parser (visit() Teil 1)

In diesem Fall werden alle Zeilennummern bis zum ersten Kinderknoten hinzugefügt (vgl. Zeile 11-14). Analog dazu wird bei Methoden (siehe Programmausschnitt 5.12) durch alle Anweisungen iteriert und geprüft, ob diese die gesuchten Zeilennummern umschließen. Ist dies der Fall, werden nicht nur die erste und letzte Zeilennummer der Anweisung, sondern auch die jeweiligen Zeilennummern des Körpers (`BlockStmt`) der Liste in Zeile 5-9 ergänzt. Um mehrzeilige Methodenköpfe zu unterstützen, werden zusätzlich ebenfalls alle Zeilen zwischen der ersten Zeile der Methode und der ersten Zeile des Körpers ergänzt, siehe Zeile 12-13.

```
1  ...
2  for(Node node: nodes){
3    for(Integer line: inSlice){
4      if(node.getBeginLine() <= line && node.getEndLine() >= line){
5        exeSlice.add(n.getBeginLine());
6        exeSlice.add(n.getEndLine());
7        exeSlice.add(n.getBody().getBeginLine());
8        exeSlice.add(n.getBody().getEndLine());
9
10       // Add all lines between method and first brackets (Fix for multiple line heads)
11       List<Integer> lines = IntStream.rangeClosed(n.getBeginLine(), n.getBody().getBeginLine
             ()).boxed().collect(Collectors.toList());
12       exeSlice.addAll(lines);
13
14       exeSlice.addAll(getStatementBody(node, line, exeSlice));
15       ...
```

Listing 5.12: Programmausschnitt vom Parser (visit() Teil 2)

Einzige Einschränkung hierbei ist, dass Kommentare, die ursprünglich durch `JavaParser` ignoriert werden, in den Slice geraten können (bspw. in Zeile 10). Gleiches gilt bspw. für Javadoc-Kommentare, welche ebenfalls als Teil der Klassen- oder Methodenknoten gelten und beim Parsen somit nicht von der eigentlichen Implementierung differenzierbar sind. Da eine korrekte Syntax für eine Analyse essentiell ist und Kommentare keine syntaktischen Auswirkungen haben, wird diese Einschränkung in Kauf genommen.

5.3.3.3. Konfigurationsmöglichkeiten

Wie bereits erläutert, bedarf Slicing generell ein gewisses Maß an Flexibilität, damit für jeden Anwendungsfall in akzeptierbarer Zeit ein sinnvoller Slice berechnet werden kann. Diese Flexibilität resultierte über die inkrementelle iterative Entwicklung in Form verschiedener Konfigurationsmöglichkeiten, die in diesem Kapitel beschrieben (siehe Abschnitt 5.1.1.1 und Abschnitt 5.2.1) und im Folgenden nochmals aufgelistet werden sollen.

© Philip Phu Dang Hoan Nguyen

Alle Konfigurationsmöglichkeiten werden durch die Exclusion- und Konfigurationsdatei gegeben, die der Auditor vor dem Slicing einliest. Die verfügbaren Optionen der Konfigurationsdatei sind in der Tabelle 5.1 aufgelistet und kurz erläutert. Fettgedruckte Optionen sind verpflichtend anzugeben, während die restlichen Optionen optional sind und einen Defaultwert (`false` oder leer) besitzen.

Eine Beispiel-Konfigurationsdatei ist dem Anhang im Abschnitt B.1 beigefügt.

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

Tabelle 5.1.: Optionen der Konfigurationsdatei

Bei größeren Anwendungen empfiehlt es sich in jedem Fall, zunächst als `mainclass` die Klasse anzugeben, in dem sich der Caller befindet, um zu prüfen, ob dieser gefunden werden kann.

Ansonsten sollte eine möglichst genaue Angabe gemacht werden, da dies maßgeblichen Einfluss auf die Ermittlung der Entrypoints hat und sich somit später ebenfalls auf die Größe des Callgraphs stark auswirkt. Dadurch kann zunächst sichergestellt werden, dass das Slice-Kriterium mit Angabe des Callers und Callee gefunden wird. Der Testfall *TF11: OSCI* (siehe Abschnitt **??**) behandelt diese Problematik, welche hier nicht weiter erläutert werden soll. Ebenfalls kann auf die Erzeugung der PDFs bei größeren Anwendungen verzichtet werden, da diese ohnehin schnell zu unübersichtlich werden und bei der Erzeugung der PDF-Datei zum SDG den Speicher schnell zum Überlaufen bringen können.

6. Untersuchungen und Evaluierung

Aus der iterativen Entwicklung des Auditors gingen diverse Erkenntnisse und Erweiterungen hervor, die mit Hilfe von Testfällen evaluiert und auf Funktionalität verifiziert werden müssen. Die Evaluierung wird in folgendem Kapitel vorgestellt und erfolgt in zwei Teilen. Nachdem kurz auf die Vorgehensweise eingegangen wird, werden im ersten Teil der Evaluation in Abschnitt 6.2 kleinere Testfälle vorgestellt, welche aufgetretene Probleme erläutern und die dazu implementierten Erweiterungen prüfen sollen. Es folgt der zweite Teil der Evaluierung (Abschnitt 6.3), in dem die korrekte Funktionsfähigkeit des Auditors mit Hilfe von zwei größeren Anwendungen mit Sicherheitsmechanismen getestet werden soll. Fokus soll in diesen Testfällen zum einen auf der Skalierbarkeit und zum anderen auf den Inhalten der Sicherheitsanalyse liegen.

6.1. Vorgehen

Wie bereits in der Einleitung erwähnt, beschreibt der erste Teil der Evaluierung Testfälle, welche während der Entwicklung definiert und getestet wurden. Diese bilden die Basis für den zweiten Teil, da nahezu alle Erweiterungen ebenfalls beim Slicing größerer Anwendungen eine Rolle spielen. Während der Entstehung der Arbeit konnten zwei Open-Source Bibliotheken als Testfälle identifiziert werden, die einen Sicherheitsbezug haben:

Openkeepass: Eine Bibliothek zur Passwortverwaltung.

OSCI: Eine Bibliothek mit Netzwerkprotokollen für die öffentliche Verwaltung.

Weitere Beschreibungen zu diesen Bibliotheken sind aus den jeweiligen Testfällen in Abschnitt 6.3 zu entnehmen.

Ziel ist es, mit einer erfolgreichen Anwendung des Auditors auf diese größeren Anwendungen zum einen den Bezug zu den sicherheitskritischen Aspekten herzustellen und zum anderen die Skalierbarkeit zu testen. Die Testfälle des ersten Teils behandeln bis auf die EJB Klinik-Anwendung (ca. 200 Programmzeilen (engl. lines of code)) kleinere selbstgeschriebene Anwendungen. Da diese aus einer oder wenigen

Klassen bestehen, verhält sich die Datenflussanalyse bezüglich der Ausführungszeit in allen Fällen performant. Erst bei größeren echten Anwendungen steigt die Komplexität und somit beispielsweise die Anzahl der interprozeduralen Datenflüsse, welche die statische Programmanalyse an seine Grenzen bringen kann.

Bei der Beschreibung der Testfälle wird in allen Fällen zunächst eine Übersicht der wichtigsten Parameter aus der Konfigurationsdatei gegeben (mit u.a. Caller, Callee, Daten- und Kontrollabhängigkeitsoptionen). Das *Exclusionsfile* wurde für alle Testfälle von Detmers übernommen (vgl. Anhang Abschnitt B.2).

Daraufhin wird der Testfall mit Nutzen und teilweisen Bezug auf die funktionalen Anforderungen in Abschnitt 5.1.1.1 erläutert. Während der vollständige Quellcode in vielen Fällen im Anhang referenziert wird (Abschnitt B.3), werden die Slicing-Ergebnisse mit Angabe der Daten- und Kontrollabhängigkeitsoptionen ausschnittsweise gezeigt und beschrieben. Die beiden Slicing-Optionen werden hierbei im Text in der Reihenfolge `DATENABHÄNGIGKEIT KONTROLLABHÄNGIGKEIT` aufgeführt und die Option `NO_EXCEPTIONAL_EDGES` zur Übersichtlichkeit auf `NO_EXC_E` gekürzt. Gleiches gilt für die Optionen `NO_EXCEPTIONS (NO_EXC)` und `NO_HEAP_NO_EXCEPTIONS (NH_NX)`. Die Angabe der `mainclass` ist bei den meisten Testfällen nicht verpflichtend, da durch die geringe Komplexität der Testanwendungen die Ermittlung der Entrypoints ebenso unspezifiziert effizient durchgeführt werden kann (siehe Abschnitt 5.2.2). Ferner besitzen die meisten Testanwendungen ohnehin lediglich eine oder zwei Klassen, welches somit bezüglich der Anzahl der Entrypoints und der Größe des Callgraphs nur einen marginalen Unterschied macht. Daher wird diese Option in den meisten Fällen unter den weiteren Optionen in Klammern aufgeführt.

Sollten Testfälle ihren Ursprung aus Mängeln der alten Implementierung haben, werden zusätzlich die aufgetretenen Probleme dargelegt und auch fehlerhafte Slices beigefügt.

Im zweiten Teil der Evaluierung wird, wie in der Einleitung beschrieben, abseits der reinen Funktionalität, ebenfalls ein Augenmerk auf die Inhalte der Slicing-Ergebnisse gelegt. Dabei soll nicht mehr auf die Optimierungen des Slicers eingegangen werden, die implementiert wurden, sondern über die Konfigurationsmöglichkeiten die Skalierbarkeit des Auditors geprüft werden.

Es soll dementsprechend geprüft werden, ob der Auditor zielgerichtet auf größere Anwendungen angewendet und anhand dieser Ergebnisse tatsächlich Sicherheitsanalysen durchgeführt werden können.

 © Philip Phu Dang Hoan Nguyen

6.2. Evaluierung Teil I

6.2.1. TF01: Arithmetisches Slicing

Jar:	`ArithmeticTest.jar`
Caller:	`main`
Callee:	`println`
Datenabhängigkeit:	`FULL`
Kontrollabhängigkeit:	`NO_EXCEPTIONAL_EDGES, NONE`
Weitere Optionen:	(`mainclass = LArithmeticTest`)

Der Testfall *TF01* soll zunächst als einfacher Funktionstest des Slicers dienen. Die Klasse `ArithmeticTest` verfügt nur eine main-Methode, in welcher die drei Variaben (des Datentyps `int`) `value, value2` und `someOtherValue` arithmetisch verarbeitet werden und am Ende `value` und `value2` über `println()` ausgegeben werden (vgl. vollständiger Quellcode B.3).

In diesem Testfall wird `value` von keiner anderen Variable beeinflusst, während `value2` mit `someOtherValue` addiert wird.

Zunächst wird der Slice 6.1 mit der ursprünglichen Implementierung ohne Erweiterungen mit `FULL NO_EXC_E` erstellt.

```
1  public class ArithmeticTest {
2    public static void main(String[] argsv){
3      value += value;
4      someOtherValue -= someOtherValue;
5      while(someOtherValue < 2){
6        someOtherValue++;
7      }
8      while(value < 2){
9        value*= 2;
10     }
11     System.out.println(value);
12   }
13 }
```

Listing 6.1: Testfall Arithmetisches Slicing - Vollständiger Slice FULL NO_EXC_E (keine Optimierungen)

Zunächst fällt auf, dass mit `println(value)` lediglich ein Callee in das Slicing eingeflossen ist und somit nur die Datenflüsse zu dieser Anweisung und der Variable `value` analysiert werden. Der Slicer erkennt als Folge dessen Datenflüsse der Variable `value` wie bspw. in Zeile 3, 8 und 9. Die Zeilen 4 bis 7 rund um `someOtherValue` erscheinen

im Slice, da von der `while`-Schleife in Zeile 5 und dieser Variable weitere Anweisungen (kontroll-)abhängig sind. Wird der Auditor mit der Kontrollabhängigkeitsoption `NONE` aufgerufen, werden diese Anweisungen in der Konsequenz ignoriert:

```
1  public class ArithmeticTest {
2
3    public static void main(String[] argsv){
4      value += value;
5      while(value < 2){
6        value*= 2;
7      }
8      System.out.println(value);
9    }
10 }
```

Listing 6.2: Testfall Arithmetisches Slicing - Vollständiger Slice FULL NONE (keine Optimierungen)

Durch die Caller- und Callee-Erweiterung des Slicers (siehe Abschnitt 5.3.3.1) ist der Auditor in der Lage mehrere Callee Anweisungen zu slicen (vgl. Slice 6.3). Eine weitere Anwendung, die sich ebenfalls dieser Funktion SF08 (Abschnitt 5.1.1.1) widmet, folgt im nächsten Testfall. Der Slice entspricht darüber hinaus den Erwartungen und zeigt, dass der WALA in diesem Fall korrektes Slicing ausführt. Die Ausgaben des Auditors zu diesem Testfall sind ebenfalls im Anhang B.4 zu finden und zeigen, dass zum Slicing-Prozess nützliche Informationen zu den Teilschritten wie bspw. Ermittlung Entrypoints, Caller/Callee, Slice oder Ausführungszeiten geliefert werden (vgl. SF02 Abschnitt 5.1.1.1).

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

Listing 6.3: Testfall Arithmetisches Slicing - Vollständiger Slice FULL NONE

© Philip Phu Dang Hoan Nguyen

6.2.2. TF02: Mehrfache Callee-Anweisungen

Jar:	`multiplecalleetest.jar`
Caller:	`main`
Callee:	`some_method`
Datenabhängigkeit:	`FULL`
Kontrollabhängigkeit:	`NO_EXCEPTIONAL_EDGES`
Weitere Optionen:	`(mainclass = LMultipleCalleeTest), (src_callee_class = ClassB), (create_pdfs = true, sdg_pdf = true)`

Der Testfall *TF02* knüpft an die Anforderung der SF08 an, die im Testfall TF01 kurz aufgegriffen wurde. Das Beispielprogramm verfügt in diesem Fall die drei Klassen `ClassA, ClassB` und `ClassC`, welche alle eine gleichnamige Methode `some_method()` verfügen, welche lediglich andere Zeichenketten zurückgeben (vgl. vollständiger Quellcode B.3). In der main-Methode der Klasse `MultipleCalleeTest` werden nun zu jeder der drei Klassen Objekte instanziiert und jeweils die Methode `some_method()` aufgerufen.

Wie im Testfall TF01 aufgezeigt, würde die ursprüngliche Implementierung des Slicers den ersten Aufruf als Callee für das Slicing einbeziehen, während durch die Erweiterung alle drei Callees erkannt werden (vgl. Slice 6.4).

```
1  public class MultipleCalleeTest {
2
3    public static void main(String[] argsv){
4      ClassA object_a = new ClassA();
5      ClassB object_b = new ClassB();
6      ClassC object_c = new ClassC();
7      String text_a = object_a.some_method();
8      String text_b = object_b.some_method();
9      String text_c = object_c.some_method();
10   }
11 }
```

Listing 6.4: TF02: Mehrfache Callee-Anweisungen - Slice FULL NO_EXC_E (ohne Angabe der src_callee_class)

Da es, wie in der Anforderung in SF08 (siehe Abschnitt 5.1.1.1) beschrieben, sinnvoll sein kann eine feste Klasse festzulegen, von dem dieser Callee Methodenaufruf stammt, wurde diese Erweiterung implementiert und u.a. mit diesem Testfall getestet.

Mit Angabe der `src_callee_class` in der Konfigurationsdatei wird beim Suchen des Callees nach der gewünschten Klasse gefiltert (hier bspw. `ClassB`) und somit der Slice auch nur mit dieser Anweisung gebildet (vgl. Slice 6.5).

```
1 public class MultipleCalleeTest {
2
3   public static void main(String[] argsv){
4     ClassB object_b = new ClassB();
5     String text_b = object_b.some_method();
6   }
7 }
```

Listing 6.5: TF02: Mehrfache Callee-Anweisungen - Slice FULL NO_EXC_E (mit Angabe der src_callee_class)

Um den Anforderung aus der Systemfunktion SF12 (siehe Abschnitt 5.1.1.1) nachzukommen, werden die bei diesem Testfall erzeugten PDF-Ausgaben im Anhang im Abschnitt B.4.1 beigelegt. Während die Ausgaben zum Callgraph bei einfachen Testfällen noch lesbar sind, verfügt die PDF-Datei zum kompletten SDG über 50 Knoten und ist für diese Arbeit, aufgrund der Größe des Graphen, nicht sinnvoll darstellbar. Mit Hilfe der Suchfunktion innerhalb des PDF-Viewers, ist es allerdings möglich interessante Knoten zu finden und näher zu verfolgen. Die Abbildung B.3 im Anhang zeigt einen kleinen Ausschnitt des SDG, in dem u.a. die Erzeugung des `ClassB`-Objekts und der Methodenaufruf `some_method()` zu erkennen sind. Ferner werden bei jeder Ausführung des Auditors der Callgraph, der SDG und die IR ebenfalls für eine weitere Analyse in textueller Form im Ausgabeverzeichnis abgelegt.

Generell konnten die PDF-Dateien mit einigen Ausnahmen alle Testfälle des ersten Teils der Evaluierung erfolgreich erzeugt werden. Für die EJB Klinikanwendung der beiden Testfälle *TF06* und *TF07* konnten keine SDG-PDFs erzeugt werden, da der SDG] in beiden Fällen über 40.000 Knoten und über 60.000 Kanten enthielt. Im Rahmen dieser Arbeit wird im Folgenden kein weiterer Bezug auf die Erläuterungen der PDFs genommen.

6.2.3. TF03: Mehrfache Caller-Anweisungen

Jar:	`multiplecallertest.jar`
Caller:	`entry_method`
Callee:	`some_method`
Datenabhängigkeit:	`FULL`
Kontrollabhängigkeit:	`NO_EXCEPTIONAL_EDGES`
Weitere Optionen:	`(mainclass = MultipleCallerTest), (multiple_caller = true)`

Der Testfall *TF03* handelt ebenfalls von der Problematik der Nichteindeutigkeit von Methodennamen in Anwendungen, in diesem Fall bei der Angabe der Caller-Methode. Die Anforderungen wurden in Beschreibungen zu der Systemfunktion SF07 (siehe Abschnitt 5.1.1.1) aufgeführt und sollen u.a. mit diesem Testfall geprüft werden.

Verfügt eine Klasse über mehrere gleichnamige Methoden, handelt es sich in der Regel um überladene Methoden oder Konstruktoren, in denen entweder unterschiedliche Datentypen oder eine andere Anzahl an Parametern übergeben wird. Das Beispielprogramm mit der Klasse `MultipleCallerTest` simuliert diesen Umstand mit einer Methode `entry_method()`, die entweder mit einem oder zwei Zeichenketten ausgeführt werden kann und somit doppelt definiert ist (vgl. vollständiger Quellcode B.6). Beide Methoden nutzen den Methodenaufruf `some_method()`, der für diesen Testfall als Callee festgelegt wird.

Der resultierende Slice 6.6 zeigt, dass ein Caller-Knoten ignoriert wird, da bei der ursprünglichen Implementierung bei Fund des Methodennamens dieser `CGNode` direkt als Caller zurückgegeben wird. Eine interessante Erkenntnis ist in diesem Zusammenhang, dass beim Slicing über den Caller `entry_method()` nicht ausschließlich die Methode selbst im Slice enthalten ist, sondern ebenso der Aufruf dessen in der Methode `main()` (Zeile 7 - 9), da diese einen Entrypoint bildet.

```
1  public class MultipleCallerTest {
2
3    public static String entry_method(String value) {
4      ClassA obj = new ClassA();
5      String result = value + obj.some_method();
6    }
7    public static void main(String[] argsv){
8      String result = entry_method(value);
9    }
10 }
```

Listing 6.6: TF03: Mehrfache Caller-Anweisungen - Slice FULL NO_EXC_E (ohne Angabe multiple_caller (default: false))

Durch die Erweiterungen des Slicers, beschrieben in Abschnitt 5.3.3.1, ist es mit Angabe des Schalters `multiple_caller` in der Konfigurationsdatei möglich, dieses Verhalten zu ändern und den kompletten Callgraph nach mehreren Caller-Knoten zu durchsuchen. Der Slice 6.7 zeigt das Ergebnis zu diesem Testfall für den Fall, dass explizit `multiple_caller` mit `true` belegt ist. Dadurch wird neben der überladenen Methode ebenfalls der Aufruf `entry_method(value, value2)` in `main()` (Zeile 13) dem Slice ergänzt.

```
1  public class MultipleCallerTest {
2
3    public static String entry_method(String value) {
4      ClassA obj = new ClassA();
5      String result = value + obj.some_method();
6    }
7    public static String entry_method(String value, String value2) {
8      ClassA obj = new ClassA();
9      String result = value + obj.some_method();
10   }
11   public static void main(String[] argsv){
12     String result = entry_method(value);
13     result += entry_method(value,value2);
14   }
15 }
```

Listing 6.7: TF03: Mehrfache Caller-Anweisungen - Slice FULL NO_EXC_E (mit Angabe multiple_caller = true)

6.2.4. TF04: Exception-Kontrollfluss

Jar:	`noexcedgestest.jar`
Caller:	`entry_method`
Callee:	`important`
Datenabhängigkeit:	`FULL`
Kontrollabhängigkeit:	`FULL` (~2,5 ms), `NO_EXCEPTIONAL_EDGES` (~2,7 ms)
Weitere Optionen:	`(mainclass = LNoExcEdgesTest)`

Der Testfall *TF04* entstand bei der Evaluation der Kontrollabhängigkeitsoptionen und soll die gewonnen Erkenntnisse dazu bestätigen (siehe Abschnitt 5.2.4). Die Beispielanwendung hat mit der Klasse `NoExcEdgesTest` in ihrer Methode `entry_method()` unterschiedliche Anweisungen definiert, die Exception-Kontrollflüsse bilden, indem sie bei der Ausführung potentiell eine *Exception* werfen können.

Unter den Anweisungen befindet sich nicht nur eine eigens definierte Methode `divide()`, welche den Datenfluss umlenken bzw. terminieren lassen könnte (bspw.

© Philip Phu Dang Hoan Nguyen

bei Division durch Null), sondern ebenso Java Wrapper-Klassen wie `Integer` und `Float`, die eine `NumberFormatException`[11] werfen können, sollten keine lesbaren Zahlen übergeben werden (vgl. vollständiger Quellcode B.7). Die genannten Anweisungen, sowie andere arithmetische Berechnungen mit den primitiven Typen `int` und `float` sollen in diesem Testfall unmissverständlich nichts mit dem Callee `important()` zu tun haben.

Zunächst wird das Slicing der Beispielanwendung mit den Optionen `FULL FULL` durchgeführt (vgl. Slice 6.8).

```
1  public class NoExcEdgesTest {
2
3    public static Integer entry_method(String value) throws Exception {
4      Integer wr_int = 0;
5      wr_int += 1;
6      wr_int += 2;
7      Float wr_float = (float) 0.1;
8      wr_float = wr_float +1;
9      wr_float = wr_float +2;
10     float other_float = divide(10,2);
11     Integer result = important(10);
12   }
13   public static void main(String[] argsv) throws Exception{
14     Integer result = entry_method(value);
15   }
16 }
```

Listing 6.8: TF04: Exception-Kontrollfluss - Slice FULL FULL

Den Erwartungen entsprechend, sind Anweisungen mit den Wrapper-Klassen `Integer wr_int` und `Float wr_float`, sowie der Aufruf `divide(10,2)` in der Zeile 4 bis 10 im Slice enthalten, obwohl diese vom Daten- und dem klassischem Kontrollfluss nicht mit dem Callee in Verbindung stehen. Die Anweisungen mit dem primitiven `int pr_int` und dem `float pr_float` sind im Gegenzug, wie zu erwarten, nicht Teil des Slices.

