Karlsruher Institut für Technologie

Software Design and Quality

Iterative Quelltextanalyse für Informationsflusssicherheit zur Überprüfung von Vertraulichkeit auf Architekturebene

Masterarbeit von

Jonas Lehmann

An der KIT-Fakultät für Informatik
KASTEL – Institut für Informationssicherheit und Verlässlichkeit

1. Prüfer/Prüferin:	Prof. Dr. Ralf H. Reussner
2. Prüfer/Prüferin:	PD Dr. Robert Heinrich
1. Betreuer/Betreuerin:	M. Sc. Frederik Reiche
2. Betreuer/Betreuerin:	Dr. Christopher Gerking

30. September 2023 – 03. Juni 2024

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Quellen und Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, 3. Juni 2024

. . . . . .

(Jonas Lehmann)

Zusammenfassung

Heutige Softwaresysteme, die Daten mit unterschiedlicher Vertraulichkeit verarbeiten, müssen strenge Sicherheitsanforderungen erfüllen. Sicherheitsanalysen sollten bereits in der Entwurfsphase auf Architekturebene eingeplant werden, da Fehlerbehebungen ein wachsender Kostenfaktor sind, je später sie im Entwicklungsprozess durchgeführt werden.

Auch wenn Architekturanalysen Verletzungen von Sicherheitsspezifikationen früh aufdecken können, schließt ein vermeintlich sicherer Entwurf nicht aus, dass zusätzliche Schwachstellen erst durch die Implementierung in das System gelangen. Daher gibt es Kopplungsansätze, die aus Quelltextanalysen gewonnene Implementierungsdetails in die Architekturanalyse einbinden. So werden Schwachstellen aufgedeckt, die ohne Kopplung unentdeckt bleiben.

Bisherige Kopplungsansätze führen die Quelltextanalyse jedoch nur einmalig durch und projizieren alle Ergebnisse auf einmal zurück in die Architektursicht. Hierbei gab es Indikatoren, dass es in einigen Anwendungsszenarien durch die einmalige Ausführung zu Genauigkeits-, Performanz-, und Skalierbarkeitsproblemen kommt.

Diese Masterarbeit adressiert die Probleme mit einem iterativen Ansatz, der die Informationen für die Quelltextanalyse partitioniert und die Analyse damit mehrfach ausführt. Da die Genauigkeit für manche Sicherheitsbegriffe außerdem von der Zusammenführung der Implementierungsdetails abhängt, kann die Architekturanalyse im iterativen Ansatz auch mehrfach auf einzelnen dieser Details ausgeführt werden.

Der iterative Ansatz wurde fallstudienbasiert evaluiert, um die Auswirkung auf Genauigkeit, Performanz und Skalierbarkeit im Vergleich zum nicht-iterativen Ansatz zu überprüfen. Im Bereich der Genauigkeit hat sich die Sensitivität gefundener Verletzungen in der Architektur erhöht, z. B. in Verbindung mit der Quelltextanalyse JOANA von 0,3 auf 0,95. Mit der steigenden Anzahl an Analysedurchführungen im iterativen Ansatz verlängert sich die Ausführungszeit, was die Performanz vermindert. Jedoch erlaubt der iterative gegenüber dem nicht-iterativen Ansatz größere Eingaben, wie sich in der Skalierbarkeitsevaluation gezeigt hat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Viele der heutigen Systeme, die Nutzeraccounts auf verschiedenen Berechtigungs- oder Bezahlstufen anbieten, verarbeiten Daten mit unterschiedlicher Vertraulichkeit. Für diese Systeme müssen Sicherheitsanforderungen wie z. B. Datenverschlüsselung gewährleistet werden, was u. a. von der DSGVO vorgeschrieben wird [1]. Da das Beheben von Entwurfsfehlern teurer wird, je später in der Entwicklung sie gefunden werden, ist es wichtig, schon vor der Implementierung die Sicherheit solcher Systeme zu untersuchen [2]. In der Modellgetriebenen Softwareentwicklung werden bereits im Entwurf Aussagen über die Sicherheitseigenschaften von geplanten Softwaresystemen möglich, wie ein Ansatz Seifermanns [3] oder die Spezifikationserweiterung UMLSec [4] zeigen.