Beim Slicing der Beispielanwendung mit `FULL NO_EXC_E` fallen im Gegensatz dazu die Exception-Kontrollflüsse und somit ebenfalls die hier nicht relevanten Anweisungen komplett weg (vgl. Slice 6.9). Da die Kontrollabhängigkeitsoption `NO_EXCEPTIONAL_EDGES` mehr intuitive und sinnvollere Ergebnisse liefert und viele unnötige Absprünge in andere Prozesse erspart, spricht der Autor ausdrücklich eine Empfehlung für diese Option aus.

[11] `https://docs.oracle.com/javase/7/docs/api/java/lang/Float.html`
`https://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html`

```
1 public class NoExcEdgesTest {
2
3   public static Integer entry_method(String value) throws Exception {
4     Integer result = important(10);
5   }
6   public static void main(String[] argsv) throws Exception{
7     Integer result = entry_method(value);
8   }
9 }
```

Listing 6.9: TF04: Exception-Kontrollfluss - Slice FULL NO_EXC_E

6.2.5. TF05: Objektverfolgung

Jar:	`SimpleTest.jar`
Caller:	`entry_method`
Callee:	`important_method`
Datenabhängigkeit:	`FULL`
Kontrollabhängigkeit:	`NO_EXCEPTIONAL_EDGES`
Weitere Optionen:	`(mainclass = LSimpleTest), advanced_mode = true`

Der Testfall *TF05* behandelt die Problematik, dass WALA Datenflüsse teilweise nur auf Ebene von Variablen verfolgt, jedoch auf Ebene von Objekten ignoriert (siehe Abschnitt 5.3.3.1). Aus dieser Erkenntnis entstanden die Anforderungen aus der Systemfunktion SF13 (siehe Abschnitt 5.1.1.1), welche u.a. mit diesem Testfall geprüft werden.

Das konstruierte Beispielprogramm besteht zum einen aus der Klasse `SimpleTestObject`, welche die drei Methoden `do_stuff(), calc_param()` und `important_method()` zur Verfügung stellt. Die Methode `do_stuff()` modifiziert eine Klassenvariable `word` (`String`), während `calc_param()` lediglich eine Zahl zurückgibt und `important_method()` den zu untersuchenden Callee darstellen soll (vgl. vollständiger Quellcode B.8). Die zweite Klasse `SimpleTest` instanziiert ein neues `SimpleTestObject` und führt darauf zunächst `do_stuff()` und darauffolgend `important_method()` aus. Zusätzlich wird ein einfacher statischer Methodenaufruf `SimpleTestObject.calc_param()` ausgeführt und in einer Variable gespeichert, die später der `important_method()` als Parameter übergeben wird.

Den Erwartungen entsprechend, wird durch die Erweiterungen der „Objektverfolgung“ (aktiviert durch den Schalter `advance_mode`, vgl. Abschnitt 5.3.3.1) die Er-

zeugung des `SimpleTestObject`-Objektes `obj` und die Anweisung `obj.do_stuff()` dem Slice hinzugefügt (vgl. Slice 6.10 Zeile 9 - 10).

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

Listing 6.10: TF05: Objektverfolgung - Slice FULL NO_EXC_E (mit Angabe advanced_mode)

Es fällt weiterhin beim Testen auf, dass der statische Aufruf `SimpleTestObject.calc_param()` (Zeile 11) ebenfalls im Slice enthalten ist, auch wenn die Einstellung `advance_mode` deaktiviert ist, während `obj.do_stuff()` (Zeile 10) entfallen würde.

Ursache hierfür ist, dass WALA beim Slicen vom Callee `important_methode()` die beiden Parameter mit einbezieht, die der Methode übergeben werden. Die als Parameter übergebene Variable `addvalue` wird in Zeile 11 anhand `SimpleTestObject.calc_param()` erzeugt und verfügt somit eine Datenabhängigkeit. Dies erläutert ebenfalls den Umstand, dass die Definition der Funktion `calc_param()` komplett im Slice existiert (Zeile 1-5).

Bezüglich der Verfolgung der Parameter-Datenflüsse war in diesem Zusammenhang nicht erkenntlich, wieso der Datenfluss vom zweiten Parameter `anothervalue` nicht vollständig im Slice abgedeckt ist. Die Variable wird der `entry_method()` als Parameter von der main-Methode übergeben (vgl. vollständiger Quellcode B.9 Zeile 14-17), in der sie ebenfalls mit arithmetischen Operationen berechnet wird. Es wäre vorstellbar, dass die Entstehung und Werte der Parameter für Analysen von Interesse sein können.

6.2.6. TF06: Clinic (Verschlüsselung)

Jar:	`isCallerInRole.jar`
Caller:	`encryptData`
Callee:	`doFinal`
Datenabhängigkeit:	`FULL`
Kontrollabhängigkeit:	`FULL` (∼59 s), `NO_EXCEPTIONAL_EDGES` (∼10 s)
Weitere Optionen:	`(mainclass = Lclinic/ClinicExampleBean), (object_inst_methods = getInstance, advanced_mode = true)`

Der Testfälle *TF06* und *TF07* basieren auf der bereits eingeführten EJB Klinik-Anwendung (ca. 200 lines of code (LoC)) (siehe Abschnitt 4.2.1), die vom TZI zur Verfügung gestellt und von Detmers und Gerken zum Testen verwendet wurde. Für detailliertere Erläuterungen zum Inhalt dieser Anwendung sei auf die vorhergehenden Arbeiten zu verweisen, da im Rahmen dieser Arbeit nur grob auf die genutzten kryptografischen Funktionen eingegangen wird.

Der vorliegende Testfall soll unter anderem die Problematik der „Objektverfolgung“, im Vergleich zum funktionalen *TF05*, zusätzlich noch in den Kontext echter Applikationen mit kryptophischen Funktionen bewegen. In Abschnitt 5.3.3.1 wurde detailliert, anhand der Methode `doFinal()` der Bibliothek `javax/crypto/Cipher`, die Motivation und Vorteile dieser Erweiterung dargelegt. Dieser Testfall beschreibt spezifisch diesen Anwendungsfall anhand der exemplarischen Klinik-Anwendung.

Die komplette Implementierung der wesentlichsten Klasse `ClinicExampleBean` sei aus dem Anhang B.10 zu entnehmen.

```
1  @Stateful
2    private Clinician getClinician(String userID) {
3      return new Clinician(userID);
4    }
5    @Override
6      Clinician clinician = getClinician(userID);
7      byte[] signedData = signData(ehr.getDiagnosis().getBytes(), clinician.getKeyName());
8      byte[] encryptedSignedDiagnosis = encryptData(signedData, key);
9    }
10   @Override
11     return mSignature.sign();
12   }
13   @Override
14     return mCipher.doFinal(data);
15   }
16 }
```

Listing 6.11: TF6: Clinic (Verschlüsselung) - Slice FULL NO_EXC_E (keine Optimierungen)

© Philip Phu Dang Hoan Nguyen

Zunächst soll das Slicing mit der ursprünglichen Slicing-Implementierung und den Optionen `FULL NO_EXC_E` (Caller ist `encryptData` und Callee `doFinal()`) durchgeführt werden (vgl. Slice 6.11).

Es fällt die fehlerhafte Syntax in Zeile 1, 5, 10 und 13 auf, deren Ursache in der ursprünglichen Implementierung des Parsers liegt (siehe Abschnitt 5.3.3.2), welche die Java-Annotationen nicht korrekt parst. Die Ergebnisse der Erweiterung des Parsers zeigen sich im Slice 6.12, welcher die fehlenden Klassen- und Methodenköpfe inklusive blockeinleitende Klammer „{“ beinhaltet.

```
1  @Stateful
2  public class ClinicExampleBean implements ClinicExample {
3
4    private Clinician getClinician(String userID) {
5      return new Clinician(userID);
6    }
7    @Override
8    public void sendDiagnosis(String userID, String ehrID, Key key) throws InvalidKeyException,
         ... {
9      Clinician clinician = getClinician(userID);
10     byte[] signedData = signData(ehr.getDiagnosis().getBytes(), clinician.getKeyName());
11     byte[] encryptedSignedDiagnosis = encryptData(signedData, key);
12   }
13   @Override
14   public byte[] signData(byte[] data, String keyAlias) throws NoSuchAlgorithmException, ... {
15     return mSignature.sign();
16   }
17   @Override
18   public byte[] encryptData(byte[] data, Key key) throws NoSuchAlgorithmException, ... {
19     return mCipher.doFinal(data);
20   }
21 }
```

Listing 6.12: TF06: Clinic (Verschlüsselung) - Slice FULL NO_EXC_E (mit Parsererweiterungen)

Anhand dieses Slices lassen sich Datenflüsse bezüglich des Slicing-Kriteriums `do_final()` erkennen (die Ziffern in den eckigen Klammern beschreiben die jeweiligen Zeilen aus dem Slice 6.12):

`doFinal()` [19] ⇒ `encryptData()` [18] ⇒ `encryptData(signedData, key)` [11] ⇒ `signedData = signData(ehr..., clinician...)` [10] ⇒ `signData()` [14] ⇒ `...`

Der Slice ist somit zwar nachvollziehbar und korrekt, jedoch fehlen die notwendigen Informationen zur Entstehung des `Cipher`-Objekts `mCipher` (Zeile 19) und die

darauf angewendete Methoden, die Informationen zum Verschlüsselungsalgorithmus, Modus und der Herkunft des Schlüssel liefern können.

Das Fehlen dieser Anweisungen beschreibt ein *false-negative*, welches bei der Evaluierung der Exception-Kontrollflüsse (siehe Abschnitt 5.2.4) ermittelt wurde. Wird beim Slicing die Option `FULL FULL` genutzt, gelangen diese interessanten Anweisungen zufällig durch die Exception-Kontrollflüsse in den Slice. Diese Exception-Kontrollflüsse erweisen sich im Verlauf der Arbeit als zu undurchsichtig bzw. nicht transparent.

Folgender Auszug vom Slice 6.13 mit `FULL FULL` zeigt die `encryptData()` Methode mit den benötigten Anweisungen.

```
1  ...
2  @Override
3  public byte[] encryptData(byte[] data, Key key) throws NoSuchAlgorithmException, ... {
4    mCipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
5    a=2;
6    mCipher.init(Cipher.ENCRYPT_MODE, key);
7    a--;
8    return mCipher.doFinal(data);
9  }
10 ...
```

Listing 6.13: TF06: Clinic (Verschlüsselung) - Auszug Slice FULL FULL (mit Parsererweiterungen)

In diesem Fall kann `Cipher.getInstance()` eine `NoSuchAlgorithException` oder `NoSuchPaddingException` verursachen, während `Cipher.init()` bei einem nicht-zulässigen Schlüssel eine `InvalidKeyException` wirft. Weitere Exception-Kontrollflüsse, die durch die Wrapper-Klasse `Integer` erzeugt wurden (Zeile 5 und 7), stammen aus dem Testfall von Gerken (vgl. [Gerken 2015], Kapitel 5.4).

Neben der mangelnden Transparenz fällt bei Nutzung der Kontrollabhängigkeitsoption `FULL` zusätzlich auf, dass die Ausführungszeit von ca. 10 auf ca. 60 Sekunden ansteigt, welches mit 6-facher Laufzeit bei größeren bzw. komplexeren Applikationen unzumutbare Ausführungszeiten als Folge haben würde. Bei der Analyse der Slicing-Ausgaben (vgl. B.11 im Anhang) zeigt sich, dass zu den 42.981 Knoten des SDGs durch die Exception-Kontrollflüsse noch über 2.860 weitere Knoten und zu den 62.844 Kanten noch 500 Kanten hinzugefügt wurden. Ferner ist auffällig, dass der Slice von 81 auf 34.440 Statements anwächst. Ursache hierfür ist, dass die genutzten Bibliotheken viele Exception-Kontrollflüsse aufweisen und WALA die Datenflüsse bis in die jeweiligen Exception-Klassen analysiert und traversiert. Die Auflistung 6.14

zeigt gekürzt Ausgaben des Auditors bezüglich dieser Exception-Klassen. Die Dateien werden in diesem Fall gesucht, da zu jeder dieser von WALA ein Slice berechnet, doch die Originaldateien zur Rekonstruktion nicht bereitgestellt wurden.

```
...
KeyStoreException.java file not found! Skipping!
IOException.java file not found! Skipping!
NoSuchProviderException.java file not found! Skipping!
NoSuchAlgorithmException.java file not found! Skipping!
SecurityException.java file not found! Skipping!
NoSuchFieldException.java file not found! Skipping!
GeneralSecurityException.java file not found! Skipping!
MalformedURLException.java file not found! Skipping!
```

Listing 6.14: TF06: Clinic (Verschlüsselung) - Auszug Slicing-Ausgaben FULL FULL

Die Tatsache, dass die Datenflussanalyse erkennbar viele Exception-Klassen erreicht, zeigt, wie schnell die Komplexität des Slicings bei kleinen Anwendungen steigen kann, durch unterschiedliche Parametrisierung. Das Einbeziehen von Exception-Kontrollflüssen ergibt in diesem Kontext dementsprechend ausschließlich Nachteile, davon abgesehen, dass das `Cipher`-Objekt und die darauf bezogenen Methodenaufrufe zufällig durch die Kontrollabhängigkeitsoption in den Slice gekommen sind.

Im Unterschied zum Testfall *TF05* geht die implementierte Pointeranalyse bzw. Objektverfolgung in diesem Testfall nicht von einem Objekt aus, welches über eine New-Operation erzeugt wurde, sondern von dem Klassenaufruf `Cipher.getInstance()`, wie bereits in Abschnitt 5.3.3.1 erläutert wurde. Daher muss neben der Option `advanced_mode` ebenfalls der Wert `getInstance` für die Option `object_inst_method` in der Konfigurationsdatei angegeben werden, damit die Instanziierung korrekt erkannt wird.

Der Auszug vom Slice 6.15 zeigt letztendlich ein korrekte Ergebnis, welches mit `FULL NO_EXC_E` und den genannten Optionen erzeugt wurde.

```
...
@Override
  public byte[] encryptData(byte[] data, Key key) throws NoSuchAlgorithmException, ... {
    mCipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
    mCipher.init(Cipher.ENCRYPT_MODE, key);
    return mCipher.doFinal(data);
}
...
```

Listing 6.15: TF06: Clinic (Verschlüsselung) - Auszug Slice FULL NO_EXC_E (mit allen Erweiterungen)

6.2.7. TF07: Clinic (Signieren)

Jar:	`isCallerInRole.jar`
Caller:	`signData`
Callee:	`sign`
Datenabhängigkeit:	`FULL`
Kontrollabhängigkeit:	`NO_EXCEPTIONAL_EDGES`
Weitere Optionen:	`(mainclass = Lclinic/ClinicExampleBean), (object_inst_methods = getInstance, advanced_mode = true)`

Der Testfall *TF07* basiert, wie der vorherige Testfall, ebenfalls auf der fiktiven Klinik-Anwendung, setzt den Fokus jedoch auf die digitale Signatur der Daten in der Anwendung mit Hilfe der Bibliothek `java.security.Signature`. Dabei wird als Slicing-Kriterium die Methode `sign()` als Callee gewählt, welche eine Signatur für einen Datenblock berechnet und ausgibt.

Zunächst zeigt der erste Slice 6.16 mit `FULL NO_EXC_E` und den Parsererweiterungen ähnliche Ergebnisse wie *TF06*, nur dass in diesem Testfall die Methode `signData()` der Caller ist.

```
1  public class ClinicExampleBean implements ClinicExample {
2    private Clinician getClinician(String userID) {
3      return new Clinician(userID);
4    }
5    @Override
6    public void sendDiagnosis(String userID, String ehrID, Key key) throws InvalidKeyException,
           ... {
7      Clinician clinician = getClinician(userID);
8      byte[] signedData = signData(ehr.getDiagnosis().getBytes(), clinician.getKeyName());
9    }
10   @Override
11   public byte[] signData(byte[] data, String keyAlias) throws NoSuchAlgorithmException, ... {
12     mSignature = Signature.getInstance("SHA256withDSA", "SUN") ;
13     return mSignature.sign();
14   }
15 }
```

Listing 6.16: TF07: Clinic (Signieren) - Slice FULL NO_EXC_E (mit Parser-Erweiterungen)

Bei der Betrachtung der Slicing-Ergebnisse fällt ins Auge, dass die Instanziierung des `Signature`-Objekts im Slice enthalten ist, obwohl weder `advanced_mode`, noch `object_inst_methods` angegeben wurde. Dieses Verhalten konnte bis zur Fertigstellung der Arbeit nicht geklärt werden.

Ferner ist die Verwendung der Bibliothek für digitalen Signatur `Signature` ähnlich der symmetrischen Verschlüsselung `Cipher` und wird über die statische Methode `getInstance()` mit Angabe des Signatur-Algorithmus' und dem Anbieter (engl. provider) instanziiert. Des Weiteren wird auf das erstellte `Signature`-Objekt mit dem Aufruf `initSign()` das Signatur-Objekt mit einem privaten Schlüssel initialisiert und daraufhin für jeden Block ein `update()`-Aufruf gemacht mit dem jeweiligen Byte-Block der definierten Blocklänge[12] (vgl. vollständiger Quellcode B.10).

Analog zu Testfall *TF06* wird der korrekte Slice mit den Optionen `FULL NO_EXC_E`, `object_inst_methods (getInstance)` und `advanced_mode (true)` erstellt und somit um die fehlenden Anweisungen ergänzt (vgl. Slice 6.17).

```
1  public class ClinicExampleBean implements ClinicExample {
2    private Clinician getClinician(String userID) {
3      return new Clinician(userID);
4    }
5    @Override
6    public void sendDiagnosis(String userID, String ehrID, Key key) throws InvalidKeyException,
           ... {
7      Clinician clinician = getClinician(userID);
8      byte[] signedData = signData(ehr.getDiagnosis().getBytes(), clinician.getKeyName());
9    }
10   @Override
11   public byte[] signData(byte[] data, String keyAlias) throws NoSuchAlgorithmException, ... {
12     char[] kpass = getKeyPass();
13     mSignature = Signature.getInstance("SHA256withDSA", "SUN") ;
14     KeyStore ks = KeyStore.getInstance("Hospital");
15     PrivateKey priv = (PrivateKey) ks.getKey(keyAlias,kpass) ;
16     mSignature.initSign(priv);
17     mSignature.update(data);
18     return mSignature.sign();
19   }
20   private char[] getKeyPass() {
21     return null;
22   }
23 }
```

Listing 6.17: TF07: Clinic (Signieren) - Slice FULL NO_EXC_E (mit allen Erweiterungen)

Dadurch sind in Zeile 16 und 17 die beiden Anweisungen `initSign()` und `update()` nachvollziehbar und mit Einbeziehung deren Parameter bspw. ebenfalls die Schlüsselbehandlung im Slice verfügbar (Zeilen 14,15 und 20-22).

[12] `https://docs.oracle.com/javase/7/docs/api/java/security/Signature.html`, Aufruf: 18.03.2018

6.2.8. TF08: Coding-Konventionen

Jar:	`codingconvtest.jar`
Caller:	`entry_method`
Callee:	`println`
Datenabhängigkeit:	`FULL`
Kontrollabhängigkeit:	`NO_EXCEPTIONAL_EDGES`
Weitere Optionen:	`(mainclass = LCodingConventionTest)`

Die Testfälle *TF08* und *TF09* entstanden, wie bereits in Abschnitt 5.1.1.1 beschrieben, aus der Anforderung die resultierenden Slices vollständig als syntaktisch-korrekte Java-Dateien auszugeben. Der Abschnitt 5.3.3.2 erläuterte bereits die Probleme des Parsers, der in seiner ursprünglichen Form unvollständige Ergebnisse lieferte.

Die Funktionsweise des neu entwickelten Parsers wurde in Abschnitt 5.3.3.2 beschrieben und soll u.a. mit diesem Testfall getestet werden. Hierbei soll das Beispielprogramm von *TF08* ungewöhnliche Coding-Konventionen nachstellen, wie bspw. mehrzeilige Methodenköpfe (resultiert durch Umbrüche) oder blockeinleitende Klammerungen, die in einer neuen Zeile gesetzt werden.

Die Klasse `CodingConventionTest` besteht neben einer `main`-Methode ausschließlich aus der Methode `entry_method()`, welche drei Parameter übergibt, die jeweils in eine neue Zeile umgebrochen werden (bspw. zur Übersichtlichkeit, vgl. vollständiger Quellcode B.12). Sie fungiert als Caller und besitzt neben einer `for`-Schleife und zwei `if-else`-Anweisungen keine nennenswerte Funktionalität. Die zweite `if-else`-Anweisung bildet ihre Besonderheit darin, dass sie unkonventionell ohne geschweifte Klammerung gebildet wird und zudem im `else`-Zweig den für diesen Test verwendeten Callee `println()` enthält.

Der Slice 6.18 soll zunächst demonstrieren, dass die Ergebnisse des alten Parsers fehlerhaft und für eine weitere Analyse nicht nutzbar sind.

```
1  public class CodingConventionTest
2    public static Integer entry_method( int value,
3      for (int i = 0; i<=duration; duration++)
4        if (i==0)
5          if (i < 5)
6          System.out.println("important");
7      }
8    }
9    public static void main(String[] argsv) throws Exception
10     Integer result = entry_method(1,2,3);
11   }
12 }
```


© Philip Phu Dang Hoan Nguyen

Listing 6.18: TF08: Coding-Konventionen - Slice FULL_NO_EXC_E (mit altem Parser)

Aus dem resultierenden Slice ist zu erkennen, dass die Syntax inkorrekt ist und alle Anweisungen unvollständig sind. Unter anderem fehlt die Klammer „{“ für die Klasse `CodingConventionTest` und die beiden Klassen `entry_method()` und `main()` in den Zeilen 1,2 und 9. Ebenfalls fehlen die Klammern für die `for`-Schleife und die `if`-Anweisung in Zeile 3 und 4, zumal die beiden `else`-Anweisungen komplett fehlen, obwohl diese zum einen den zweiten `if-else` und zum anderen das Slice Kriterium `println()` beinhalten.

Mit Benutzung des neuen Parsers entsteht ein syntaktisch korrekter Slice (vgl. Quellcode 6.19). Die in Abschnitt 5.3.3.2 eingeführten Optimierungen zur korrekten Darstellung der mehrzeiligen Klassen- und Methodenköpfe erfüllen zwar ihren Zweck, können jedoch ebenfalls unerwünschte Nebeneffekte wie mitgeparste Kommentare (siehe Zeile 3) verursachen.

```
1  public class CodingConventionTest
2  {
3    // entry_method does something
4    public static Integer entry_method( int value,
5                    int value2,
6                    int value3)
7                    throws Exception
8    {
9      for (int i = 0; i<=duration; duration++)
10     {
11       if (i==0)
12       {
13       }
14       else
15       {
16         if (i < 5)
17         else
18         System.out.println("important");
19       }
20     }
21   }
22   public static void main(String[] argsv) throws Exception
23   {
24     Integer result = entry_method(1,2,3);
25   }
26 }
```

Listing 6.19: TF08: Coding-Konventionen - Slice FULL_NO_EXC_E (mit neuem Parser)

6.2.9. TF09: Tief-verschachtelter Callee

Jar:	`deepcaleetest.jar`
Caller:	`entry_method`
Callee:	`important`
Datenabhängigkeit:	`FULL`
Kontrollabhängigkeit:	`NO_EXCEPTIONAL_EDGES`
Weitere Optionen:	`(mainclass = LDeepCallee)`

Der Testfall *TF09* bildet, wie im vorherigen Testfall erläutert, einen weiteren Test für den neu-implementierten Parser. Der Fokus wird in diesem Testfall auf tief-verschachtelte Anweisungsblöcke gesetzt, in welchen sich ein Callee befinden kann. Da mit Hilfe der Bibliothek JavaParser jede Anwendung in einen AST transformiert werden kann, ist es möglich diese verschachtelten Anweisungen gezielt zu traversieren und Callee-relevante Teile inkl. Klammerung zu parsen (siehe Abschnitt 5.3.3.2).

Die Beispielanwendung besteht aus der Klasse `DeepCallee` mit der Methode `entry_method()` (in diesem Fall der Caller) und einer Methode `important()`, welche die Callee-Anweisung für diesen Testfall darstellen soll. Die Methode `entry_method()` verschachtelt mehrfache `if-else`-Anweisungen, die in einer `for`-Schleife aufgerufen werden. Eine weitere Ebene fügt eine `while`-Schleife hinzu, in welcher die genannte `for`-Schleife definiert ist (vgl. vollständiger Quellcode B.13).

Der Slice 6.20 zeigt zunächst die Ergebnisse des alten Parsers.

```
1  public class DeepCallee {
2    public static Integer entry_method(String value) {
3      while (stay){
4        for (int i = 0; i<=duration; duration++){
5          if (i==0){
6          else if (i==5){
7            if (i % 2 == 1){
8              result = important(i);
9      }
10   }
11   public static void main(String[] argsv){
12     Integer result = entry_method(value);
13   }
14 }
```

Listing 6.20: TF09: Tief-verschachtelter Callee - Slice FULL_NO_EXC_E (mit altem Parser)

Zunächst ist zu erkennen, dass die Syntax der `entry_method()` unvollständig ist, da mehrere schließende geschweifte Klammern fehlen. Bei genauerem Vergleich mit dem

 © Philip Phu Dang Hoan Nguyen

vollständigen Quellcode fällt zusätzlich auf, dass die Anweisung `if (i % 2 == 1){` (Zeile 7) nicht zu der `else-if`-Anweisung der vorhergehenden Zeile gehört, sondern ursprünglich Teil des `else`-Zweigs ist, welcher nicht im Slice enthalten ist.

Die jeweiligen Anweisungen aus den Zeilen von 3-7 sind in diesem Fall nur im Slice enthalten, da sie Kontrollabhängigkeiten bilden.

Mit Benutzung des neuen Parsers, werden über einen rekursiven Aufruf einer Methode, die relevanten Anweisungen zu einer Zeile der Sliceliste berechnet (siehe Abschnitt 5.3.3.2). Der resultierende Slice wird durch die implementierten Erweiterungen syntaktisch korrekt (vgl. Slice 6.19).

```
1  public class DeepCallee {
2
3    public static Integer entry_method(String value) {
4      while (stay){
5        for (int i = 0; i<=duration; duration++){
6          if (i==0){
7          }
8          else if (i==5){
9          }
10         else {
11           if (i % 2 == 1){
12             result = important(i);
13           } else {
14           }
15         }
16       }
17     }
18   }
19   public static void main(String[] argsv){
20     Integer result = entry_method(value);
21   }
22 }
```

Listing 6.21: TF09: Tief-verschachtelter Callee - Slice FULL_NO_EXC_E (mit neuem Parser)

6.2.10. Zusammenfassung

Mit Hilfe einer Reihe von Testfällen konnten grundsätzlich alle Systemfunktionen des Anforderungskatalogs (siehe Abschnitt 5.1.1.1) getestet und im Ergebnis erfolgreich verifiziert werden. Es ist somit davon auszugehen, dass der Auditor allen funktionalen Anforderungen gerecht wird und für Sicherheitsanalysen angewendet werden kann.

Der erste Teil der Evaluierung ergab ebenfalls, dass beim Slicing die Kontrollabhängigkeitsoption `NO_EXCEPTIONAL_EDGES` mit den Erweiterungen der „Objektverfolgung“ zufriedenstellende Ergebnisse erzielt. Die Option `FULL` ist dementsprechend nicht empfehlenswert, da die Behandlung von Exception-Kontrollflüssen viele nicht benötigte Abhängigkeiten dem SDG ergänzt und die Slicing-Resultate undurchsichtig beeinflusst.

Einzige Unklarheiten verbleiben ausschließlich beim Slicing im Testfall *TF06* und *TF07*, bei welchen nicht deutlich wurde, wieso die Instanziierung eines Objektes mit der Methode `getInstance()` in einem Fall erkannt und im anderen Fall nicht erkannt wurde.

Ebenfalls war in *TF05* nicht eindeutig zu erkennen, in welchem Maße die Parameter eines Callees in die Analyse beeinflussen. Der Testfall zeigte in diesem Fall für zwei Parameter unterschiedlich-detaillierte Analysen der Datenflüsse. Dadurch war bei einem Parameter zwar zu erkennen, dass die Variable über einen übergeordneten Methodenaufruf direkt an die Callee-Methode übergeben wurde, hingegen nicht, wie diese Variable entstanden ist und verarbeitet wurde (siehe Abschnitt 6.2.5).

 © Philip Phu Dang Hoan Nguyen

6.3. Evaluierung Teil II

6.3.1. TF10: Openkeepass

Jar:	`openkeepass-0.6.1.jar`
Caller:	`transformData`
Callee:	`doFinal`
Datenabhängigkeit:	`FULL`
Kontrollabhängigkeit:	`NO_EXCEPTIONAL_EDGES`
Weitere Optionen:	`(mainclass = Lde/slackspace/openkeepass/crypto/Aes), (mainclass = Lde/slackspace/openkeepass/main/Main), advanced_mode = true, object_inst_methods = getInstance`

Openkeepass ist eine Java-Bibliothek mit der *KeePass*-Datenbanken gelesen und geschrieben werden können [openkeepass 2018]. *KeePass* ist eine leichtgewichtige Open-Source-Anwendung zur Verwaltung von Passwörtern. Die Grundfunktionalität besteht darin, dass beim Öffnen einer KeePass-Datenbank entweder ein Generalschlüssel (engl. master key) oder eine Schlüsseldatei (engl. key file) angegeben werden muss, bevor der Zugriff auf die Passwort-Datenbank gewährt wird. Mit Hilfe dieser Anwendung können zum einen komplexere Passwörter generiert und verwendet und zum anderen für andere Konten unterschiedliche Passwörter gewählt werden. Diese Vorgehensweise verhindert, dass bei einem Datenleck eines Anbieters nicht alle Passwörter anderer Konten betroffen sind.

Da die KeePass-Datenbanken in Form einer Datei (.kdbx) bereitgestellt werden, müssen die Inhalte dieser Datei zum Schutz verschlüsselt werden mit Hilfe eines Generalschlüssels oder einer Schlüsseldatei. Die Art der Verschlüsselung und Entsperrung der Datenbank wird mit folgendem Testfall untersucht.

Bei der `openkeepass-0.6.1.jar` (ca. 6.000 LoC) handelt es sich nicht um eine fertige Anwendung, sondern um eine Bibliothek, die in jeder Java- oder Android-Anwendung eingebunden werden kann. Dennoch bietet *openkeepass* Testfälle, welche die grundlegenden Funktionen der Bibliothek implementieren und somit der realen Analyse einer Applikation mit KeePass-Funktionalität nahe kommen.

Für den Testfall *TF10* wurde aus *openkeepass* (Version 0.6.1) der Modultest (Unit Test) `whenWritingDatabaseFileShouldBeAbleToReadItAlso` von `src/test/java/de/slackspace/openkeepass/api/KeepassDatabaseWriterTest.java` in eine Main-Klasse umgeschrieben, gebaut und in ein Jar-Archive gebündelt (vgl. Quellcode 6.22).

```
1  public class Main {
2      public static String getResource(String path) {
3          return Main.class.getClassLoader().getResource(path).getPath();
4      }
5      public static void whenWritingDatabaseFileShouldBeAbleToReadItAlso() throws IOException {
6          byte[] protectedStreamKey = ByteUtils.hexStringToByteArray("
               ec77a2169769734c5d26e5341401f8d7b11052058f8455d314879075d0b7e257");
7          FileInputStream fileInputStream = new FileInputStream(getResource("
               testDatabase_decrypted.xml"));
8          KeePassFile keePassFile = new KeePassDatabaseXmlParser(new SimpleXmlParser()).fromXml(
               fileInputStream);
9          new ProtectedValueProcessor().processProtectedValues(new DecryptionStrategy(Salsa20.
               createInstance(protectedStreamKey)), keePassFile);
10         FileOutputStream file = new FileOutputStream("writeDatabase.kdbx");
11         KeePassDatabase.write(keePassFile, "abcdefg", file);
12         KeePassDatabase database = KeePassDatabase.getInstance(writeDatabase);
13         KeePassFile openDatabase = database.openDatabase("abcdefg");
14     }
15   public static void main(String[] argsv) throws IOException{
16     whenWritingDatabaseFileShouldBeAbleToReadItAlso();
17   }
18 }
```

Listing 6.22: TF10: Openkeepass - Quellcode

Grundsätzlich erstellt die Beispielanwendung eine neue verschlüsselte KeePass-Datei mit Hilfe einer XML-Datei, die mit Beispieldaten gefüllt ist. Diese wird im Folgeschritt in ein `KeePassDatabase`-Objekt initialisiert, verschlüsselt und mit Hilfe des Passworts erneut geöffnet.

Von dieser Anwendung werden zwar weder die Methoden, noch Aufrufe innerhalb der Methoden als Slice-Kriterium verwendet, doch soll später der Datenfluss von diesem Programmaufruf bis zu den Verschlüsselungsfunktionen untersucht werden. Als Callee wird der Aufruf `doFinal()` untersucht, um zu analysieren, welche Programmteile der Bibliothek in Verbindung zu symmetrischer Verschlüsselung stehen. Der Aufruf der Verschlüsselung bzw. Entschlüsselung befindet sich in der Methode `transformData()` in der Klasse `Aes` (`de/slackspace/openkeepass/crypto/Aes.java`), weswegen `transformData` als Caller gesetzt wird.

Die erste Analyse mit Hilfe des Auditors soll zunächst mit den Optionen `FULL NO_EXCEPTIONAL_EDGES`, `object_inst_methods (getInstance)` und Angabe der `mainclass` (`Lde/slackspace/openkeepass/crypto/Aes`) durchgeführt werden.

Der resultierende Slice besteht, wie zu erwarten, einzig aus der `Aes.java` (vgl. Slice 6.23).

© Philip Phu Dang Hoan Nguyen

```
1  public class Aes {
2    public static byte[] decrypt(byte[] key, byte[] ivRaw, byte[] data) {
3      return transformData(key, ivRaw, data, Cipher.DECRYPT_MODE);
4    }
5    public static byte[] encrypt(byte[] key, byte[] ivRaw, byte[] data) {
6      return transformData(key, ivRaw, data, Cipher.ENCRYPT_MODE);
7    }
8    private static byte[] transformData(byte[] key, byte[] ivRaw, byte[] encryptedData, int
         operationMode) {
9      try {
10       Cipher cipher = Cipher.getInstance(DATA_TRANSFORMATION);
11       Key aesKey = new SecretKeySpec(key, KEY_ALGORITHM);
12       IvParameterSpec iv = new IvParameterSpec(ivRaw);
13       cipher.init(operationMode, aesKey, iv);
14       return cipher.doFinal(encryptedData);
15     } catch (NoSuchAlgorithmException e) {
16     }
17   }
18 }
```

Listing 6.23: TF10: Openkeepass - Slice FULL NO_EXC_E (mit Angabe mainclass)

Der Slice und die Rekonstruktion des Original-Quellcodes sind zunächst syntaktisch und nachvollziehbar korrekt. Zwar sind alle Anweisungen von der Instanzierung und Initialisierung des `Cipher`-Objekts bis zum Aufruf von `doFinal()` im Slice enthalten, doch werden die meisten Parameter über Konstanten gesetzt, deren Initialisierung nicht im Slice sind. Die drei Methoden `decrypt()` `encrypt()` und `transformData()` unterscheiden sich im Slice vom Original nicht bedeutend bis auf die entfernte Exceptionbehandlung (siehe Quellcodeauszug B.14 im Anhang).

Dass die Datenflüsse ausschließlich isoliert innerhalb der Klasse `Aes` verfolgt werden, liegt an der fest definierten Entrypoint-Klasse über die Option `mainclass`. Damit die Datenflussanalyse interprozedural über andere Klassen durchgeführt wird, muss für die Angabe der Option `mainclass` zur Ermittlung der Entrypoints eine andere Klasse definiert oder komplett darauf verzichtet werden, um alternativ im kompletten Programm nach Entrypoints zu suchen (siehe Abschnitt 5.2.2). Im Zuge dessen wird erzielt, dass über verschiedene Algorithmen andere Programm-Einstiegspunkte ermittelt und letztendlich dem Callgraph ergänzt werden. Dies hat wesentliche Auswirkungen auf das Backward-Slicing, bei dem vom Slice-Kriterium (Callee-Knoten) über alle Kanten des SDGs bis zu den Entrypoints Pfade gesucht und gesliced werden.

Zunächst wird das Slicing ohne Angabe einer `mainclass` getestet, welches intern in der Implementierung des Auditors in der Voreinstellung der Verwendung der WALA-Funktion `makeMainEntrypoints()` entspricht. Im Rahmen der Evaluierung

sollte zum Vergleich für eine gezieltere und effizientere Analyse direkt die erzeugte Beispielanwendung `Main` als `mainclass` getestet werden. Entgegen den Erwartungen erzielten diese Tests, wie in der Tabelle 6.1 festgehalten, schlechtere Ergebnisse bezüglich der Verarbeitungszeit. Ursache für die viermal längere Slicingdauer ist die Anzahl der ermittelten Entrypoints. Die Funktion `makeMainEntrypoints()` ermittelte exakt einen Entrypoint, während die beiden Algorithmen von Gulmann und Detmers jeweils vier Entrypoints berechneten. Bei der Betrachtung der beiden Implementierungen ist allerdings ersichtlich, dass zunächst die gesuchte Klasse aus der Klassenhierarchie ermittelt wird und anschließend die zugehörigen Methoden als Entrypoints ausgewählt werden. In diesem Fall gibt es keine Differenzierung der beiden Algorithmen, da alle Methoden der Beispielanwendung als `public` deklariert sind. Die Ursache, weswegen in diesem Fall vier Entrypoints bei drei Methoden ermittelt wurden, konnte nicht eindeutig geklärt werden. Im Folgenden wird allerdings mit dem Ergebnis weitergearbeitet, welches beim effizientesten Slicing mit einem Entrypoint entstanden ist.

Mainclass:	**AES**	**Main**	**Ohne**
Ausführungszeit:	ca. 10 s	ca. 20 min	ca. 5 min
Callgraph Knoten:	243	587	585
SDG Knoten:	45.650	127.776	124.095
SDG Kanten:	68.908	190.211	183.593
Statements im Slice:	52	103.828	83.213
Betroffene Klassen:	1	33	28

Tabelle 6.1.: Vergleich Slicing mit und ohne Angabe der mainclass

Wie erwähnt gibt die Tabelle 6.1 einen Überblick über die vom Slicer zu verarbeitenden Daten im Vergleich zum vorherigen Slicing über die Klasse `AES`, bei dem ausschließlich die Datenflüsse innerhalb der Klasse analysiert wurden.

Ungeachtet dessen, dass beim Slicing von `AES` mehrere Entrypoints ermittelt werden, befindet sich diese Klasse in einer tieferen Programmschicht. Die zugehörigen Methoden werden hauptsächlich aufgerufen, um Daten zu verarbeiten (bspw. mit der Bibliothek `Cipher`), verursachen im Gegensatz dazu vergleichsweise wenige neue Aufrufe von Funktionen oder Prozessen.

Die Testdaten zeigen diesbezüglich, dass das Slicing der Beispielanwendung einen mehr als doppelt-großen Callgraph erzeugt. Dies lässt sich damit erklären, dass

die Testanwendung auf höchster Programmschicht das Schreiben und Lesen der KeePass-Datenbank anstößt und somit alle Programmaufrufe nach sich zieht.

Der größere Callgraph erklärt ebenfalls das Anwachsen des SDGs um 78.445 Knoten und 114.685 Kanten. Ein weiterer Aspekt ist, dass der Programmeinstiegspunkt und das Slice-Kriterium für diesen Testfall bezüglich des Datenflusses weit auseinander liegen, weswegen erwartet werden kann, dass beim Traversieren des SDGs deutlich mehr Kanten durchlaufen werden müssen.

Im Gegensatz zur vorherigen Ausführung enthält der Slice dementsprechend 83.213 Statements (im Vergleich vorher 52) aus 28 Java-Dateien von *openkeepass* und 58 weiteren betroffenen Dateien/Klassen, die nicht rekonstruiert werden konnten, da diese aus Bibliotheken stammen (bspw. FileOutputStream, JceSecurity, FileDescriptor, SecurityException usw.). Abbildung 6.1 zeigt alle aus dem Slice rekonstruierten Java-Dateien. Durch das Slicing wurde der Quellcode von ca. 6.000 auf 943 LoC reduziert, was einer Reduktion von ca. 84% entspricht[13]. Die Ausführungszeit von ca. fünf Minuten ist in Anbetracht der steigenden Komplexität plausibel.

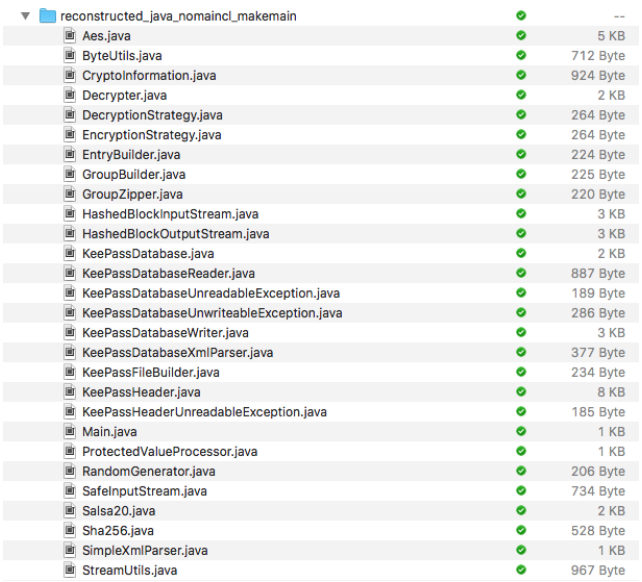

Abbildung 6.1.: TF10: Rekonstruierte Java-Dateien

[13]Berechnung ohne Berücksichtigung von Imports und Kommentaren

Im folgenden Abschnitt soll auf die Inhalte der Slicing-Ergebnisse eingegangen werden. Bezüglich dessen müssen die rekonstruierten Dateien zum einen von der Syntax korrekt sein und zum anderen alle relevanten Anweisungen vom Slice-Kriterium `doFinal()` zu einem Programm-Einstiegspunkt vorweisen können.

Die Slicing-Ergebnisse betrachtend, ist zunächst zu erkennen, dass die eigens erstellte Klasse `Main` vorhanden ist und somit gewisse Datenflüsse und Abhängigkeiten zwischen dem `doFinal()` und der Beispielanwendung bestehen müssen. Die Abbildung 6.2 zeigt diesen Datenfluss, der durch das Backward-Slicing des SDGs entstanden ist.

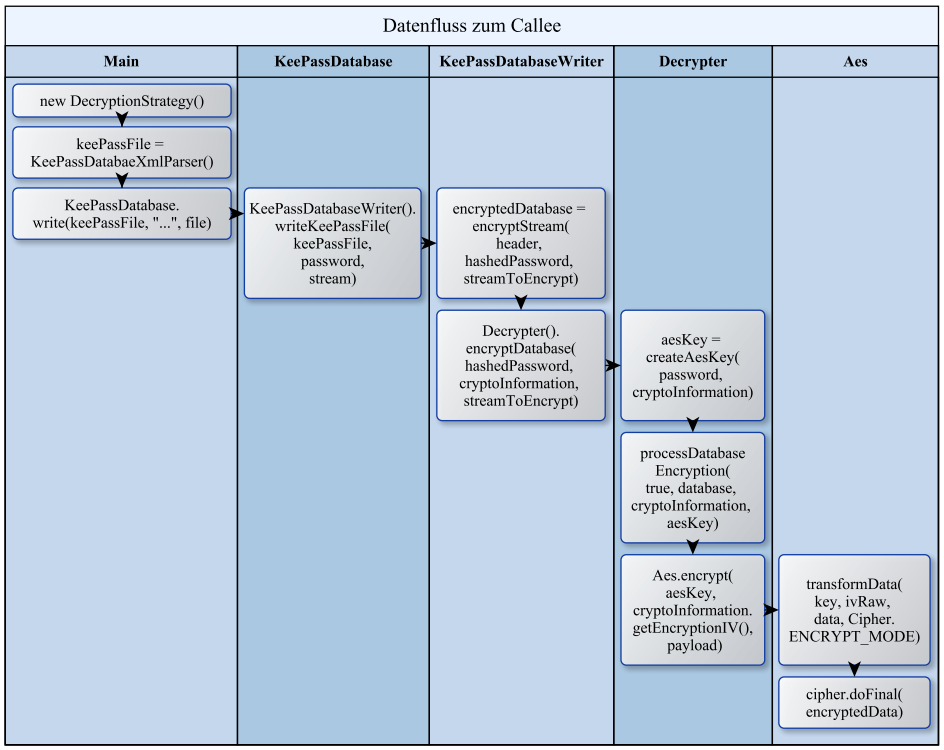

Abbildung 6.2.: TF10: Datenflüsse zwischen Main-Klasse und doFinal()

Mit Hilfe von Suchfunktionen einer beliebigen Integrated Development Environment (IDE) und dem reduzierten Quellcode, kann hierbei vereinfacht von der Klasse `Aes` nach dem Aufruf `Aes.encrypt()` gesucht und von diesem wiederholend die aufrufenden Methoden bzw. Klassen identifiziert werden. Folglich ist zu erkennen, dass der Entrypoint für diesen Testfall in der `Main` dem Aufruf `KeePassDatabase.write()` entspricht, welche das Schreiben der KeePass-Datei anstößt.