Mit Architekturanalysen ist es möglich, Verletzungen von annotierten Sicherheitsspezifikationen zu finden, wie es Kramer et al. zeigen [5]. Auf der Komponentensicht annotieren sie hierfür Sicherheitslevel an Daten und Systemnutzer und stellen Relationen von erlaubten Zugriffen auf. Die zugehörige Architekturanalyse erkennt Inkonsistenzen in diesen Relationen, wodurch Schwachstellen in der Vertraulichkeit bereits auf Architekturebene aufgedeckt werden können.

Einige dieser Annotationen hängen von der Implementierung des modellierten Systems ab. Durch Abhängigkeiten wie Lese- und Schreiboperationen von Zustandsvariablen können auf Quelltextebene Informationen, die durch anliegende Sicherheitslevel als vertraulich eingestuft sind, an andere Programmteile weitergegeben werden, die als nicht vertraulich gelten. Wenn diese Informationsflüsse ungewollt durch Programmierfehler entstehen, kann beispielsweise bereits die Reihenfolge von Methodenaufrufen Aufschluss über das Sicherheitslevel der involvierten Daten geben [6]. Bezüglich der Vertraulichkeitseigenschaft sind solche Informationsflüsse unerwünscht. Da eine Architekturanalyse auf einer höheren Abstraktionsebene arbeitet und die konkrete Implementierung zum Analysezeitpunkt noch nicht vorliegt, kann sie solche invaliden Informationsflüsse nicht finden. Eine Kopplung einer Architekturanalyse mit einer Informationsflussanalyse auf Quelltextebene bietet jedoch eine Lösung, um Verletzungen auf beiden Abstraktionsebenen miteinander zu verknüpfen. Hierfür wird überprüft, ob die Informationsflüsse konform zu der Modellierung auf Architekturebene sind, wie es verschiedene Ansätze umsetzen [7] [8] [9].

Existierende Kopplungsansätze wie der von Häring [9] führen die Quelltextanalyse nur einmalig durch und transformieren alle Informationen auf einmal in die Architekturanalyse. Hierbei können aber Probleme in der Skalierbarkeit, Genauigkeit und Performanz auftreten. Um diesen Problemen zu begegnen, stellt diese Masterarbeit einen iterativen Kopplungsansatz vor, der die Quelltextanalyse nicht nur einmal ganzheitlich, sondern durch partitionierte Eingaben mehrmals ausführt.

Große Sicherheitsverbände, die bei Häring als Eingabe für die Quelltextanalyse komplett dargestellt werden, skalieren schlecht. Der entwickelte iterative Ansatz nutzt eine optimierte Verbandsgenerierung, um die Skalierbarkeit zu verbessern.

Um das Architekturmodell mit Informationen über die gefundenen Informationsflüsse anzureichern, ist eine Rückprojektion und Neuberechnung der betroffenen Sicherheitslevel nötig. Häring nutzt hierfür einen Operator, der eine Unterabschätzung in Kauf nimmt [9], wodurch keine vollständige Abdeckung der Verletzungen erreicht wird. Mit dem iterativen Ansatz werden die Ergebnisse in einzelne Modelle rückprojiziert und separat durch eine Kopplung mit der Architekturanalyse ausgewertet, wodurch die Genauigkeit optimiert wird.

Damit für die Quelltextanalyse alle Flüsse als Annotationen im Quelltextmodell abgedeckt werden, entstehen bei Häring große Eingabedateien, die einen hohen Arbeitsspeicherverbrauch erzeugen [9]. Durch eine Partitionierung dieser Eingaben sollen leichtgewichtigere Analyseausführungen ermöglicht und die Performanz erhöht werden.

Um den Ansatz zu evaluieren, wird die Fallstudie *TravelPlanner* von Katkalov et al. [10] genutzt. Sie wurde erweitert, um die Äquivalenzklassen von denjenigen Flüssen vollständig abzudecken, die sich in der Rückprojektion ins Architekturmodell unterscheiden und so die Genauigkeit der Analyseergebnisse beeinträchtigen können. Die Evaluationsergebnisse zeigen, dass der iterative Ansatz die Genauigkeit verbessern kann. Die Performanz in der gewählten Umsetzung verschlechtert sich unwesentlich, aber die Skalierbarkeit wird optimiert.