Folgende Auszüge der Slices der Klasse `KeePassDatabaseWriter` (vgl. Slice 6.24), `KeePassDatabaseWriter` (vgl. Slice 6.25) und `Decrypter` (vgl. Slice 6.26) sollen

exemplarisch zeigen, dass wichtige Informationen zum Datenfluss zur symmetrischen Verschlüsselung im Slice enthalten sind.

```
1 public class KeePassDatabase {
2   ...
3   public static void write(KeePassFile keePassFile, String password, OutputStream stream) {
4     if (stream == null) {
5       throw new IllegalArgumentException("You must provide a stream to write to.");
6     }
7     new KeePassDatabaseWriter().writeKeePassFile(keePassFile, password, stream);
8   }
9 }
```

Listing 6.24: TF10: Openkeepass - Slice FULL NO_EXC_E (ohne Angabe mainclass) KeePassDatabase.java

```
1  public class KeePassDatabaseWriter {
2
3    public void writeKeePassFile(KeePassFile keePassFile, String password, OutputStream stream)
         {
4      try {
5        if (!validateKeePassFile(keePassFile)) {
6          throw new KeePassDatabaseUnwriteableException(
7        }
8        KeePassHeader header = new KeePassHeader(new RandomGenerator());
9        byte[] hashedPassword = hashPassword(password);
10       byte[] keePassFilePayload = marshallXml(keePassFile, header);
11       ByteArrayOutputStream streamToZip = compressStream(keePassFilePayload);
12       ByteArrayOutputStream streamToHashBlock = hashBlockStream(streamToZip);
13       ByteArrayOutputStream streamToEncrypt = combineHeaderAndContent(header,
             streamToHashBlock);
14       byte[] encryptedDatabase = encryptStream(header, hashedPassword, streamToEncrypt);
15       stream.write(encryptedDatabase);
16     } catch (IOException e) {
17       throw new KeePassDatabaseUnwriteableException("Could not write database file", e);
18     } finally {
19     }
20   }
21     private byte[] hashPassword(String password) throws UnsupportedEncodingException {
22     byte[] passwordBytes = password.getBytes(UTF_8);
23     return Sha256.hash(passwordBytes);
24   }
25   private byte[] encryptStream(KeePassHeader header, byte[] hashedPassword,
         ByteArrayOutputStream streamToEncrypt) throws IOException {
26     CryptoInformation cryptoInformation = new CryptoInformation(KeePassHeader.
           VERSION_SIGNATURE_LENGTH, header.getMasterSeed(), header.getTransformSeed(),
27     header.getEncryptionIV(), header.getTransformRounds(), header.getHeaderSize());
28     return new Decrypter().encryptDatabase(hashedPassword, cryptoInformation, streamToEncrypt.
           toByteArray());
29   }
30   ...
```

Listing 6.25: TF10: Openkeepass - Slice FULL NO_EXC_E (ohne Angabe mainclass) KeePassDatabaseWriter.java

Unter anderem ist mit Hilfe des Slices die Verarbeitung des Passworts nachvollziehbar. Mit dem Klartext-Passwort wird zunächst in der Methode `writeKeePassFile()` mit der Funktion `hashPassword()` und dem Algorithmus `Sha256` ein Hash erzeugt (vgl. Slice 6.25 Zeile 9 und 21-24). Später wird dieser Hash für die Verschlüsselung in der Methode `createAesKey()` ein zweites Mal mit `Sha256` gehasht, bevor er für die Transformieren des AES-Schlüssels verwendet wird in der Methode `createAesKey()` (vgl. Slice 6.26 Zeile 23-25).

```
1  public class Decrypter {
2
3    ...
4    public byte[] encryptDatabase(byte[] password, CryptoInformation cryptoInformation, byte[]
         database) throws IOException {
5      byte[] aesKey = createAesKey(password, cryptoInformation);
6      return processDatabaseEncryption(true, database, cryptoInformation, aesKey);
7    }
8    private byte[] processDatabaseEncryption(boolean encrypt, byte[] database,
         CryptoInformation cryptoInformation, byte[] aesKey) throws IOException {
9      byte[] metaData = new byte[cryptoInformation.getVersionSignatureLength() +
           cryptoInformation.getHeaderSize()];
10     SafeInputStream inputStream = new SafeInputStream(new BufferedInputStream(new
           ByteArrayInputStream(database)));
11     inputStream.readSafe(metaData);
12     byte[] payload = StreamUtils.toByteArray(inputStream);
13     if (encrypt) {
14       processedPayload = Aes.encrypt(aesKey, cryptoInformation.getEncryptionIV(), payload);
15     } else {
16       processedPayload = Aes.decrypt(aesKey, cryptoInformation.getEncryptionIV(), payload);
17     }
18     ByteArrayOutputStream output = new ByteArrayOutputStream();
19     output.write(metaData);
20     output.write(processedPayload);
21     return output.toByteArray();
22   }
23   private byte[] createAesKey(byte[] password, CryptoInformation cryptoInformation) {
24     byte[] hashedPwd = Sha256.hash(password);
25     byte[] transformedPwd = Aes.transformKey(cryptoInformation.getTransformSeed(), hashedPwd,
           cryptoInformation.getTransformRounds());
26     byte[] transformedHashedPwd = Sha256.hash(transformedPwd);
27     ByteArrayOutputStream stream = new ByteArrayOutputStream();
28     stream.write(cryptoInformation.getMasterSeed(), 0, 32);
29     stream.write(transformedHashedPwd, 0, 32);
30     return Sha256.hash(stream.toByteArray());
31   }
32 }
```

Listing 6.26: TF10: Openkeepass - Slice FULL NO_EXC_E Decrypter.java (ohne Angabe mainclass)

© Philip Phu Dang Hoan Nguyen

Weitere Informationen zur Parametrisierung der AES-Verschlüsselung lassen sich u.a. anhand des `KeePassHeader`-Objekts erkennen, welches in Zeile 26 bis auf den Parameter `VERSION_SIGNATURE_LENGTH` das `CryptoInformation` Objekt bildet und zusammen mit dem gehashten Passwort in Zeile 28 zum Verschlüsseln genutzt wird.

Die Erzeugung des `KeePassHeaders` erfolgt über eine dem Konstruktor übergebene Zufallszahl und erzeugt mit dieser u.a. den MasterSeed, den TransformSeed und den Intialisierungsvektor (IV) (vgl. Slice 6.27 der Klasse KeePassHeader Zeile 11-15). Aus dem Slice ist ebenfalls der Kompressions-Algorithmus `Gzip`, `Salsa20` als der Algorithmus für den Cyclic Redundancy Check (CRC) und die Rundenanzahl 8000 ersichtlich (Zeile 8-10).

```
1  public class KeePassHeader {
2
3    private static final byte[] DATABASE_V2_FILE_SIGNATURE_1 = ByteUtils.hexStringToByteArray("
         03d9a29a");
4    private static final byte[] DATABASE_V2_FILE_SIGNATURE_2 = ByteUtils.hexStringToByteArray("
         67fb4bb5");
5    private static final byte[] DATABASE_V2_FILE_VERSION = ByteUtils.hexStringToByteArray("
         00000300");
6    private static final byte[] DATABASE_V2_AES_CIPHER = ByteUtils.hexStringToByteArray("31
         C1F2E6BF714350BE5805216AFC5AFF");
7    public KeePassHeader(ByteGenerator byteGenerator) {
8      setCompression(CompressionAlgorithm.Gzip);
9      setCrsAlgorithm(CrsAlgorithm.Salsa20);
10     setTransformRounds(8000);
11     setMasterSeed(byteGenerator.getRandomBytes(32));
12     setTransformSeed(byteGenerator.getRandomBytes(32));
13     setEncryptionIV(byteGenerator.getRandomBytes(16));
14     setProtectedStreamKey(byteGenerator.getRandomBytes(32));
15     setStreamStartBytes(byteGenerator.getRandomBytes(32));
16     setCipher(DATABASE_V2_AES_CIPHER);
17     ...
```

Listing 6.27: TF10: Openkeepass - Slice FULL NO_EXC_E KeePassHeader.java (ohne Angabe mainclass)

Die Implementierung der `RandomGenerator.java` zeigt abschließend, dass als Zufallsgenerator für den `KeePassHeaders` die Bibliothek `java.security.SecureRandom` verwendet wurde mit dem Algorithmus `SHA1PRNG`.

Aufgrund der korrekten Slicing-Ergebnisse, war es im Zuge der Evaluation möglich, eine grobe Sicherheitsanalyse auf einen reduzierten Auszug des Original-Quellcodes durchzuführen. Die Slicingdauer von knapp über *5 Minuten* ist für eine Reduktion des Quellcodes um *84%* akzeptabel, weswegen die Ergebnisse dieses Testfalls als positiv gewertet werden.

6.3.2. TF11: OSCI

Jar:	`osci-bibliothek.jar, egov_withexample.jar`
Caller:	`close`
Callee:	`doFinal`
Datenabhängigkeit:	`FULL, NO_HEAP_NO_EXCEPTIONS, NO_EXCEPTIONS`
Kontrollabhängigkeit:	`NO_EXCEPTIONAL_EDGES`
Weitere Optionen:	`(mainclass = Lde/osci/helper/SymCipherOutputStream), (mainclass = Lde/osci/osci12/samples/OneWayMessage_PassiveRecipient), advanced_mode = true, object_inst_methods = getInstance, multiple_caller = true, only_public_entry = true`

OSCI ist ein Java-Protokollstandard für die sichere, vertrauliche und rechtsverbindliche Übertragung elektronischer Daten im E-Government [KoSIT 2018]. Das Protokoll bietet die technische Basis für eine sichere Übermittlung von Nachrichten, um Sicherheitsziele wie Integrität, Authentizität, Vertraulichkeit und Nachvollziehbarkeit zu gewährleisten. Es wurde vom Bundesministerium des Inneren im Rahmen von SAGA als obligatorischer Standard für elektronische Transaktionen mit der Bundesverwaltung gesetzt und wird im Besonderen in unsicheren Netzen wie dem Internet, jedoch ebenso in sicheren Netzen für ergänzende Funktionalitäten genutzt.

Der *OSCI*-Transport basiert auf den Standards Extensible Markup Language (XML) und Simple Object Access Protocol (SOAP) und bietet laut der Koordinierungsstelle für IT-Standards (KoSIT) transparente Verschlüsselungsverfahren und von der W3C empfohlene Verfahren zur digitalen Signatur. Diese bieten Möglichkeiten, Verwaltungsdienstleistungen authentisiert und identifiziert zwischen der öffentlichen Verwaltung und Kunden (Bürger und Unternehmen) oder zwischen den verschiedenen Verwaltungsabteilungen durchführen zu können. Da im Rahmen dieser Arbeit eine detaillierte Erläuterung der Funktionsweise von *OSCI* nicht möglich ist, beschränken sich der Testfall und die weiteren Erläuterungen zum größten Teil auf das Slicing und die Verwendung der Sicherheitsfunktionen.

Im folgenden Testfall soll die *OSCI*-Bibliothek (Version 1.2) (ca. 20.000 LoC) erneut nach dem Methodenaufruf `doFinal()` gesliced werden. Der Aufruf findet sich in der Bibliothek ein Mal in der Methode `doRSADecryption()` der Klasse `Crypto (src/de/osci/osci12/encryption/Crypto.java)` und ein Mal in der Methode `close()` der Klasse `SymCipherOutputStream` (`de/osci/helper/SymCipherOutputStream.java`). Da die Methode `doRSADecryption()`, wie der Name verrät, für die Entschlüsselung zuständig ist, wird der Fokus dieses Testfalls auf die Verschlüsselung

 © Philip Phu Dang Hoan Nguyen

in `SymCipherOutputStream` gesetzt und dementsprechend der Caller und Callee gewählt. Ein Slicing der Entschlüsselungsfunktion mit `doRSADecryption()` wurde im Rahmen der Evaluation dessen ungeachtet zusätzlich durchgeführt und zeigt ähnliche Resultate wie der folgende Testfall. Der Slice ist dem Anhang B.16 beigefügt und weist generell eine ähnliche Struktur wie im Testfall *TF10* auf, mit den Aufrufen `getInstance(), init()` und `doFinal()`. Auf weitere Ausführungen wird in diesem Fall verzichtet.

Das Ausführen des Slicings (Dauer ca. 10 Sekunden) mit den Optionen `FULL NO_EXCEPTIONAL_EDGES` und der Angabe der Caller-Klasse `SymCipherOutputStream` als `mainclass` ergibt, wie im Testfall *TF10*, zunächst keine aussagekräftigen und sicherheitsrelevanten Ergebnisse (vgl. Slice 6.28). Andererseits kann der Test genutzt werden, um das Finden des Callee und Callers zu prüfen und zusätzlich zu testen, ob der Slice trotz besonderer Programmier-Konventionen syntaktisch korrekt ist. Erste Tests schlugen beispielsweise anfangs fehl, weil durch den Eintrag `java.io` in der Exclusionfile die Klasse `SymCipherOutputStream` ignoriert wurde, die von dieser Klasse erbt.

```
1  public class SymCipherOutputStream extends FilterOutputStream
2  {
3
4    // private static Log log = LogFactory.getLog(SymCipherOutputStream.class);
5    @Override
6    public void close() throws IOException
7    {
8      try
9      {
10       out.write(cipher.doFinal());
11     }
12   }
13 }
```

Listing 6.28: TF11: OSCI - Slice FULL NO_EXC_E (mit Angabe mainclass) SymCipherOutputStream.java

Eine genaue Betrachtung des Original-Quellcodes zeigt auf, dass die Methode `close()` bis auf eine Anweisung und Fehlerbehandlung vollständig ist. Die Erzeugung und Initialisierung des `Cipher`-Objekts findet im Konstruktor von `SymCipherOutputStream` statt, während das blockweise Verschlüsseln, mit Hilfe der `Cipher`-Methode `update()`, in der Methode `write()` durchgeführt wird.

Damit die Datenflussanalyse interprozedural durchgeführt wird, müssen wie im Testfall *TF10*, mit Anpassung der Option `mainclass`, zunächst sinnvolle Programm-Einstiegspunkte definiert werden. Um eine Anwendung zu simulieren, welche Funk-

tionen von *OSCI* implementiert, wurden alle mitgelieferten Beispielanwendungen bzw. Klassen wie

- `OneWayMessage_ActiveRecipient`,
- `OneWayMessage_PassiveRecipient`,
- `PartialOneWayMessage_ActiveRecipient`,
- `AuthorOriginatorContentInterchange`
- und `RequestResponse`

der Bibliothek ergänzt, gebaut und in das Java-Archive `egov_withexample.jar` gebündelt. Weitere Tests wurden auf Basis dieser ergänzen Jar ausgeführt.

Zunächst muss die Parametrisierung des Auditors für diesen Testfall evaluiert werden, da das Nutzen falscher Optionen zu langen Ausführungszeiten und zum Überlaufen des Speichers führen kann. Wird die Option `mainclass` entfernt und die Entrypoints über `makeMainEntrypoints()` erzeugt, führen die Datenabhängigkeitsoptionen `FULL`, `NO_HEAP` oder `NO_EXCEPTION` beim Slicing aufgrund eines zu großen SDGs (beim Erzeugen oder beim Verarbeiten) den Speicher zum Überlaufen mit der Meldung:

```
1 Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
```

Anders als im Testfall *TF10*, bewirkt das Festlegen der `mainclass` mit der Klasse einer Beispielanwendung (bspw. `Lde/osci/osci12/samples/OneWayMessage_PassiveRecipient`), dass weniger Programm-Einstiegspunkte ermittelt werden. Während ohne Angabe der `mainclass` die Methode `makeMainEntrypoints()` fünf Entrypoints ermittelt werden, wird mit Angabe einer Beispielanwendung die Zahl auf drei reduziert und zur Folge das Slicing beschleunigt. Wie beschrieben liefert OSCI exakt fünf Beispielklassen, welche jeweils eine `main()`-Methode definieren. Mit dem Entfernen nicht benötigter Beispiele könnte der Testfall auf einen Entrypoint reduziert werden. Da bei einer Sicherheitsanalyse jedoch nicht zwangsläufig die Möglichkeit gegeben ist, ein Jar-Archive nach Belieben zu modifizieren, soll der aktuelle Umstand als realitätsnahes Szenario genutzt werden.

Ein akzeptables Ergebnis wurde mit zwei verschiedenen Varianten erzielt. Zum einen kann mit Angabe der Beispielanwendung als `mainclass` (somit Analyse dreier Entrypoints) und der Datenabhängigkeitsoptionen `NO_EXCEPTION` (abgekürzt `NO_EXC`) in 193 Minuten erfolgreich ein Slice erzeugt werden (vgl. Tabelle 6.2). Alternativ kann mit der Option `NO_HEAP_NO_EXCEPTION` (abgekürzt `NH_NX`) in wesentlich kürzerer

Zeit ein Slice erzeugt werden. Diese Variante benötigt für einen SDG mit 16.385 Knoten und 28.604 Kanten zehn Sekunden und gibt im Resultat 1.457 Slice-Anweisungen aus. Aus dem Slice können hierbei 26 Java-Dateien der `OSCI-Bibliothek` rekonstruiert werden.

Variante:	**1: NX_NH**	**2: NO_EXC**
Ausführungszeit:	ca. 10 s	ca. 200 min
SDG Knoten:	16.385	457.630
SDG Kanten:	28.604	979.362
Statements im Slice:	1.457	145.303
LoC (% vom Code):	717 (∼4%)	2.299 (∼12%)
Betroffene Klassen:	26	47

Tabelle 6.2.: Vergleich Slicing mit verschiedenen Datenabhängigkeitsoptionen

Bei der Betrachtung der Ergebnisse fällt ins Auge, dass sich die Werte der beiden Slicing-Ergebnisse stark unterscheiden. Besonders in der Ausführungszeit, der Größe des erzeugten SDGs (in beiden Fällen entstehend aus einem Callgraphen mit 750 Knoten) und der resultierenden Anzahl der Anweisungen im Slice gibt es extreme Abweichungen. Dies lässt sich ohne weitere Analyse auf den stark reduzierten SDG zurückführen, welcher durch die verwendete Option `NH_NX` alle Datenabhängigkeiten zu `try-catch`-Knoten und zusätzlich Abhängigkeiten von und zum *Heap* ignoriert.

Einzig die Anzahl der Programmzeilen, die nach dem Slicing noch als Slice verbleibt, entspricht nicht den Erwartungen und ist mit einer Reduktion des Quellcodes um 88 % und 96 % nicht übermäßig unterschiedlich. Betrachtet man die Ergebnisse der langsamen, aber genaueren Variante `NO_EXC` näher, können 21 zusätzliche OSCI-Klassen und 26 weitere Klassen aus anderen Bibliotheken identifiziert werden, die in der schnellen Variante fehlen und auf denen sich vermutlich die fehlenden Slice-Anweisungen verteilen.

Allein wegen der langen Ausführungszeit der Variante `NO_EXC` stellt sich somit die Fragestellung, ob für eine Sicherheitsanalyse die Heap-Informationen zwingend notwendig sind, oder ob schnelles Slicing mit der Option `NH_NX` hinreichende Informationen zum Datenfluss liefert. Aus weiteren Tests mit nur zwei zusätzlichen Entrypoints (Algorithmus `makeMainEntrypoints()`) zeigte sich, dass der SDG zwar noch erstellt werden kann, mit über 1,6 Millionen Knoten und über 3,4 Millionen Kanten mit den verfügbaren Mitteln nicht mehr zu verarbeiten ist. Der gleiche Testfall, ausgeführt

mit der Option `NH_NX`, verlängerte die Ausführungszeit im Kontrast dazu nur um eine Sekunde.

Ähnlich wie im vorherigen Testfall *TF10*, wird anhand der Slices versucht, die Datenflüsse von der Anwendung bis zur Verschlüsselung (`doFinal()`) nachzuvollziehen. Dies soll soweit möglich, anhand der Slices der ersten schnellen Variante `NH_NX` geschehen. Fehlen Informationen zur Analyse der Datenflüsse, soll auf die Slices der zweiten Variante `NO_EXC` zurückgegriffen werden.

Im Folgenden soll aus diesem Grund exemplarisch auf den Datenfluss der Beispielanwendung *OneWayMessage_PassiveRecipient* eingegangen werden, welche einfache synchrone Kommunikation nach *OSCI* 1.2 Spezifikationen [OSCI Leitstelle 2002b] nachbilden soll. Die Anwendung sendet eine einmalige Nachricht an einen passiven Empfänger, wobei im *OSCI*-Kommunikationsmodell ein sog. *Intermediär* als zentrale Vermittlungsstelle zwischen den Kommunikationspartnern genutzt wird. Dieser ist zur Realisierung asynchroner Kommunikation notwendig, wenn Sender und Empfänger nicht zeitgleich online sind und wird zusätzlich für die Gewährleistung der Rechtsverbindlichkeit und Vertraulichkeit über Signaturen, Zertifikate oder Public-Key-Kryptografie eingesetzt (siehe Abschnitt 2.2 und Abschnitt 2.3.3).

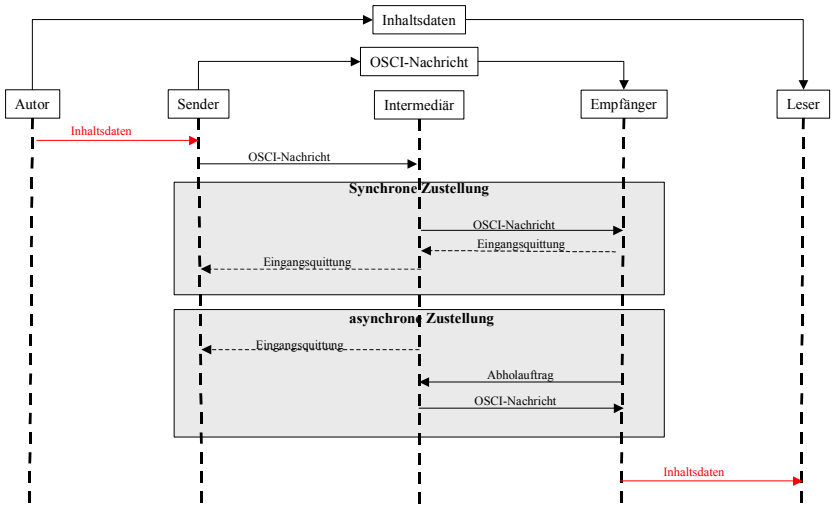

Abbildung 6.3.: TF11: OSCI-Kommunikationsmodell (Quelle: [OSCI Leitstelle 2002a])

Die Abbildung 6.3 zeigt das Kommunikationsmodell mit dem *Intermediär* als Vermittlungsstelle für synchrone und asynchrone Zustellung. Dieser ist, wie ebenfalls zu erkennen, zuständig für die Verarbeitung von Eingangsquittierungen und Abholaufträge. Da es sich bei `OneWayMessage_PassiveRecipient.java` um einen passiven Empfänger handelt, der nicht aktiv die Daten abholt [OSCI Leitstelle 2002b], wird

dementsprechend das obere Szenario der synchronen Zustellung nachgestellt (vgl. vollständiger Quellcode B.15).

Zunächst sind im Slice 6.29 vom OneWayMessage_PassiveRecipient übersichtlich die wesentlichen Anweisungen zu sehen, welche eine Relevanz zu `doFinal()` haben müssen. In der Zeile 9 ist noch ein Parse-Fehler erkennbar, welcher ähnlich der mehrzeiligen Klassen- oder Methodenköpfe behandelt werden müsste. Ferner ergründet eine genauere Analyse, dass die Zeile 7 bis 12 zwei wichtige Schritte des genannten Szenarios widerspiegeln. `getMsgID.send()` sendet zunächst eine Nachricht an den `Intermediär`, um für diese Kommunikation mit dem Empfänger eine sog. `MessageId` zu erhalten. Die `MessageId` ist weltweit und zeitlich unbegrenzt eindeutig und wird ausschließlich vom *Intermediär* vergeben [OSCI Leitstelle 2002b].

```
1  public class OneWayMessage_PassiveRecipient
2  {
3    public ResponseToForwardDelivery sendForwardDelivery(String urlRecipient)
4                                          throws ...
5    {
6      DialogHandler clientDialog = new DialogHandler(user_1, intermed, new de.osci.osci12.
           samples.impl.HttpTransport());
7      GetMessageId getMsgID = new GetMessageId(clientDialog);
8      ResponseToGetMessageId rsp2GetMsgID = getMsgID.send();
9      Addressee user_2 = new Addressee(null,
10     ForwardDelivery forwardDel = new ForwardDelivery(clientDialog, user_2, urlRecipient,
           rsp2GetMsgID.getMessageId());
11     data.addContent(new Content("Any content data."));
12     ResponseToForwardDelivery rsp2FwdDel = forwardDel.send();
13   }
14   public static void main(String[] args)
15   {
16     try
17     {
18       OneWayMessage_PassiveRecipient scenario_2 = new OneWayMessage_PassiveRecipient(args[0]);
19       ResponseToForwardDelivery responseForward = scenario_2.sendForwardDelivery(args[1]);
20     }
21   }
22 }
```

Listing 6.29: TF11: OSCI - Slice NH_NX NO_EXC_E OneWayMessage_PassiveRecipient.java

Die Anweisung `forwardDel.send()` sendet anschließend nach Erhalt einer `MessageId` einen Weiterleitungsauftrag (mit der MessageId als Kenner) an den *Intermediär*, welcher im Folgeschritt den Annahmeauftrag mit der Zustellung an den Empfänger schickt und auf eine Annahmeantwort wartet. Der `Intermediär` notiert hierbei den Empfang, die Weiterleitung der Nachricht und ebenfalls den Erhalt der Annahmeant-

wort auf einem „Laufzettel“ bevor dieser als Weiterleitungsantwort (`ResponseToForwardDelivery`) an den Empfänger zurückgesendet wird.

Für die Sicherheitsanalyse ist in diesem Zusammenhang das Objekts `DialogHandler` interessant, da bei der Erzeugung Informationen zur Signatur und der Verschlüsselung bereitstellt und ebenfalls definiert wird, ob Sicherheitsmechanismen aktiv sind. Die Instanziierung ist in diesem Fall zwar sichtbar (Zeile 6), doch fehlen die Variablen `user_1` und `intermed`, welche Rückschlüsse auf die Zertifikate geben. Die Informationen sind in der Variante `NO_EXC` vorhanden (vgl. Slice B.15 Zeile 6-8, 13).

Die Abbildung 6.4 zeigt als Übersicht den kompletten evaluierten Datenfluss bis zum Aufruf der Verschlüsselung.

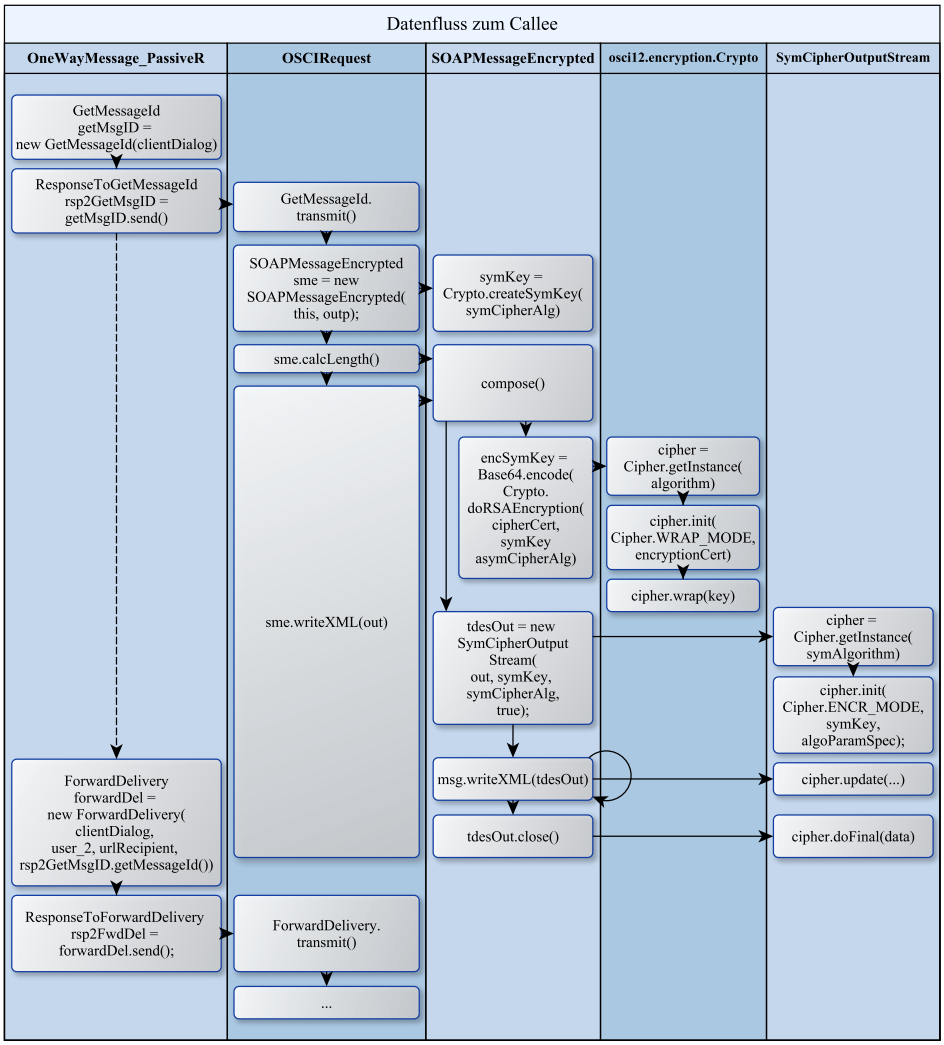

Abbildung 6.4.: TF11: Datenflüsse zwischen OneWayMessage_PassiveRecipient und doFinal()

© Philip Phu Dang Hoan Nguyen

Die Aufrufe `getMsgID.send()` und `forwardDel.send()` im Slice betrachtend, fällt auf, dass die Klassen `GetMessageId` und `ForwardDelivery` von der Klasse `OSCIRequest` erben und beide die Funktion `transmit()` aufrufen, welche in der Basisklasse enthalten ist. Da der Aufrufpfad somit für das `ForwardDelivery`-Objekt ab einem Knoten analog verläuft, soll im Folgenden einzig das erste Senden des MessageId-Anforderungsantrags betrachtet werden. Der Datenfluss vom Senden des Weiterleitungsauftrags wird somit im nächsten Abschnitt nicht weiterverfolgt, in der Abbildung jedoch noch angedeutet.

Ein Auszug vom Slice der Klasse `OSCIRequest` wird im Slice 6.30 dargestellt und zeigt den Methodenaufruf `transmit()` (vgl. vollständiger Slice B.17). Wie beschrieben legt das `DialogHandler`-Objekt fest, ob bspw. in Zeile 7-8 eine Signatur oder in der Zeile 11-16 die Verschlüsselung angestoßen wird. Eine Analyse der Slices ergibt, dass bei der Erzeugung des Objekts für diesen Testfall die Voreinstellungen genutzt werden und daraus folgend die Verwendung von Signaturen und Verschlüsselung aktiv ist.

```
1  public abstract class OSCIRequest extends OSCIMessage
2  {
3    // private static Log log = LogFactory.getLog(OSCIRequest.class);
4    protected OSCIMessage transmit(OutputStream outp, OutputStream inp) throws ...
5    {
6      ...
7      if (dialogHandler.isCreateSignatures())
8        sign();
9      try
10     {
11       if (dialogHandler.isEncryption())
12       {
13         SOAPMessageEncrypted sme = new SOAPMessageEncrypted(this, outp);
14         out = transport.getConnection(uri, sme.calcLength());
15         sme.writeXML(out);
16       }
17       ...
18     }
19     ...
20   }
21   ...
22 }
```

Listing 6.30: TF11: OSCI - Auszug Slice NH_NX NO_EXC_E OSCIRequest.java

Die Einsicht in die Klassen `DialogHandler` (vgl. Quellcode 6.31) und `Constants` (vgl. Quellcode 6.32) zeigt grob, welche kryptografischen Algorithmen für die Anwendung genutzt werden bzw. genutzt werden können. Die hier aufgeführten relevanten Informationen zum `DialogHandler` sind nur im Slice der langsamen Variante

`NO_EXC` vorhanden (ohne Kommentare), während die Informationen zu der Klasse `Contants` in beiden Slices fehlt.

```
1  ...
2   /** Hashalgortihmus (Voreinstellung SHA-256) */
3  private static String digestAlgorithm = Constants.DIGEST_ALGORITHM_SHA256;
4
5  /** Symmetrischer Verschlüsselungsalgorithmus (Nachrichtenverschlüsselung) */
6  private String symmetricCipherAlgorithm = Constants.DEFAULT_SYMMETRIC_CIPHER_ALGORITHM;
7
8  /** Asymmetrischer Verschlüsselungsalgorithmus (Verschlüsselung des Sitzungsschlüssels) */
9  private String asymmetricCipherAlgorithm = Constants.ASYMMETRIC_CIPHER_ALGORITHM_RSA_1_5;
10
11 /** Zufallsgenerator-Algorithmus */
12 private static String secureRandomAlgorithm = Constants.SECURE_RANDOM_ALGORITHM_SHA1;
13
14 /** Signaturalgorithmus (Nachrichtensignatur) */
15 private static String signatureAlgorithm = Constants.SIGNATURE_ALGORITHM_RSA_SHA256;
16 ...
```

Listing 6.31: TF11: OSCI - Auszug Original-Quellcode DialogHandler.java

```
1  ...
2  public static final String DIGEST_ALGORITHM_SHA256 = "http://www.w3.org/2001/04/xmlenc#sha256
       ";
3
4  public static final String DEFAULT_SYMMETRIC_CIPHER_ALGORITHM = Constants.
       SYMMETRIC_CIPHER_ALGORITHM_AES256;
5
6  @Deprecated
7  public static final String SYMMETRIC_CIPHER_ALGORITHM_AES256 = "http://www.w3.org/2001/04/
       xmlenc#aes256-cbc";
8
9  public static final String SECURE_RANDOM_ALGORITHM_SHA1 = "SHA1PRNG";
10 ...
```

Listing 6.32: TF11: OSCI - Auszug Original-Quellcode Constants.java

Interessant ist hierbei die Erkenntnis, dass zum Zeitpunkt der Erstellung dieser Arbeit, die aktuellste Version von OSCI (1.2) als Default-Algorithmus zur symmetrischen Verschlüsselung AES mit 256 Bit Schlüssellänge und Betriebsmodus *CBC* nutzt (vgl. Quellcode Constants 6.32 Zeile 4-7). Zwar wird dieser Algorithmus als „veraltet“ markiert und AES alternativ ebenfalls mit *GCM* angeboten, jedoch verbleibt die Voreinstellung in dieser Version und wird somit in diesem Testfall genutzt. Das IT-Sicherheits-Beratungsunternehmen *SEC Consult* veröffentlichte hierzu Mitte 2017 einen Bericht, der kritische Sicherheitslücken in der OSCI-Kommunikation aufdeckte [SEC Consult 2017]. Ursache für diese Sicherheitslücken war eine falsche

Verwendung von Blockchiffren im CDB-Modus, welche in bestimmten Szenarien anfällig für sogenannte *Padding Oracle*-Angriffe ist. SEC Consult beschreibt ferner:

> „Einem Angreifer ist es so möglich, durch wiederholtes Senden besonders gewählter Nachrichten, den Empfänger oder Intermediär dazu zu bringen den Klartext preiszugeben. Dieser Angriff erlaubt es einem Angreifer an die Metadaten des Auftrags zu gelangen.“ [SEC Consult 2017]

Der Herausgeber des Standards World Wide Web Consortium (W3C) empfiehlt u.a. aufgrund dessen seit 2013 diese Algorithmen nicht mehr zu verwenden, da gewisse Sicherheitsrisiken bestehen können.

Da geklärt werden konnte, dass die Verschlüsselung aktiv ist, kann der weitere Datenfluss wie folgt beschrieben werden. In der Zeile 13-15 der Klasse `OSCIRequest` (vgl. Slice 6.30) wird in der Methode `transmit()` ein Objekt der Klasse `SOAPMessageEncrypted` instanziiert und in den Folgeanweisungen u.a. mit den Methoden `calcLength()` und `writeXML()` aufgerufen.

Bei der Analyse der Datenflüsse fällt hierbei auf, dass die Erzeugung des symmetrischen Schlüssels `symKey` mit Hilfe der Slices der ersten Variante `NH_NX` nicht nachvollzogen werden kann. Dies ist auf einen Mangel des Slice der Klasse `SOAPMessageEncrypted` (vgl. vollständiger Slice B.18) zurückzuführen, in welchem die Konstruktor-Methode nicht gesliced wurde. Dementsprechend fehlt dem Slice aus dem Konstruktor ebenfalls der Aufruf:

```
1 symKey = Crypto.createSymKey(symmetricCipherAlgorithm);
```

Der Konstruktor ist hingegen im Slice der zweiten Variante `NO_EXC` vorhanden (vgl. Slice B.19 Zeile 10-20). Der Slice der ersten Variante enthält allerdings die Methoden `calcLength(), compose()` und `writeXML()`, mit dessen Hilfe u.a. `symKey` und letztendlich ebenfalls die Daten verschlüsselt werden.

Die Verschlüsselung des Schlüssels geschieht in `compose()` mit der Anweisung

```
1 encSymKey = Base64.encode(Crypto.doRSAEncryption(cipherCert, symKey, msg.getDialogHandler().
      getAsymmetricCipherAlgorithm()))
```

, die zunächst die Methode `doRSAEncryption()` der OSCI-Klasse `Crypto` (`de/osci/osci12/encryption/Crypto.java`) aufruft mit einem Zertifikat, dem symmetrischen Schlüssel und einem asymmetrischen Verschlüsselungsalgorithmus (vgl. Abbildung 6.4). Es erschließt sich aus dem Aufruf, dass wie bereits im Quellcode-

auszug 6.31 erwähnt, der asymmetrische Verschlüsselungsalgorithmus RSA in der Version 1.5 benutzt wird, um den *symmetrischen* (Sitzungs-)Schlüssel mit Hilfe des öffentlichen Schlüssels vom Verschlüsselungs-Zertifikat *asymmetrisch* zu verschlüsseln. Dass es sich bei der Variable `cipherCert` um das Verschlüsselungszertifikat des Intermediär handelt, kann einzig mit Verwendung der Slices der ersten Variante nicht erschlossen werden. Eine Analyse der zweiten Variante `NO_EXC` zeigt, dass beim Erzeugen des `DialogHandlers` der Intermediär mit Zertifikat gesetzt wird und bestätigt dementsprechend diese Vermutung.

Für die Verschlüsselung des Sitzungsschlüssels in der Klasse `Crypto` wurde der Verschlüsselungsmodus `WRAP_MODE` der Klasse `Cipher` genutzt, welcher anstatt `doFinal()` den Schlüssel direkt mit dem Methodenaufruf `wrap(key)` verschlüsselt (vgl. Sliceauszug 6.33 Zeile 26).

```
1  public class Crypto
2  {
3
4    public static javax.crypto.SecretKey createSymKey(String algorithm) throws ...
5    {
6      return keyGenerator.generateKey();
7    }
8    public static byte[] doRSAEncryption(java.security.cert.X509Certificate encryptionCert,
9                        Key key,
10                       String algorithm) throw ...
11   {
12     try
13     {
14       if (DialogHandler.getSecurityProvider() == null)
15         cipher = javax.crypto.Cipher.getInstance(Constants.JCA_JCE_MAP.get(algorithm));
16       else
17         cipher = javax.crypto.Cipher.getInstance(Constants.JCA_JCE_MAP.get(algorithm),
18                                     DialogHandler.getSecurityProvider());
19       if (Constants.ASYMMETRIC_CIPHER_ALGORITHM_RSA_OAEP.equals(algorithm))
20       {
21         OAEPParameterSpec oaepParameters = new OAEPParameterSpec(digestAlgorithm, "MGF1",
               mgfParameterSpec,
22         cipher.init(Cipher.WRAP_MODE, encryptionCert.getPublicKey(), oaepParameters);
23       }
24       else
25         cipher.init(Cipher.WRAP_MODE, encryptionCert);
26       return cipher.wrap(key);
27     }
28   }
29     ...
30 }
```

Listing 6.33: TF11: OSCI - Auszug Slice NH_NX NO_EXC_E Crypto.java

© Philip Phu Dang Hoan Nguyen

Bezüglich Korrektheit der Syntax ist in Zeile 21 erneut ein Parser-Fehler zu erkennen, ähnlich dem Fehler im Slice 6.29.

Die Aufrufe von der Methode `getInstance()` in den Zeilen 15 und 17 zeigen, dass der Verschlüsselungsalgorithmus über den übergebenen Parameter `algorithm` und einem Dictionary der Klasse `Constants` definiert wird. Der Auszug des Slices der ersten Variante `NH_NX` zeigt ebenfalls farblich markiert das Fehlen der Anweisungen in zur Initialisierung des Cipher-Objekts (`cipher.init()` in den Zeilen 19-25). Diese gibt Ausschluss darüber, dass zum einen der Modus `Cipher.WRAP_MODE` und zum anderen die Variable `encryptionCert` genutzt wird. Die genannten Informationen sind hingegen im Slice der zweite Variante `NO_EXC` vorhanden.

Nachdem der Sitzungsschlüssel erzeugt und asymmetrisch verschlüsselt wurde, erzeugt die Methode `writeXML()` nach der Ausführung von `compose()` ein Objekt der Klasse `SymCipherOutputStream`.

Der aus dem Slicing der ersten Variante `NH_NX` resultierende Slice der Klasse `SymCipherOutputStream` enthält einzig die Methode `close()`, die auf das `Cipher`-Objekt `cipher doFinal()` aufruft. Der Slice 6.34 zeigt, dass die Methode `close()` bis auf die Exception-Behandlung und die Anweisung `out.flush()` (Zeile 12-20) vollständig ist.

```
1  public class SymCipherOutputStream extends FilterOutputStream
2  {
3
4    // private static Log log = LogFactory.getLog(SymCipherOutputStream.class);
5    @Override
6    public void close() throws IOException
7    {
8      try
9      {
10       out.write(cipher.doFinal());
11     }
12     catch (Exception ex)
13     {
14       if (encrypt)
15         throw new IOException(DialogHandler.text.getString("encryption_error"));
16       else
17         throw new IOException(DialogHandler.text.getString("decryption_error"));
18     }
19
20     out.flush();
21   }
22 }
```

Listing 6.34: TF11: OSCI - Slice NH_NX NO_EXC_E SymCipherOutputStream.java

Wie das `Cipher`-Objekt erzeugt und initialisiert wurde, ist nur mit Hilfe des Slices der zweiten Variante `NO_EX` erkennbar, da die jeweiligen Anweisungen zum einen im Konstruktor und der Methode `write()` der Klasse `SymCipherOutputStream` ausgeführt werden. Der Slice B.20 im Anhang zeigt weitere fehlende Informationen zur symmetrischen Verschlüsselung der Daten. Anhand der beiden Methoden ist beispielsweise zu erkennen, wie `Cipher.getInstance()` in den Zeilen 25 und 29 und `cipher.update()` in Zeile 81 aufgerufen werden.

Mit Abschluss der Analyse der Klasse `SymCipherOutputStream`, endet ebenfalls der zu analysierende Datenfluss, der am Anfang des Testfalls mit der Abbildung 6.4 aufgezeigt wurde. Mit der durchgeführten Analyse konnte demzufolge mit Ausnahmen der Datenfluss von der Testanwendung `OneWayMessage_PassiveRecipient` bis zu den Quellcodestellen mit Verschlüsselungsfunktionalität verfolgt und analysiert werden. Während bei Verwendung der Slicing-Ergebnisse der schnellen Variante `NH_NX` gewisse Informationen fehlten, konnte der komplette Datenfluss mit den Ergebnissen der zweiten Variante `NO_EX` nachvollzogen werden.

Auf die genaue Funktionsweise der Verschlüsselung des Java `OutputStreams` innerhalb der Klassen `SymCipherOutputStream, OSCIRequest` und `SOAPMessage-Encrypted` soll im Rahmen dieser Arbeit nicht weiter Bezug genommen werden.

6.3.3. Zusammenfassung

Der zweite Teil der Evaluierung testete das Verhalten des Auditors im Zusammenspiel mit größeren Anwendungen bzw. Bibliotheken (*openkeepass* und *OSCI*). Zwar lag der Fokus in diesem Teil der Evaluation nicht mehr explizit auf den Optimierungen, welche im ersten Teil bereits untersucht wurden, doch fielen bei der Evaluation der beiden Anwendungen noch einige Parser-Fehler auf, die bis zur Veröffentlichung der Arbeit nicht mehr korrigiert werden konnten.

Ansonsten wurde bei der Evaluation dieser Anwendungen deutlich, dass die Angabe der Option `mainclass` und somit die Wahl des Algorithmus zur Bestimmung der Programm-Einstiegspunkte, beim Slicing ausschlaggebend für die Komplexität und Ausführungszeit ist.

Während das Slicing im Testfall *TF10*, mit den Slicingoptionen `FULL NO_EXCEPTIONAL_EDGES` bei unterschiedlicher Anzahl von Entrypoints zwischen 5 und 20 Minuten benötigte, konnte bei OSCI im Testfall TF11 bei falscher Parameterwahl kein Slice in akzeptierbarer Zeit erzeugt werden. Es wurde evaluiert, dass für ein effizientes

Slicing für jede Anwendung der effizienteste Algorithmus zur Bestimmung der Entrypoints erprobt werden muss. Um den Callgraph nach Möglichkeit klein zu halten, sollte im Idealfall wie in *TF10* exakt ein Entrypoint in die Analyse einfließen.

Mit den Slicing-Ergebnissen im Testfall *TF10* konnte letztendlich der gesuchte Datenfluss im Slice erkannt und ebenfalls zielgerichtet analysiert werden.

Die Slicing-Ergebnisse im Testfall *TF11* wurden konträr dazu mit zwei verschiedenen Varianten erzeugt, um den Trade-off zwischen Ausführungszeit und Detailgrad des Slices zu evaluieren. Der schnellen Slicing-Variante (10 Sekunden) mit der Datenabhängigkeitsoption `NO_HEAP_NO_EXCEPTIONS` steht die detaillierte, jedoch langsame Variante (ca. 200 Minuten) mit der Option `NO_EXCEPTIONS` gegenüber. Hierbei sollte überprüft werden, ob die Ergebnisse der schnellen Variante für eine Sicherheitsanalyse und folglich für das Verfolgen eines Datenflusses ausreichend sind.

Daraus resultierend konnten ebenso mit den Ergebnissen der schnellen Variante die grundlegenden Datenflüsse nachvollzogen werden. Einschränkungen wurden in einigen Fällen erkannt, bei denen zusätzliche Informationen, bspw. aus Parametern gefehlt haben, welche aus der Instanziierung von Objekten entstanden sind. Dies entspricht den Erwartungen, da das Ignorieren der Datenabhängigkeiten von und zum Heap genau dazu führt, dass Instanzvariablen und die Initialisierung dieser durch Konstruktoren wegfallen.

Es fehlten infolgedessen beispielsweise Instanzvariablen und somit ebenfalls Objekte, die Rückschlüsse auf die Herkunft der Zertifikate, Definition der Verschlüsselungsalgorithmen oder die Erstellung des symmetrischen Schlüssels liefern. Ferner fehlten in der Klasse `SymCipherOutputStream` zum Aufruf `cipher.doFinal()` die Erzeugung des `Cipher`-Objekts und dessen `update()`-Aufruf.

6.4. Funktionsfähigkeit des Auditors

Mit Abschluss der umfangreichen Evaluation konnte grundsätzlich gezeigt werden, dass der Auditor für Sicherheitsanalysen einsetzbar ist. Besonders der erste Teil der Evaluation deckte mit den Testfällen *TF01* bis *TF09* viele funktionale Anforderungen ab, die im Abschnitt 5.1 beschrieben wurden. Folglich wurden alle genannten Systemfunktionen in den Auditor implementiert und verifiziert.

Die Funktionalität des Parsers, welche die resultierenden Slice-Anweisungen in korrekten Quellcode zurückführt, konnte ebenfalls gewährleistet werden. Die resultierenden Slices aller Testfälle waren bis auf wenige Ausnahmen syntaktisch korrekt, lesbar und konnten für Analysen genutzt werden.

Mit Hilfe der umfassenden Konfigurationsdatei und den eingeführten Erweiterungen (siehe Abschnitt 5.3.3) ist es einem Sicherheits-Experten zudem möglich, die Parameter für das Slicing, entsprechend der zu analysierenden Anwendung, beliebig anzupassen. Da abhängig der Größe und Komplexität der Anwendung das Slicing schnell zu rechenintensiv werden kann, sind für eine effiziente Analyse genaue Feineinstellungen unerlässlich.

Der zweite Teil der Evaluation zeigte diesbezüglich, dass der Auditor mit Hilfe dieser Konfigurationsdateien beiläufig eine gewisse Skalierbarkeit gewährleistet. Der Auditor war mit korrekter Parametrisierung in der Lage die kleinen funktionalen Tests auf zwei größere Anwendungen (6.000-20.000 LoC) zu skalieren.

Einschränkungen

Wie in der Arbeit geschildert, handelt es sich beim Auditor trotz positiver Ergebnisse um einen fortgeschrittenen Prototyp.

Neben den im Abschnitt 5.1.1.3 genannten Einschränkungen zur Benutzbarkeit, Korrektheit und Skalierbarkeit, zeigte die Evaluation, dass die Implementierung weitere Optimierungen benötigt. Neben Korrekturen am Parser, dessen Auswirkungen für die Sicherheitsanalyse lediglich von geringen Interesse sind, müssen vor allem Anpassungen ergänzt werden, die die Effizienz des Slicing-Prozesses verbessern.

Der Testfall *TF11: OSCI* zeigte Einschränkungen auf bezüglich der Einstellungsmöglichkeiten bei der Ermittlung der Programm-Einstiegspunkte. Zwar werden in der Implementierung verschiedene Algorithmen angeboten, jedoch sind diese in einigen Fällen zu ungenau bzw. nicht genau dokumentiert und getestet. Aus den Erfahrungen aus *TF11* erschließt sich, dass bei größeren Anwendungen sichergestellt werden muss, dass nur ein Entrypoint für den Slice verwendet wird. Mit den bisherigen Implementierungen (und Angabe der Option `mainclass`) ist dies nicht immer gewährleistet.

Zusätzlich ist anzumerken, dass die in der Evaluation getesteten Anwendungen *openkeepass* und *OSCI* lediglich im Verhältnis zu den funktionalen Tests als groß beschrieben sind. Das Verhalten des Auditors bei tatsächlich großen Anwendungen, mit beispielsweise über 200.000 LoC konnte bislang nicht evaluiert werden.

Zuletzt muss zur Fertigstellung des Auditors ebenfalls die Fehlerbehandlung und die Dokumentation innerhalb und außerhalb des Quellcodes verbessert werden.

 © Philip Phu Dang Hoan Nguyen

7. Fazit und Ausblick

7.1. Fazit

Gemäß der gestellten Anforderungen wurde ein Auditor-Werkzeug entwickelt, welches mit Hilfe der Software-Engineering Technik *Programm-Slicing* aus großen Java-Applikationen die sicherheitsrelevanten Programmteile extrahieren kann. Sicherheits-Experten können diese Resultate (genannt Slices) für weiterführende Analysen nutzen und beispielsweise bestimmte Datenflüsse nachvollziehen. Hilfreich ist hierbei die Reduktion der LoC der Applikation auf die relevanten Anweisungen, die über die Daten- und Kontrollabhängigkeiten zu einem interessanten Programmpunkt errechnet werden.

Basierend auf der Bibliothek WALA und Erkenntnissen anderer wissenschaftlicher Arbeiten des TZI, konnte anhand eines agilen inkrementell iterativen Software-Entwicklungsverfahren zur Laufzeit der Evaluation ein voll funktionsfähiges Werkzeug erstellt werden. Neben der Ausgabe syntaktisch korrekter Slices, wurde bei der Evaluation des Auditors ein besonderes Augenmerk auf die Skalierbarkeit gerichtet. Anhand der Evaluation von WALA (inklusive der Parametrisierungsmöglichkeiten) und zweier Testfälle konnte diesbezüglich veranschaulicht werden, welche Auswirkungen bestimmte Einstellungen des Auditors auf das Ergebnis des Slicings haben. Die Ergebnisse der Arbeit verifizieren somit nicht nur die Funktionstüchtigkeit des Auditors, sondern demonstrieren auch die Herangehensweise einer Sicherheitsanalyse. Die somit illustrierten Parametrisierungen können resultierend daraus als effizientes Exempel gesehen werden.

7.2. Ausblick

Da der Auditor einem fortgeschrittenen Prototyp entspricht, ergeben sich über diese Arbeit hinausgehend verschiedene mögliche Tätigkeiten. Wie im Abschnitt 6.4 beschrieben bestehen Einschränkungen des Auditors, die in einem weiteren Entwicklungszyklus gelöst werden müssen, um die Anwendung zu vervollständigen und darauf veröffentlichen zu können.

Bezüglich der Skalierbarkeit muss der Auditor zum einen zusätzlich in der Effizienz optimiert werden (u.a. mit verbesserte Ermittlung der Entrypoints). Zum anderen wäre eine Evaluation auf große Anwendungen, mit beispielsweise über 200.000 LoC diesbezüglich denkbar.

Eine weitere Optimierung könnte sein, dass die rechenintensive Datenflussanalyse außerhalb der eigentlichen Anwendung verhindert wird. Die Testfälle *TF10* und *TF11* zeigen deutlich, dass ein Großteil der Datenflussanalyse innerhalb der eingebundenen Bibliotheken stattfindet und beim Slicing ebenfalls Slice-Ausgaben erzeugt. Da dessen Quellcode nicht rekonstruiert wird, kann auf dessen Datenanalyse vermutlich verzichtet werden und somit Berechnungszeit eingespart werden. Um dieses Verhalten zu erzeugen, wäre es denkbar sogenannte *Stubs* für Klassen und Methoden zu erzeugen und diese statt der echten Bibliotheken einzubinden. Stubs beschreiben leeren nicht implementierten Programmcode.

Generell sollten WALA und ebenso andere Werkzeuge in Zukunft weiter beobachtet werden. Da Projekte wie WALA nicht intensiv vorangetrieben werden, lässt sich vermuten, dass die Bibliothek potenzielle Optimierungen verbirgt. Graf veröffentlichte bspw. im Jahr 2010 ein Paper, in dem er, aufsetzend auf WALA, die SDG-Erzeugung optimierte [Graf 2010]. Dies wurde mit einer eigenständigen Implementierung auf Basis von Objekt-Graphen (anstelle von Objekt-Bäumen) erreicht und sollte ebenfalls weiter evaluiert werden.

A. Verzeichnisse

A.1. Abbildungsverzeichnis

 © Philip Phu Dang Hoan Nguyen

A.2. Literaturverzeichnis

Aho u. a. 2008
Aho, A.V. ; M.S., Lam ; R., Sethi ; D., Ullmann J.: *Compiler: Prinzipien, Techniken und Werkzeuge*. Pearson Education Deutschland, 2008 (Pearson Studium Informatik). https://books.google.de/books?id=pTKAwL64NkoC. – ISBN 9783827370976 3.1, 3.1, 3.1.1, 3.3, 3.1.2, 3.2.2, 3.3, 3.4, 3.4.1, 3.4.5, A.1

Binkley und Gallagher 1996
Binkley, David ; Gallagher, Keith B.: Program slicing. In: *Advances of Computing* 43 (1996), S. 1–50 3.5

Bormann u. a. 2010
Bormann ; Sohr ; Pollem: Informationssicherheit 1, WS 2010/2011, Vorlesungsfolien. 2010 2

Chess und West 2007
Chess, Brian ; West, Jacob: *Secure Programming with Static Analysis*. First. Addison-Wesley Professional, 2007. – 35–46 S. – ISBN 9780321424778 5.1.1.3

Cockburn 2008
Cockburn, Dr. A.: Using Both Incremental and Iterative Development. In: *Humans and Technology Technical Report* (2008) 4.1

Contini 2017
Contini, Scott: TOP 10 DEVELOPER CRYPTO MISTAKES. (2017). https://littlemaninmyhead.wordpress.com/2017/04/22/top-10-developer-crypto-mistakes/. – Aufgerufen 06.11.2017 1

Crypteron 2016
Crypteron: 6 encryption mistakes that lead to data breaches. (2016). https://www.crypteron.com/blog/the-real-problem-with-encryption/ 1

Cytron u. a. 1991
Cytron, Ron ; Ferrante, Jeanne ; Rosen, Barry K. ; Wegman, Mark N. ; Zadeck, F. K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In: *ACM Trans. Program. Lang. Syst.* 13 (1991), Oktober, Nr. 4, 451–490. http://dx.doi.org/10.1145/115372.115320. – DOI 10.1145/115372.115320. – ISSN 0164–0925 3.2.2

datenleck.net 2011
datenleck.net: Sony-Serie - Heute: Sonys 100 Millionen Kunden. (2011). http://www.datenleck.net/?id=275 1

 © Philip Phu Dang Hoan Nguyen

Detmers 2016
Detmers, Mathias: *Evaluation des WALA-Slicers bzgl. der Anwendbarkeit auf sicherheitskritische Java-Programme*, University Bremen, Diplomarbeit, 2016 1, 4.1, 4.2.1, 4.2.2, 5.1.1.3, 5.2.2, 5.2.3, 5.2.4

Dolby und Sridharan 2010
Dolby, Julian ; Sridharan, Manu: Static and Dinamic Program Analysis Using WALA - PLDI 2010 Tutorial, 2010 5.1, 5.2, 5.4, 5.2.3, 5.2.3.2, 5.6, A.1

EETimes 2012
EETimes: How secure is AES against brute force attacks. (2012). `http://www.eetimes.com/documents.asp?doc_id=1279619` 1

Gerken 2015
Gerken, Patrick: *Statische Sicherheitsanalyse von Java Enterprise-Anwendungen mittels Program-Slicing*, University Bremen, Diplomarbeit, 2015 4.1, 4.2.1, 4.2.1, 4.2.2, 5.1.1.1, 5.2.4, 6.2.6

Goltz u. a. 2010
Goltz, Ursula ; Gehrke, Thomas ; Lochau, Malte: *Compilerbau - Vorlesungsskript*. 2010. – nicht veröffentlicht, begleitend zur Vorlesung 3.2, 3.1.2, A.1

Graf 2010
Graf, Jürgen: Speeding up context-, object- and field-sensitive SDG generation. In: *9th IEEE International Working Conference on Source Code Analysis and Manipulation*, 2010, S. 105–114 3.4.6, 7.2

Green und Smith 2016
Green, Matthew ; Smith, Matthew: Developers are Not the Enemy - The Need for Usable Security APIs. In: *IEEE Security & Privacy* 14 (2016), Nr. 5, S. 40–46 1

Griswold 2001
Griswold, William G.: Making Slicing Practical: The Final Mile. In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. New York, NY, USA : ACM, 2001 (PASTE '01). – ISBN 1–58113–413–4, 1– 4.1

Gulmann 2014
Gulmann, Markus: *Statische Sicherheitsanalyse der Android Systemservices*, University Bremen, Diplomarbeit, 2014 4.1, 5.2.2

Harvard 2011
Harvard: *Pointer Analysis*. `https://www.seas.harvard.edu/courses/cs252/`

2011sp/slides/Lec06-PointerAnalysis.pdf. Version: 2011. – nicht veröffentlicht 5.2.2

Hind 2001
Hind, Michael: Pointer Analysis: Haven'T We Solved This Problem Yet? In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering.* New York, NY, USA : ACM, 2001 (PASTE '01). – ISBN 1–58113–413–4, 54–61 5.2.2

Horwitz u. a. 1988
Horwitz, S. ; Reps, T. ; Binkley, D.: Interprocedural Slicing Using Dependence Graphs. In: *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation.* New York, NY, USA : ACM, 1988 (PLDI '88). – ISBN 0–89791–269–1, 35–46 3.5.2

IEEE 2011
IEEE: Systems and software engineering – Life cycle processes –Requirements engineering. In: *ISO/IEC/IEEE 29148:2011(E)* (2011) 5.1.1

Jenkov 2018
Jenkov, Jakob: Tutorial: Java Cipher. (2018). http://tutorials.jenkov.com/java-cryptography/cipher.html 13

Korel und Laski 1988
Korel, B. ; Laski, J.: Dynamic Program Slicing. In: *Inf. Process. Lett.* 29 (1988), Oktober, Nr. 3, 155–163. http://dx.doi.org/10.1016/0020-0190(88)90054-3. – DOI 10.1016/0020–0190(88)90054–3. – ISSN 0020–0190 3.6

Koschke 2014
Koschke, Rainer: *Software-Reengineering - Grundlagen der Softwareanalyse und -transformation.* 2014. – nicht veröffentlicht, begleitend zur Vorlesung 3.1, 3.4, 3.5, 3.2.2, 3.4.5, 3.4.6, 3.5, 3.14, 3.15, 3.5.2, 3.6, A.1

KoSIT 2018
KoSIT: *OSCI.* https://www.xoev.de/detail.php?gsid=bremen83.c.3355.de. Version: 2018. – zuletzt aufgerufen 23.03.2018 6.3.2

Krinke 2003
Krinke, Jens: *Advanced Slicing of Sequential and Concurrent Programs*, Universität Passau, Diss., 2003 3.3, 3.4, 3.4.1, 3.4.2, 3.7, 3.4.3, 3.4.3, 3.4.3, 3.11, 3.4.4, 3.12, 3.5, 3.16, 5.2.1, A.1

Martin 2012
Martin, Keith M.: *Everyday cryptography*. Oxford : : Oxford University Press„ 2012 2, 2.2, 2.3

Matthias Braun u. a. 2013
Matthias Braun ; Sebastian Buchwald ; Sebastian Hack ; Roland Leissa ; Christoph Mallon ; Andreas Zwinkau: *Lecture Notes in Computer Science*. Bd. 7791: *Compiler Construction: 22nd International Conference, CC 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings: Simple and Efficient Construction of Static Single Assignment Form*. Berlin, Heidelberg, 2013. – 102–122 S. http://dx.doi.org/10.1007/978-3-642-37051-9_6. – ISBN 9783642370502 and 978364237051903029743 and 16113349 3.2.2

openkeepass 2018
openkeepass: *openkeepass*. https://github.com/cternes/openkeepass. Version: 2018. – zuletzt aufgerufen 23.03.2018 6.3.1

OSCI Leitstelle 2002a
OSCI Leitstelle: *OSCI-Transport 1.2 - Entwurfsprinzipien, Sicherheitsziele und -mechanismen*. https://www.xoev.de/sixcms/media.php/13/osci_entwurfsprinzipien_1_2.pdf. Version: 2002 6.3, A.1

OSCI Leitstelle 2002b
OSCI Leitstelle: *OSCI-Transport 1.2 - Spezifikation*. https://www.xoev.de/sixcms/media.php/13/osci_spezifikation_1_2_deutsch.pdf. Version: 2002 6.3.2, 6.3.2

Ottenstein und Ottenstein 1984
Ottenstein, Karl J. ; Ottenstein, Linda M.: The Program Dependence Graph in a Software Development Environment. In: *SIGPLAN Not.* 19 (1984), April, Nr. 5, 177–184. http://dx.doi.org/10.1145/390011.808263. – DOI 10.1145/390011.808263. – ISSN 0362–1340 3.5.1

Paar und Pelzl 2009
Paar, Christof ; Pelzl, Jan: *Understanding Cryptography: A Textbook for Students and Practitioners*. 1st. Springer Publishing Company, Incorporated, 2009. – 87–89, 143–144 S. – ISBN 3642041000, 9783642041006 1, 2, 2.1, 2.1, 2.2, 2.3, 2.1.2, 2.2, 2.4, 2.5, 2.2, 2.6, 2.7, A.1

Padberg und Tichy 2007
Padberg, Frank ; Tichy, Walter F.: Empirische Methodik in der Softwaretechnik im Allgemeinen und bei der Software-Visualisierung im Besonderen. In: In-

formatik, Gesellschaft für (Hrsg.): *Software Engineering 2007 - Beiträge zu den Workshops*, 2007 (LNI), S. 211–222 4.1

Reps u. a. 1995
Reps, Thomas ; Horwitz, Susan ; Sagiv, Mooly: Precise Interprocedural Dataflow Analysis via Graph Reachability. In: *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA : ACM, 1995 (POPL '95). – ISBN 0–89791–692–1, 49–61 5.2.1

Robschink 2004
Robschink, Torsten: *Pfadbedingungen in Abhängigkeitsgraphen und ihre Anwendung in der Softwaresicherheitstechnik*, Universität Passau, Diss., 2004 3.3, 3.4.2, 3.4.3, 3.4.3, 3.8, 3.9, 3.4.3, 3.10, A.1

SEAGATE 2008
SEAGATE: Technology Paper - 128-Bit Versus 256-Bit AES Encryption. (2008) 1

SEC Consult 2017
SEC Consult: German E-Government: Details about critical vulnerabilities in core communication library. (2017). `https://sec-consult.com/en/blog/2017/06/german-e-government-details-vulnerabilities/index.html` 6.3.2

Smith u. a. 2017
Smith, Nicholas ; Van Bruggen, Danny ; Tomassetti, Federico: *JavaParser: Visited*. JavaParser.org, 2017 11, 5.8, 5.9, A.1

Sohr u. a. 2015a
Sohr ; Mustafa ; Gulmann ; Gercken: Towards Security Program Comprehension with Design by Contract and Slicing. (2015) 4.2.1

Sohr u. a. 2015b
Sohr ; Mustafa ; Hirch ; Gulmann: Supporting Security Code Audits with Design by Contract and Slicing. (2015) 1

TIOBE 2017
TIOBE: TIOBE Index for October 2017. (2017). `https://www.tiobe.com/tiobe-index/` 1

Van Houtven 2013
Van Houtven, Laurens: *Crypto101*. 2013. – 11–12 S. `http://github.com/crypto101/crypto101.github.io/raw/master/Crypto101.pdf` 1

Veracode 2016
Veracode: State of Software Security 2016. (2016) 1

WALA a
WALA: *WALA Analysis-Scope*. `https://github.com/wala/WALA/wiki/Analysis-Scope`, . – Abgerufen: 20.02.2018 5.2.2

WALA b
WALA: *WALA Intermediate Representation*. `https://github.com/wala/WALA/wiki/Intermediate-Representation-(IR)`, . – Abgerufen: 20.02.2018 5.2.1

WALA c
WALA: *WALA Mapping to source code*. `https://github.com/wala/WALA/wiki/Mapping-to-source-code`, . – Abgerufen: 20.02.2018 5.2.2

WALA d
WALA: *WALA Pointer Analysis*. `https://github.com/wala/WALA/wiki/Pointer-Analysis`, . – Abgerufen: 20.02.2018 5.2.2

WALA e
WALA: *WALA Slicer*. `https://github.com/wala/WALA/wiki/Slicer`, . – Abgerufen: 20.02.2018 5.2.1, 5.2.2, 5.2.3

WALA f
WALA: *WALA SourceForge Documentation*. `https://github.com/wala/WALA/wiki/SourceForge-Documentation`, . – Abgerufen: 20.02.2018 5.2.1

WALA g
WALA: *WALA Technical Overview*. `https://github.com/wala/WALA/wiki/Technical-Overview`, . – Abgerufen: 20.02.2018 5.2.2

Weiser 1981
Weiser, Mark: Program Slicing. In: *Proceedings of the 5th International Conference on Software Engineering, San Diego, California, USA, March 9-12, 1981.*, 1981, 439–449 1

Weiser 1982
Weiser, Mark: Programmers Use Slices when Debugging. In: *Commun. ACM* 25 (1982), Juli, Nr. 7, 446–452. `http://dx.doi.org/10.1145/358557.358577`. – DOI 10.1145/358557.358577. – ISSN 0001–0782 3.5

Weiser 1984
Weiser, Mark: Program Slicing. In: *IEEE Trans. Software Eng.* 10 (1984), Nr. 4, 352–357. `http://dx.doi.org/10.1109/TSE.1984.5010248`. – DOI 10.1109/TSE.1984.5010248 3.5

Wilhelm u. a. 2013
Wilhelm, Reinhard ; Seidl, Helmut ; Hack, Sebastian: *Compiler Design: Analysis and Transformation*. 1st. Springer Publishing Company, Incorporated, 2013. – ISBN 978–3–642–17539–8 , 978–3–642–43591–1 3.1, 3.1.1, 3.1.2

Wu 2005
Wu, Hongjun: *The Misuse of RC4 in Microsoft Word and Excel*. 2005 1

 © Philip Phu Dang Hoan Nguyen

A.3. Akronyme

AES Advanced Encryption Standard. 11

AST Abstract Syntax Tree. 23, 26, 73, 83, 86, 87

CFG Control Flow Graph. 31, 32, 34, 39, 60

CG Call Graph. 36

DES Data Encryption Standard. 10, 11

DHKE Diffie-Hellman Key Exchange. 14

DLP Diskreter Logarithmus-Problem. 15

DSA Digital Signature Algorithm. 14, 15, 19

ECDSA Elliptic Curve Digital Signature Algorithm. 14, 19

EJB Enterprise Java Bean. 45–47, 55

GUI Graphical User Interface. 49, 62

ICFG Interprocedural Control Flow Graph. 35

IR Intermediate Representation. 23, 26

Java EE Java Enterprise Edition. 46

LGPL GNU Lesser General Public License. 73

LoC lines of code. 93

MAC Message Authentication Code. 16–18

MD5 Message-Digest Algorithm 5. 17

NBS US National Bureau of Standards. 11

OTP One-Time Pad. 10

PDG Program Dependence Graph. 32, 37, 39, 40, 59

PKI Public Key Infrastructure. 20

PRNG Pseudorandom Number Generator. 10

RSA Rivest-Shamir-Adleman Algorithm. 14, 15, 19

SDG System Dependence Graph. 37, 40, 59–61, 66, 68–70, 72, 92

SHA Secure Hash Algorithm. 17

SSA Static Single Assignment Form. 23, 27–29, 55, 60, 79–81

TOC Three Address Code. 23, 26

TRNG True Random Number Generator. 10

TZI Technologie-Zentrum Informatik und Informationstechnik. 42

WALA T.J.Watson Libraries for Analysis. 59, 73

 © Philip Phu Dang Hoan Nguyen

B. Anhang B

B.1. Beispiel-Konfigurationsdatei

```
# Slicer configuration

jar = multipleinsttest.jar

mainclass = LMultipleInstancesTest

src_caller = main
src_callee = println

multiple_caller = false
src_callee_class = ClassB

# Data dependence options:
# full, no_base_ptrs, no_base_no_heap, no_heap, none, reflection,
# no_base_no_exceptions, no_base_no_heap_no_exceptions, no_exceptions, no_heap_no_exceptions
datadep_opts = no_exceptions

# Control dependence options:
# full, none or no_exceptional_edges
controldep_opts = no_exceptional_edges


# Settings for CG PDF Printing
pdfview_exe = /Applications/Preview.app/Contents/MacOS/Preview
dot_exe = /usr/local/bin/dot

create_pdfs = false
sdg_pdf = false

output_dir = multinstances_output

# Define, if reconstructed code is split in original java files or merged into one file
# default = false
split_java_output = true

# Advanced mode traces object instantiation and all calls on that object
advanced_mode = true

# Add object instantiation methods like "getInstance()"
object_inst_methods =
```

Listing B.1: Programmauszug vom Parser (visit() Teil 2)

B.2. Exclusionsfile

```
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.*
```


© Philip Phu Dang Hoan Nguyen

```
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.*
```

```
111 javax.sql.rowset.serial.*
112 javax.sql.rowset.spi.*
113 javax.swing.*
114 javax.swing.border.*
115 javax.swing.colorchooser.*
116 javax.swing.event.*
117 javax.swing.filechooser.*
118 javax.swing.plaf.*
119 javax.swing.plaf.basic.*
120 javax.swing.plaf.metal.*
121 javax.swing.plaf.multi.*
122 javax.swing.plaf.nimbus.*
123 javax.swing.plaf.synth.*
124 javax.swing.table.*
125 javax.swing.text.*
126 javax.swing.text.html.*
127 javax.swing.text.html.parser.*
128 javax.swing.text.rtf.*
129 javax.swing.tree.*
130 javax.swing.undo.*
131 javax.tools.*
132 javax.transaction.*
133 javax.transaction.xa.*
134 javax.xml.*
135 javax.xml.bind.*
136 javax.xml.bind.annotation.*
137 javax.xml.bind.annotation.adapters.*
138 javax.xml.bind.attachment.*
139 javax.xml.bind.helpers.*
140 javax.xml.bind.util.*
141 javax.xml.crypto.*
142 javax.xml.crypto.dom.*
143 javax.xml.crypto.dsig.*
144 javax.xml.crypto.dsig.dom.*
145 javax.xml.crypto.dsig.keyinfo.*
146 javax.xml.crypto.dsig.spec.*
147 javax.xml.datatype.*
148 javax.xml.namespace.*
149 javax.xml.parsers.*
150 javax.xml.soap.*
151 javax.xml.stream.*
152 javax.xml.stream.events.*
153 javax.xml.stream.util.*
154 javax.xml.transform.*
155 javax.xml.transform.dom.*
156 javax.xml.transform.sax.*
157 javax.xml.transform.stax.*
158 javax.xml.transform.stream.*
159 javax.xml.validation.*
160 javax.xml.ws.*
161 javax.xml.ws.handler.*
162 javax.xml.ws.handler.soap.*
163 javax.xml.ws.http.*
164 javax.xml.ws.soap.*
165 javax.xml.ws.spi.*
166 javax.xml.ws.spi.http.*
167 javax.xml.ws.wsaddressing.*
```

 © Philip Phu Dang Hoan Nguyen

```
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
```

 © Philip Phu Dang Hoan Nguyen

```
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.CipherInputStream
javax.crypto.CipherOutputStream
javax.crypto.CipherSpi
javax.crypto.EncryptedPrivateKeyInfo
javax.crypto.ExemptionMechanism
javax.crypto.ExemptionMechanismSpi
javax.crypto.KeyAgreement
javax.crypto.KeyAgreementSpi
javax.crypto.KeyGenerator
javax.crypto.KeyGeneratorSpi
javax.crypto.Mac
javax.crypto.MacSpi
javax.crypto.NullCipher
javax.crypto.SealedObject
javax.crypto.SecretKeyFactory
javax.crypto.SecretKeyFactoryjavax.crypto.
```

```
javax.crypto.AEADBadTagException
javax.crypto.ExemptionMechanismException
javax.crypto.ShortBufferException


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\/.*

//java.security\/.*
//javax.crypto\/.*
```

Listing B.2: Exclusionsfile

B.3. Quellcode der Testfälle

B.3.1. TF01: Arithmetisches Slicing

```
1  public class ArithmeticTest {
2
3    public static void main(String[] argsv){
4      int value = 1;
5      int value2 = 2;
6      value += value;
7      int someOtherValue = 2;
8      value2 += someOtherValue;
9      someOtherValue -= someOtherValue;
10     value = value--;
11     while(someOtherValue < 2){
12       someOtherValue++;
13     }
14     while(value < 2){
15       value*= 2;
16     }
17     while(value2 < 10){
18       value2*= 2;
19     }
20     System.out.println(value);
21     System.out.println(value2);
22   }
23 }
```

Listing B.3: Testfall Arithmetisches Slicing - Vollständiger Quellcode

```
1  Calling Slicer with:
2  ArithmeticTest.jar LArithmeticTest main println NO_BASE_NO_HEAP NO_EXCEPTIONAL_EDGES
3  Class Hierarchy has 4965 classes
4
5  == GET ENTRY POINTS ==
6
7  Got Entrypoints using method: getEntryPoints() (Detmers)
8
9  == CALL GRAPH ==
10
11 Saved CG to call_craph.txt
12
13 == FIND CALLER/CALLEE ==
14
15 Found Caller main 1 time(s).
16 Found Callee println
17 Found Callee println
18
19 == POINTER ANALYSIS ==
20
21 Saved PA to pointeranalysis.txt
```

```
22
23 == SLICING ==
24
25 Computing backward slice...
26
27 .
28
29 .
30
31 Saved raw slice to arithm_output/slice_raw.txt
32
33 {ArithmeticTest.java=[17, 18, 20, 21, 6, 8, 9, 11, 12, 14, 15]}
34 Number of Statements: 20
35 Callgraph & Slice buildtime: {0=300 ms, 1=134 ms}
36 10 callgraph nodes
37 20 statements in slice
38
39 All line numbers:
40 [17, 18, 20, 21, 6, 8, 9, 11, 12, 14, 15]
41
42 All slice line numbers:
43 {ArithmeticTest.java=[17, 18, 20, 21, 6, 8, 9, 11, 12, 14, 15]}
44
45 == RECONSTRUCTING CODE ==
46
47 KEYVAL: ArithmeticTest.java[17, 18, 20, 21, 6, 8, 9, 11, 12, 14, 15]
48 Slice line numbers (sorted) for file ArithmeticTest.java: [6, 8, 9, 11, 12, 14, 15, 17, 18,
     20, 21]
49 Line numbers with Begin & EndLine: [1, 2, 3, 6, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
      21, 22, 23]
50 Line numbers with Begin & EndLine(sorted): [1, 2, 3, 6, 8, 9, 11, 12, 13, 14, 15, 16, 17, 18,
      19, 20, 21, 22, 23]
51 Saved reconstructed java to arithm_output//reconstructed_java/output_slice.java
52
53 Full Slicing time: 2426 ms
54
55 ============
56 Slicing done.
```

Listing B.4: Testfall Arithmetisches Slicing - Ausgaben des Auditors

B.3.2. TF02: Mehrfache Callee-Anweisungen

```
1 public class ClassA {
2
3   public String some_method() {
4     return "this is Class A\n";
5   }
6 }
7
8 public class ClassB {
9
```

 © Philip Phu Dang Hoan Nguyen

```
10   public String some_method() {
11     return "this is Class B\n";
12   }
13 }
14
15 public class ClassC {
16
17   public String some_method() {
18     return "this is Class C\n";
19   }
20 }
21
22
23 public class MultipleCalleeTest {
24
25   public static void main(String[] argsv){
26     String result = "Start\n";
27     ClassA object_a = new ClassA();
28     ClassB object_b = new ClassB();
29     ClassC object_c = new ClassC();
30
31     result += "Call some_method()\n";
32
33     String text_a = object_a.some_method();
34     String text_b = object_b.some_method();
35     String text_c = object_c.some_method();
36
37     System.out.println(result);
38     System.out.println(text_a + text_b + text_c);
39     System.out.println("test done");
40   }
41 }
```

Listing B.5: Testfall Mehrfache Callee-Anweisungen - Vollständiger Quellcode

B.3.3. TF03: Mehrfache Caller-Anweisungen

```
1  public class ClassA {
2
3    public String some_method() {
4      return "this is Class A\n";
5    }
6  }
7
8  public class MultipleCallerTest {
9
10   public static String entry_method(String value) {
11     ClassA obj = new ClassA();
12     String result = value + obj.some_method();
13     return result;
14   }
15
```

```
16   public static String entry_method(String value, String value2) {
17     ClassA obj = new ClassA();
18     String result = value + obj.some_method();
19     result += "\n" + value2;
20     return result;
21   }
22
23   public static void main(String[] argsv){
24     String value = "Start Test:\n";
25     String result = entry_method(value);
26
27     String value2 = "Same method other parameters:\n";
28     result += entry_method(value,value2);
29
30     System.out.println("test done");
31     System.out.println(result);
32   }
33 }
```

Listing B.6: Testfall Mehrfache Caller-Anweisungen - Vollständiger Quellcode

B.3.4. TF04: Exception-Kontrollfluss

```
1  public class NoExcEdgesTest {
2
3    public static Integer important(Integer num) {
4      Integer result = num + 123;
5      return result;
6    }
7
8    public static float divide(int a, int b) throws Exception{
9      float result = 0;
10     try{
11       result = a / b;
12     }catch(Exception e1){
13       System.out.println("ERROR");
14     }
15     return result;
16   }
17
18   public static Integer entry_method(String value) throws Exception {
19     Integer wr_int = 0;
20     wr_int += 1;
21     wr_int += 2;
22     int pr_int = 0;
23     pr_int += 1;
24
25     Float wr_float = (float) 0.1;
26     wr_float = wr_float +1;
27     wr_float = wr_float +2;
28     float pr_float = (float) 0.1;
29     pr_float = pr_float +1;
```


© Philip Phu Dang Hoan Nguyen

```
30
31    float other_float = divide(10,2);
32
33    Integer result = important(10);
34    return result;
35  }
36
37
38  public static void main(String[] argsv) throws Exception{
39    String value = "Start Test:\n";
40    Integer result = entry_method(value);
41
42    System.out.println("test done");
43    System.out.println(result);
44  }
45 }
```

Listing B.7: Testfall Exception-Kontrollfluss - Vollständiger Quellcode

B.3.5. TF05: Objektverfolgung

```
1  public class SimpleTestObject {
2
3     Integer value = 0;
4     String word = "start";
5
6     public void do_stuff(){
7       //int a = 1;
8       //a++;
9       word = "new word";
10    }
11
12    public static Integer calc_param(){
13      return 5;
14    }
15
16    public String important_method(Integer add, Integer val){
17      value += add;
18      String summary = word + " and " + value + " and " + val;
19      return summary;
20    }
21 }
```

Listing B.8: Testfall Objektverfolgung - Vollständiger Quellcode SimpleTestObject.java

```
1  public class SimpleTest {
2    Integer value;
3    String word;
```

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

Listing B.9: Testfall Objektverfolgung - Vollständiger Quellcode SimpleTest.java

B.3.6. TF06/TF07: Clinic (Verschlüsselung und Signierung)

```
1  package clinic;
2
3  import java.io.BufferedInputStream;
4  import java.io.FileInputStream;
5  import java.io.IOException;
6  import java.net.URL;
7  import java.security.GeneralSecurityException;
8  import java.security.InvalidKeyException;
9  import java.security.Key;
10 import java.security.KeyStore;
11 import java.security.KeyStoreException;
12 import java.security.NoSuchAlgorithmException;
13 import java.security.NoSuchProviderException;
14 import java.security.PrivateKey;
15 import java.security.Signature;
16 import java.security.SignatureException;
17 import java.security.UnrecoverableKeyException;
18 import java.security.cert.CertificateException;
19
20 import javax.annotation.Resource;
21 import javax.crypto.BadPaddingException;
22 import javax.crypto.Cipher;
23 import javax.crypto.IllegalBlockSizeException;
24 import javax.crypto.NoSuchPaddingException;
25 import javax.ejb.EJBContext;
26 import javax.ejb.Stateful;
27 import javax.net.ssl.HttpsURLConnection;
28 import javax.net.ssl.SSLContext;
```

 © Philip Phu Dang Hoan Nguyen

```
29 import javax.net.ssl.TrustManagerFactory;
30
31 import sun.net.www.protocol.https.DefaultHostnameVerifier;
32
33 @Stateful
34 public class ClinicExampleBean implements ClinicExample {
35
36   @Resource
37   EJBContext ctx;
38
39   Cipher mCipher;
40   Signature mSignature;
41
42   StringBuffer sb = new StringBuffer();
43
44   public String readPrescriptions(String ehrID, String userID) {
45     Clinician clinician = getClinician(userID);
46     EHR ehr = getEHR(ehrID);
47     if (!(ctx.isCallerInRole("Physician") || ctx.isCallerInRole("Nurse")) &&
48         ehr.getWard().equals(clinician.getWard())) {
49       throw new SecurityException("No sufficient access rights.");
50     }
51     return ehr.getPrescriptions();
52   }
53
54   private EHR getEHR(String ehrID) {
55     return new EHR(ehrID);
56   }
57
58   private Clinician getClinician(String userID) {
59     return new Clinician(userID);
60   }
61
62   @Override
63   public void writePrescriptions(String ehrID, String userID, String pres) {
64     Clinician clinician = getClinician(userID);
65     EHR ehr = getEHR(ehrID);
66     if(!(ctx.isCallerInRole(getRole()) || ctx.isCallerInRole("Nurse")) &&
67         ehr.getWard().equals(clinician.getWard())) {
68       throw new SecurityException("No sufficient access rights.");
69     }
70     ehr.setPrescriptions(pres);
71   }
72
73   private String getRole() {
74     return "Physician";
75   }
76
77   @Override
78   public String readPatientData(String userID, String ehrID) {
79     Clinician clinician = getClinician(userID);
80     EHR ehr = getEHR(ehrID);
81     if (!ctx.isCallerInRole("Physician") && ehr.getWard().equals(clinician.getWard())) {
82       throw new SecurityException("No sufficient access rights.");
83     }
84     String patientData = ehr.getAdministrativeData() + ehr.getPrescriptions() + ehr.
         getDiagnosis();
```

```
85    return patientData;
86  }
87
88  @Override
89  public void sendDiagnosis(String userID, String ehrID, Key key) throws InvalidKeyException,
        NoSuchAlgorithmException, NoSuchPaddingException, IllegalBlockSizeException,
        BadPaddingException, NoSuchProviderException, UnrecoverableKeyException,
        KeyStoreException, CertificateException, SignatureException, IOException {
90    Clinician clinician = getClinician(userID);
91    EHR ehr = getEHR(ehrID);
92    if(!ctx.isCallerInRole("Physician") && ehr.getWard().equals(clinician.getWard())) {
93      throw new SecurityException("No sufficient access rights.");
94    }
95    byte[] signedData = signData(ehr.getDiagnosis().getBytes(), clinician.getKeyName());
96    byte[] encryptedSignedDiagnosis = encryptData(signedData, key);
97    String mailAddress = getMailAddress(userID);
98    sendPatientDataToMailAddress(encryptedSignedDiagnosis, mailAddress);
99  }
100
101 private void sendPatientDataToMailAddress(byte[] encryptedSignedDiagnosis,
102     String mailAddress) {
103   // TODO Auto-generated method stub
104
105 }
106
107 private String getMailAddress(String userID) {
108   // TODO Auto-generated method stub
109   return null;
110 }
111
112 @Override
113 public byte[] signData(byte[] data, String keyAlias) throws NoSuchAlgorithmException,
        NoSuchProviderException, KeyStoreException, CertificateException, IOException,
        UnrecoverableKeyException, InvalidKeyException, SignatureException {
114   char[] spass = getSignPass();
115   char[] kpass = getKeyPass();
116   String ksName = getKsName();
117
118   mSignature = Signature.getInstance("SHA256withDSA", "SUN") ;
119   KeyStore ks = KeyStore.getInstance("Hospital");
120   FileInputStream ksfis = new FileInputStream(ksName);
121   BufferedInputStream ksbufin = new BufferedInputStream(ksfis);
122   ks.load(ksbufin, spass) ;
123   PrivateKey priv = (PrivateKey) ks.getKey(keyAlias,kpass) ;
124   System.out.println("output");
125   mSignature.initSign(priv);
126   mSignature.update(data);
127   return mSignature.sign();
128 }
129
130 private char[] getKeyPass() {
131   return null;
132 }
133
134 private String getKsName() {
135   // TODO Auto-generated method stub
136   return null;
```


© Philip Phu Dang Hoan Nguyen

```
137   }
138
139   private char[] getSignPass() {
140     // TODO Auto-generated method stub
141     return null;
142   }
143
144   @Override
145   public byte[] encryptData(byte[] data, Key key) throws NoSuchAlgorithmException,
146     NoSuchPaddingException, InvalidKeyException, IllegalBlockSizeException,
          BadPaddingException {
147     Integer a;
148     mCipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
149     a=2;
150     mCipher.init(Cipher.ENCRYPT_MODE, key);
151     a--;
152     return mCipher.doFinal(data);
153   }
154
155   @Override
156   public void openSSLConnection(URL url) throws IOException,
157       GeneralSecurityException {
158     TrustManagerFactory tmf = TrustManagerFactory.getInstance("X509");
159     tmf.init(getKeyStore());
160     SSLContext context = SSLContext.getInstance("TLS");
161     context.init(null, tmf.getTrustManagers(), null);
162     HttpsURLConnection urlConnection = (HttpsURLConnection) url.openConnection();
163     urlConnection.setSSLSocketFactory(context.getSocketFactory());
164     urlConnection.setHostnameVerifier(new DefaultHostnameVerifier());
165     urlConnection.connect();
166   }
167
168   private KeyStore getKeyStore() {
169     // TODO Auto-generated method stub
170     return null;
171   }
172 }
```

Listing B.10: Testfall Clinic - Vollständiger Quellcode

```
1  Calling Slicer with:
2  clinic/isCAllerInRole.jar Lclinic/ClinicExampleBean encryptData doFinal FULL FULL
3  Couldn't create output folder 'clinic_dofinal_output/'
4  Probably no rights or already existing.
5  Couldn't create output folder 'clinic_dofinal_output//reconstructed_java'
6  Probably no rights or already existing.
7  Class Hierarchy has 4972 classes
8
9  == GET ENTRY POINTS ==
10
11 Got Entrypoints using method: getEntryPoints() (Detmers)
12
13 == CALL GRAPH ==
14
```

```
Saved CG to call_craph.txt

== FIND CALLER/CALLEE ==

Found Caller encryptData 1 time(s).
Found Callee doFinal

== ADVANCED SLICING with object tracing ==

new object_no: 7 (getInstance)
new var_name: mCipher (put)
new object_no: 11 (get on mCipher)
Found method call: init (on object <Application,Ljavax/crypto/Cipher>)
new object_no: 19 (get on mCipher)

== END ADVANCED SLICING ==


== POINTER ANALYSIS ==

Saved PA to pointeranalysis.txt

== SLICING ==

Computing backward slice...

.

.

Saved raw slice to clinic_dofinal_output/slice_raw.txt

# Edges: 67910
# Nodes: 45841

Number of Statements: 34440
Callgraph & Slice buildtime: {0=858 ms, 1=51243 ms}
297 callgraph nodes
34440 statements in slice

All line numbers:
[58, 64, 65, 66, 131, 67, 68, 136, ..., 306, 308, 310, 312, 57, 314, 316, 319, 70, 60]

All slice line numbers:
{KeyStoreException.java=[58], ClinicExampleBean.java=[64, 65, 66, ...], ...,
    NullPointerException.java=[70, 60]}

== RECONSTRUCTING CODE ==

KEYVAL: KeyStoreException.java[58]
Slice line numbers (sorted) for file KeyStoreException.java: [58]
KeyStoreException.java file not found! Skipping!

KEYVAL: ClinicExampleBean.java[64, 65, 66, 131, 67, 68, 136, 74, 141, 79, 80, 81, 82, 148,
    149, 150, 151, 152, 90, 91, 92, 93, 95, 96, 45, 46, 47, 48, 49, 114, 115, 116, 118, 55,
    119, 120, 121, 122, 59, 123, 124, 125, 126, 127]
```


© Philip Phu Dang Hoan Nguyen

```
Found file at: /Users/philipnguyen/Dropbox/Masterarbeit/Evaluation/clinic/ClinicExampleBean.
    java
Slice line numbers (sorted) for file ClinicExampleBean.java: [45, 46, 47, 48, 49, 55, 59, 64,
     65, 66, 67, 68, 74, 79, 80, 81, 82, 90, 91, 92, 93, 95, 96, 114, 115, 116, 118, 119,
    120, 121, 122, 123, 124, 125, 126, 127, 131, 136, 141, 148, 149, 150, 151, 152]
Line numbers with Begin & EndLine: [128, 130, 131, 132, 134, 136, 137, 139, 141, 142, 144,
    145, 146, 148, 149, 150, 151, 152, 153, 34, 172, 44, 45, 46, 47, 48, 49, 50, 52, 54, 55,
    56, 58, 59, 60, 62, 63, 64, 65, 66, 67, 68, 69, 71, 73, 74, 75, 77, 78, 79, 80, 81, 82,
    83, 86, 88, 89, 90, 91, 92, 93, 94, 95, 96, 99, 112, 113, 114, 115, 116, 118, 119, 120,
    121, 122, 123, 124, 125, 126, 127]
Line numbers with Begin & EndLine(sorted): [34, 44, 45, 46, 47, 48, 49, 50, 52, 54, 55, 56,
    58, 59, 60, 62, 63, 64, 65, 66, 67, 68, 69, 71, 73, 74, 75, 77, 78, 79, 80, 81, 82, 83,
    86, 88, 89, 90, 91, 92, 93, 94, 95, 96, 99, 112, 113, 114, 115, 116, 118, 119, 120, 121,
    122, 123, 124, 125, 126, 127, 128, 130, 131, 132, 134, 136, 137, 139, 141, 142, 144, 145,
     146, 148, 149, 150, 151, 152, 153, 172]
KEYVAL: IOException.java[58]
Slice line numbers (sorted) for file IOException.java: [58]
IOException.java file not found! Skipping!

KEYVAL: NoSuchProviderException.java[56]
Slice line numbers (sorted) for file NoSuchProviderException.java: [56]
NoSuchProviderException.java file not found! Skipping!

...

Saved reconstructed java to clinic_dofinal_output//reconstructed_java/output_slice.java

Full Slicing time: 59409 ms

============
Slicing done.
```

Listing B.11: Testfall Clinic (Verschlüsselung) - Gekürzte Ausgaben des Auditors

B.3.7. TF08: Coding-Konventionen

```
public class CodingConventionTest
{
  // entry_method does something
  public static Integer entry_method( int value,
                  int value2,
                  int value3)
                  throws Exception
  {
    int duration = 10;
    int result = 0;

    for (int i = 0; i<=duration; duration++)
    {
      if (i==0)
      {
        // comment
```

```
17        result += value;
18      }
19      else
20      {
21        if (i < 5)
22        result = value2;
23        else
24        System.out.println("important");
25      }
26    }
27    String unused = "test";
28
29    return result;
30  }
31
32
33  public static void main(String[] argsv) throws Exception
34  {
35    String value = "Start Test:\n";
36    Integer result = entry_method(1,2,3);
37  }
38 }
```

Listing B.12: Testfall Coding-Konventionen - Vollständiger Quellcode

B.3.8. TF09: Tief-verschachtelter Callee

```
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);
```


© Philip Phu Dang Hoan Nguyen

```
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}
```

Listing B.13: Testfall Tief-verschachtelter Callee - Vollständiger Quellcode ClinicExampleBean.java

B.3.9. TF10 OpenKeePass

```
1 public class Aes {
2
3   private static final String MSG_KEY_MUST_NOT_BE_NULL = "Key must not be null";
4   private static final String MSG_IV_MUST_NOT_BE_NULL = "IV must not be null";
5   private static final String MSG_DATA_MUST_NOT_BE_NULL = "Data must not be null";
6   private static final String KEY_TRANSFORMATION = "AES/ECB/NoPadding";
7   private static final String DATA_TRANSFORMATION = "AES/CBC/PKCS5Padding";
8   private static final String KEY_ALGORITHM = "AES";
9
10  private Aes() {
11  }
12
13  ...
14
15  public static byte[] decrypt(byte[] key, byte[] ivRaw, byte[] data) {
16    if (key == null) {
17      throw new IllegalArgumentException(MSG_KEY_MUST_NOT_BE_NULL);
18    }
19    if (ivRaw == null) {
20      throw new IllegalArgumentException(MSG_IV_MUST_NOT_BE_NULL);
21    }
22    if (data == null) {
23      throw new IllegalArgumentException(MSG_DATA_MUST_NOT_BE_NULL);
24    }
25
26    return transformData(key, ivRaw, data, Cipher.DECRYPT_MODE);
27  }
```

```
28
29  public static byte[] encrypt(byte[] key, byte[] ivRaw, byte[] data) {
30    if (key == null) {
31      throw new IllegalArgumentException(MSG_KEY_MUST_NOT_BE_NULL);
32    }
33    if (ivRaw == null) {
34      throw new IllegalArgumentException(MSG_IV_MUST_NOT_BE_NULL);
35    }
36    if (data == null) {
37      throw new IllegalArgumentException(MSG_DATA_MUST_NOT_BE_NULL);
38    }
39
40    return transformData(key, ivRaw, data, Cipher.ENCRYPT_MODE);
41  }
42
43  private static byte[] transformData(byte[] key, byte[] ivRaw, byte[] encryptedData, int
       operationMode) {
44    try {
45      Cipher cipher = Cipher.getInstance(DATA_TRANSFORMATION);
46      Key aesKey = new SecretKeySpec(key, KEY_ALGORITHM);
47      IvParameterSpec iv = new IvParameterSpec(ivRaw);
48      cipher.init(operationMode, aesKey, iv);
49      return cipher.doFinal(encryptedData);
50    } catch (NoSuchAlgorithmException e) {
51      throw new UnsupportedOperationException("The specified algorithm is unknown", e);
52    } catch (NoSuchPaddingException e) {
53      throw new UnsupportedOperationException("The specified padding is unknown", e);
54    } catch (InvalidKeyException e) {
55      throw createCryptoException(e);
56    } catch (InvalidAlgorithmParameterException e) {
57      throw createCryptoException(e);
58    } catch (IllegalBlockSizeException e) {
59      throw createCryptoException(e);
60    } catch (BadPaddingException e) {
61      throw createCryptoException(e);
62    }
63  }
```

Listing B.14: Testfall OpenKeePass - Auszug Quellcode (Aes.java)

B.3.10. TF11 OSCI

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


© Philip Phu Dang Hoan Nguyen

```
9   }
10  public ResponseToForwardDelivery sendForwardDelivery(String urlRecipient)
11                          throws GeneralSecurityException, ...
12  {
13    Originator user_1 = new Originator(new de.osci.osci12.samples.impl.crypto.PKCS12Signer("/
          de/osci/osci12/samples/zertifikate/test_alice_signature.p12",
14    DialogHandler clientDialog = new DialogHandler(user_1, intermed, new de.osci.osci12.
          samples.impl.HttpTransport());
15    GetMessageId getMsgID = new GetMessageId(clientDialog);
16    ResponseToGetMessageId rsp2GetMsgID = getMsgID.send();
17    Addressee user_2 = new Addressee(null,
18                de.osci.helper.Tools.createCertificate(getClass().getResourceAsStream("/de/
                      osci/osci12/samples/zertifikate/test_bob_cypher.cer")));
19    ForwardDelivery forwardDel = new ForwardDelivery(clientDialog, user_2, urlRecipient,
          rsp2GetMsgID.getMessageId());
20    forwardDel.setSubject("Subject");
21    forwardDel.setQualityOfTimeStampCreation(false);
22    forwardDel.setQualityOfTimeStampReception(false);
23    ContentContainer data = new ContentContainer();
24    data.addContent(new Content("Any content data."));
25    data.addContent(new Content(new Attachment(new ByteArrayInputStream("Any attached data".
          getBytes()), "attachment_id")));
26    forwardDel.addContentContainer(data);
27    ResponseToForwardDelivery rsp2FwdDel = forwardDel.send();
28  }
29  public static void main(String[] args)
30  {
31    try
32    {
33      OneWayMessage_PassiveRecipient scenario_2 = new OneWayMessage_PassiveRecipient(args[0]);
34      ResponseToForwardDelivery responseForward = scenario_2.sendForwardDelivery(args[1]);
35    }
36  }
37}
```

Listing B.15: TF11: OSCI - Slice NO_EXC NO_EXC_E OneWayMessage_PassiveRecipient.java

```
1  public class Crypto
2  {
3    public static String toHex(byte[] b)
4    {
5      for ( int i = 0 ; i < b.length ; i++ )
6      {
7        j = (b[i] >> 4) & 0xf;
8        j = b[i] & 0xf;
9      }
10   }
11   public static byte[] doRSADecryption(Key key, byte[] data)
12     throws OSCICipherException, NoSuchAlgorithmException
13   {
14     return doRSADecryption(key, data, Constants.ASYMMETRIC_CIPHER_ALGORITHM_RSA_1_5, null,
           null, null);
15   }
```

```
16  public static byte[] doRSADecryption(Key key,
17                               byte[] data,
18                               String algorithm,
19                               String mgfAlgorithm,
20                               String digestAlgorithm,
21                               byte[] oaepParams)
22  throws OSCICipherException, NoSuchAlgorithmException
23  {
24    try
25    {
26      if (DialogHandler.getSecurityProvider() == null)
27        cipher = javax.crypto.Cipher.getInstance(Constants.JCA_JCE_MAP.get(algorithm));
28      else
29        cipher = javax.crypto.Cipher.getInstance(Constants.JCA_JCE_MAP.get(algorithm),
30                                    DialogHandler.getSecurityProvider());
31      if (algorithm.equals(Constants.ASYMMETRIC_CIPHER_ALGORITHM_RSA_OAEP))
32      {
33        if (mgfAlgorithm.equals(Constants.MASK_GENERATION_FUNCTION_1_SHA256))
34        {
35        }
36        else if (mgfAlgorithm.equals(Constants.MASK_GENERATION_FUNCTION_1_SHA512))
37        {
38        }
39        else
40        {
41          throw new IllegalArgumentException("Unsupported mask generation function " +
                mgfAlgorithm);
42        }
43        OAEPParameterSpec oaepParameters = new OAEPParameterSpec(Constants.JCA_JCE_MAP.get(
              digestAlgorithm),
44        cipher.init(Cipher.DECRYPT_MODE, key, oaepParameters);
45      }
46      else
47      {
48        cipher.init(Cipher.DECRYPT_MODE, key);
49      }
50      return cipher.doFinal(data);
51    }
52    catch (Exception ex)
53    {
54      throw new OSCICipherException("decryption_error");
55    }
56  }
57}
```

Listing B.16: TF11: OSCI - Slice FULL NO_EXC_E Crypto.java (mit Angabe mainclass)

```
1 public abstract class OSCIRequest extends OSCIMessage
2 {
3   // private static Log log = LogFactory.getLog(OSCIRequest.class);
4   protected OSCIMessage transmit(OutputStream outp, OutputStream inp)
5                          throws IOException,
6                          OSCIException,
```


© Philip Phu Dang Hoan Nguyen

```
7                           NoSuchAlgorithmException
8   {
9     transport = dialogHandler.getTransportModule().newInstance();
10    boolean intermed = ((this instanceof AcceptDelivery) || (this instanceof ProcessDelivery))
          ;
11    if (!intermed)
12      uri = ((de.osci.osci12.roles.Intermed) dialogHandler.getSupplier()).getUri();
13    else
14      uri = uriReceiver;
15    if (dialogHandler.isCreateSignatures())
16      sign();
17    try
18    {
19      if (dialogHandler.isEncryption())
20      {
21        SOAPMessageEncrypted sme = new SOAPMessageEncrypted(this, outp);
22        out = transport.getConnection(uri, sme.calcLength());
23        sme.writeXML(out);
24      }
25      else
26      {
27        out = transport.getConnection(uri, calcLength());
28        if (outp != null)
29        {
30          StoreOutputStream sos = new StoreOutputStream(out, outp);
31          writeXML(sos);
32          sos.close();
33        }
34        else
35          writeXML(out);
36      }
37    }
38    try
39    {
40      in = transport.getResponseStream();
41      rsp = parser.parseStream(in, dialogHandler, false, inp);
42    }
43    return rsp;
44  }
45  void sign()
46             throws IOException,
47                    OSCIException,
48                    de.osci.osci12.common.OSCICancelledException,
49                    java.security.NoSuchAlgorithmException
50  {
51    super.sign(dialogHandler.getClient());
52  }
53  protected void compose() throws OSCIException,
54                           NoSuchAlgorithmException,
55                           IOException
56  {
57    super.compose();
58  }
59 }
```

Listing B.17: TF11: OSCI - Slice NH_NX NO_EXC_E OSCIRequest.java

```
1  class SOAPMessageEncrypted extends OSCIMessage
2  {
3
4    @Override
5    protected long calcLength() throws IOException, OSCIException, NoSuchAlgorithmException
6    {
7      if ((stateOfMsg & STATE_COMPOSED) == 0)
8      compose();
9      len += length;
10     len += Integer.toString(length).getBytes(Constants.CHAR_ENCODING).length;
11     len += (4 * msg.boundary_string.getBytes(Constants.CHAR_ENCODING).length);
12     len += contentID.getBytes(Constants.CHAR_ENCODING).length;
13     try
14     {
15       len += Base64.calcB64Length(cipherCert.getEncoded().length);
16     }
17     if (msg.base64)
18     {
19       len += 6;
20       len += Base64.calcB64Length(Crypto.calcSymEncLength(msg.calcLength(),
             symmetricCipherAlgorithm));
21     }
22     else
23     {
24       len += 5;
25       len += Crypto.calcSymEncLength(msg.calcLength(), symmetricCipherAlgorithm);
26     }
27     return len;
28   }
29   @Override
30   protected void compose() throws OSCIException, NoSuchAlgorithmException, IOException
31   {
32     if ((msg.stateOfMsg & STATE_COMPOSED) == 0)
33     msg.compose();
34     encSymKey = Base64.encode(Crypto.doRSAEncryption(cipherCert,
35     msg.getDialogHandler().getAsymmetricCipherAlgorithm()))
36   }
37   @Override
38   protected void writeXML(OutputStream out)
39   throws IOException, OSCIException, NoSuchAlgorithmException
40   {
41     if ((stateOfMsg & STATE_COMPOSED) == 0)
42     compose();
43     try
44     {
45       out.write(Base64.encode(cipherCert.getEncoded()).getBytes(Constants.CHAR_ENCODING));
46     }
47     if (msg.base64)
48     {
49       tdesOut = new SymCipherOutputStream(b64out, symKey, symmetricCipherAlgorithm, true);
50     }
51     else
52     tdesOut = new SymCipherOutputStream(out, symKey, symmetricCipherAlgorithm, true);
53     if (storeStream == null)
54     {
55       msg.writeXML(tdesOut);
```


© Philip Phu Dang Hoan Nguyen

```
56      tdesOut.close();
57    }
58    else
59    {
60      StoreOutputStream sos = new StoreOutputStream(tdesOut, storeStream);
61      msg.writeXML(sos);
62      sos.close();
63    }
64  }
65 }
```

Listing B.18: TF11: OSCI - Slice NH_NX NO_EXC_E SOAPMessageEncrypted.java

```
1  class SOAPMessageEncrypted extends OSCIMessage
2  {
3
4    private String symmetricCipherAlgorithm = null;
5      xml_0 = ...
6      xml_1a = ...
7      xml_1b = ...
8      xml_2 = ...
9      xml_3 = ...
10   public SOAPMessageEncrypted(OSCIMessage msg, OutputStream storeStream)
11   throws NoSuchAlgorithmException
12   {
13     this.msg = msg;
14     if (this.msg != null)
15     {
16       this.symmetricCipherAlgorithm = msg.dialogHandler.getSymmetricCipherAlgorithm();
17       symKey = Crypto.createSymKey(symmetricCipherAlgorithm);
18     }
19     this.storeStream = storeStream;
20   }
21   @Override
22   protected long calcLength() throws IOException, OSCIException, NoSuchAlgorithmException
23   {
24     if ((stateOfMsg & STATE_COMPOSED) == 0)
25       compose();
26     len += length;
27     len += Integer.toString(length).getBytes(Constants.CHAR_ENCODING).length;
28     len += (4 * msg.boundary_string.getBytes(Constants.CHAR_ENCODING).length);
29     len += contentID.getBytes(Constants.CHAR_ENCODING).length;
30     try
31     {
32       len += Base64.calcB64Length(cipherCert.getEncoded().length);
33     }
34     if (msg.base64)
35     {
36       len += 6;
37       len += Base64.calcB64Length(Crypto.calcSymEncLength(msg.calcLength(),
           symmetricCipherAlgorithm));
38     }
39     else
```

```
40    {
41      len += 5;
42      len += Crypto.calcSymEncLength(msg.calcLength(), symmetricCipherAlgorithm);
43    }
44    return len;
45  }
46  @Override
47  protected void compose() throws OSCIException, NoSuchAlgorithmException, IOException
48  {
49    if ((msg.stateOfMsg & STATE_COMPOSED) == 0)
50      msg.compose();
51    if (msg instanceof OSCIRequest)
52      cipherCert = msg.getDialogHandler().getSupplier().getCipherCertificate();
53    else
54      cipherCert = msg.getDialogHandler().getClient().getCipherCertificate();
55    encSymKey = Base64.encode(Crypto.doRSAEncryption(cipherCert,
56           msg.getDialogHandler().getAsymmetricCipherAlgorithm()))
57           .getBytes(Constants.CHAR_ENCODING);
58    algo = symmetricCipherAlgorithm.getBytes(Constants.CHAR_ENCODING);
59    asymAlgo = constructEncryptionAlgo();
60    length = xml_0.length + algo.length + xml_1a.length + asymAlgo.length + xml_1b.length +
          xml_2.length
61    stateOfMsg |= STATE_COMPOSED;
62  }
63  private byte[] constructEncryptionAlgo() throws UnsupportedEncodingException
64  {
65    if (Constants.ASYMMETRIC_CIPHER_ALGORITHM_RSA_OAEP.equals(msg.getDialogHandler()
66    .getAsymmetricCipherAlgorithm()))
67    {
68      if (DialogHandler.getDigestAlgorithm().equals(Constants.DIGEST_ALGORITHM_SHA512))
69      {
70      }
71      else // default
72      {
73      }
74    }
75    else
76      ret = Constants.ASYMMETRIC_CIPHER_ALGORITHM_RSA_1_5 + "\">";
77    return ret.getBytes(Constants.CHAR_ENCODING);
78  }
79  @Override
80  protected void writeXML(OutputStream out)
81    throws IOException, OSCIException, NoSuchAlgorithmException
82  {
83    if ((stateOfMsg & STATE_COMPOSED) == 0)
84      compose();
85             + ">\r\n").getBytes(Constants.CHAR_ENCODING));
86    try
87    {
88      out.write(Base64.encode(cipherCert.getEncoded()).getBytes(Constants.CHAR_ENCODING));
89    }
90    out.write((msg.base64 ? "text/base64" : "application/octet-stream").getBytes(Constants.
          CHAR_ENCODING));
91    out.write(("\r\nContent-Transfer-Encoding: ").getBytes(Constants.CHAR_ENCODING));
92    out.write((msg.base64 ? "7bit" : "binary").getBytes(Constants.CHAR_ENCODING));
93    out.write(("\r\nContent-ID: <osci_enc>\r\n\r\n").getBytes(Constants.CHAR_ENCODING));
94    if (msg.base64)
```


© Philip Phu Dang Hoan Nguyen

```
95      {
96        b64out = new Base64OutputStream(out, false);
97        tdesOut = new SymCipherOutputStream(b64out, symKey, symmetricCipherAlgorithm, true);
98      }
99      else
100       tdesOut = new SymCipherOutputStream(out, symKey, symmetricCipherAlgorithm, true);
101     if (storeStream == null)
102     {
103       msg.writeXML(tdesOut);
104       tdesOut.close();
105     }
106     else
107     {
108       StoreOutputStream sos = new StoreOutputStream(tdesOut, storeStream);
109       msg.writeXML(sos);
110       sos.close();
111     }
112     if (msg.base64)
113       b64out.flush(true);
114   }
115 }
```

Listing B.19: TF11: OSCI - Slice NO_EXC NO_EXC_E SOAPMessageEncrypted.java

```
1  public class SymCipherOutputStream extends FilterOutputStream
2  {
3
4    // private static Log log = LogFactory.getLog(SymCipherOutputStream.class);
5    private int index = 0;
6    public SymCipherOutputStream(OutputStream outStream, SecretKey symKey, String symAlgorithm,
         boolean encrypt)
7      throws IOException
8    {
9      this(outStream, symKey, symAlgorithm, encrypt, null);
10   }
11   SymCipherOutputStream(OutputStream outStream,
12             SecretKey symKey,
13             String symAlgorithm,
14             boolean encrypt,
15             byte[] iv)
16             throws IOException
17   {
18     super(outStream);
19     this.symKey = symKey;
20     this.encrypt = encrypt;
21     try
22     {
23       if (DialogHandler.getSecurityProvider() == null)
24       {
25         cipher = Cipher.getInstance(Constants.JCA_JCE_MAP.get(symAlgorithm));
26       }
27       else
28       {
```

```
29        cipher = Cipher.getInstance(Constants.JCA_JCE_MAP.get(symAlgorithm),
30        DialogHandler.getSecurityProvider());
31      }
32      if (symKey.getAlgorithm().startsWith("DESede"))
33        this.iv = new byte[8];
34      else
35        this.iv = new byte[16];
36      if (encrypt)
37      {
38        if (iv == null)
39          this.iv = Tools.createRawRandom(this.iv.length);
40          System.arraycopy(iv, 0, this.iv, 0, this.iv.length);
41        IvParameterSpec algoParamSpec = new javax.crypto.spec.IvParameterSpec(this.iv);
42        cipher.init(Cipher.ENCRYPT_MODE, symKey, algoParamSpec);
43        outStream.write(this.iv);
44      }
45    }
46  }
47  @Override
48  public void write(int b) throws IOException
49  {
50    write(new byte[]{(byte)b}, 0, 1);
51  }
52  @Override
53  public void write(byte[] b, int off, int len) throws IOException
54  {
55    if ((!encrypt) && (index < iv.length))
56    {
57      if (len < (iv.length - index))
58      {
59        for ( int i = 0 ; i < len ; i++ )
60          iv[index++] = b[off + i];
61      }
62      else
63      {
64        for ( i = 0 ; i < (iv.length - index) ; i++ )
65          iv[i + index] = b[off + i];
66        index += i;
67        try
68        {
69          IvParameterSpec algoParamSpec = new javax.crypto.spec.IvParameterSpec(iv);
70          cipher.init(Cipher.DECRYPT_MODE, symKey, algoParamSpec);
71          tmp = cipher.update(b, off + i, len - i);
72          if (tmp != null)
73            out.write(tmp);
74        }
75      }
76    }
77    else
78    {
79      if (b.length > 0)
80      {
81        tmp = cipher.update(b, off, len);
82        if (tmp != null)
83          out.write(tmp);
84      }
85    }
```


© Philip Phu Dang Hoan Nguyen

```
86   }
87   @Override
88   public void close() throws IOException
89   {
90     try
91     {
92       out.write(cipher.doFinal());
93     }
94     out.flush();
95   }
96 }
```

Listing B.20: TF11: OSCI - Auszug Slice NO_EXC NO_EXC_E SymCipherOutputStream.java

B.4. PDF-Ausgaben

B.4.1. TF02: Mehrfache Callee-Anweisungen

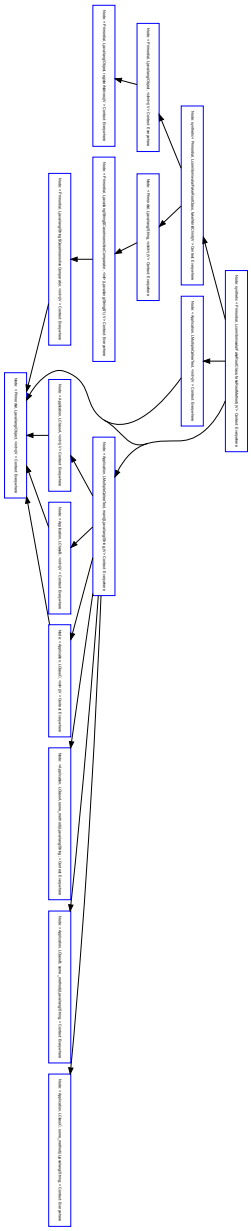

Abbildung B.1.: TF02: Vollständiger Callgraph

© Philip Phu Dang Hoan Nguyen

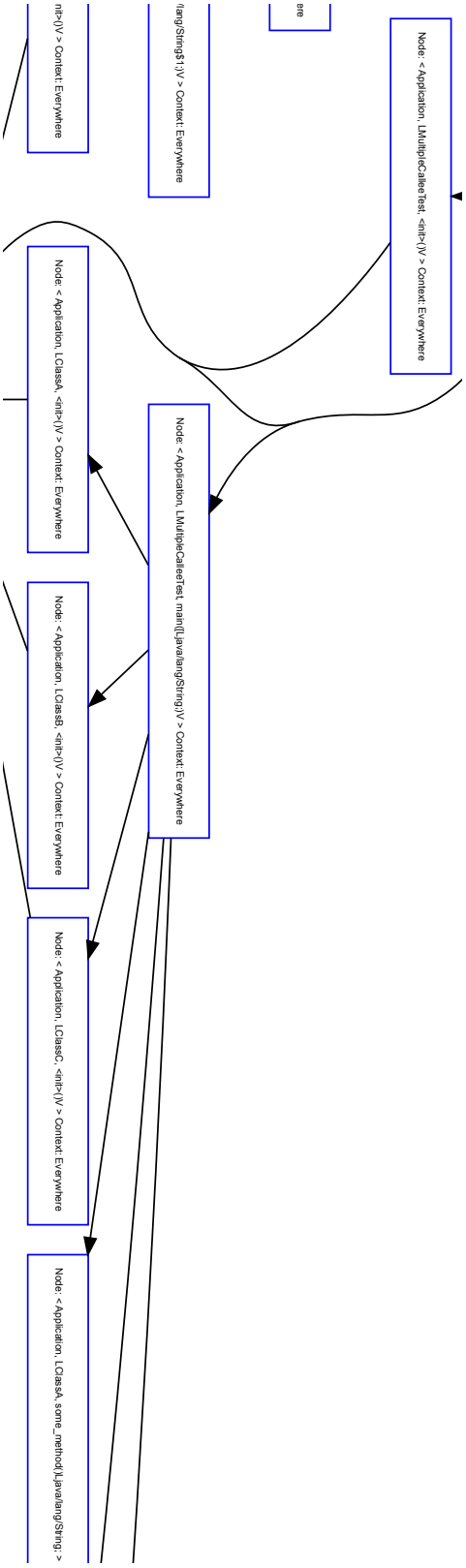

Abbildung B.2.: TF02: Ausschnitt Callgraph

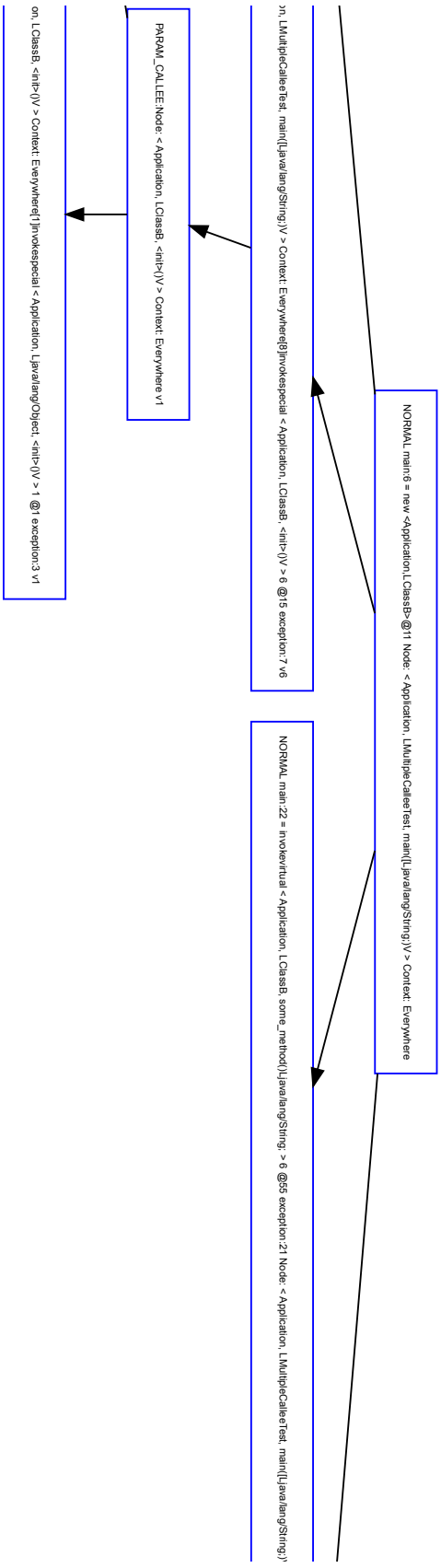

Abbildung B.3.: TF02: Ausschnitt SDG

© Philip Phu Dang Hoan Nguyen