Diese Arbeit ist wie folgt gegliedert: Zuerst werden Grundlagen und relevante Begriffe in Kapitel 2 vorgestellt. Kapitel 3 soll die vorliegende gegenüber verwandten Arbeiten abgrenzen. Ein fortlaufendes Beispiel wird durch das Reiseplanungssystem *TravelPlanner* in Kapitel 4 eingeführt. In Kapitel 5 wird der iterative Analyseansatz vorgestellt und Iterationsarten und deren Mechanismen eingeordnet. Es folgt die Optimierung der Verbandsgenerierung in Kapitel 6. Kapitel 7 beschreibt die Umsetzung des Ansatzes als Erweiterung eines bestehenden Frameworks und die Implementierung von konkreten Konfigurationen für die zwei gewählten Analyseprogramme JOANA und CodeQL. Die Genauigkeit, Performanz und Skalierbarkeit wird in Kapitel 8 evaluiert. Abschließend fasst Kapitel 9 die Inhalte dieser Arbeit in übersichtlicher Weise zusammen.

2 Grundlagen

Für diese Arbeit sind die Themenbereiche der Sicherheitsmodellierung und der Sicherheitsanalysen wichtig. In Abschnitt 2.1 werden die Grundlagen zur Modellierung von Sicherheitsbegriffen und zur Darstellung und Interpretation von Sicherheitsleveln beschrieben. Es folgt in Abschnitt 2.2 eine Übersicht über statische Analysen, die entweder auf Architekturebene oder auf Quelltextebene verschiedene Sicherheitsanforderungen überprüfen. Zuletzt wird in Abschnitt 2.3 ein Framework vorgestellt, das die Ausführung derartiger Analysen automatisiert.

2.1 Modellierung von Sicherheitssystemen

Zuerst wird der Kontext der Modellgetriebenen Softwareentwicklung in Unterabschnitt 2.1.1 eingeführt. Im Anschluss folgt eine Definition von Sicherheitseigenschaften in Unterabschnitt 2.1.2, die Beschreibung ihrer Verknüpfung in Unterabschnitt 2.1.3 und eine Taxonomie verschiedener Sicherheitsleveldarstellungen in Unterabschnitt 2.1.4. Es werden in Unterabschnitt 2.1.5 Sicherheitsverbände als mathematisches Konstrukt eingeführt, um verknüpfte Sicherheitslevel zu repräsentieren. In Unterabschnitt 2.1.6 wird zuletzt der Assume-Guarantee Ansatz vorgestellt, um Probleme auf Teilprobleme aufzuteilen.

2.1.1 Modellgetriebene Softwareentwicklung

Die Modellgetriebene Softwareentwicklung (engl.: model-driven software development, MDSD) stellt Techniken dar, bei denen Quelltext automatisiert aus einem Modell erzeugt wird [11]. Da die Generierung von Quelltext ausgelagert wird, kann der Systementwurf unabhängig von späteren Implementierungsfragen auf einem höheren Abstraktionslevel durchgeführt werden. Es werden unter den Entwicklern verschiedene Rollen zugewiesen, so dass sie sich entweder ganz auf den Entwurf konzentrieren und auf Architekturlevel arbeiten oder sich als Programmierer ganz auf die Quelltextebene konzentrieren und so die Gesamtsicht auf das Systems unwichtig ist [12]. Durch diese Trennung der Belange kann Komplexität reduziert werden und die Produktivität der Entwickler erhöht sich [2]. Die Entwicklungsdauer wird reduziert, Anforderungen werden auf einem höheren Abstraktionslevel leichter definierbar und Testfälle können klarer verständlich beschrieben werden. Durch Analysewerkzeuge auf Architektursicht lassen sich bereits in frühen Phasen auf abstrakter Modellebene Aussagen über Verhalten und Performanz gewinnen und bieten so schnelle Entscheidungsgrundlagen zum Systementwurf [13].

Modelle Modelle reduzieren zu diesem Zweck die Komplexität. Nach Stachowiaks [14] allgemeiner Modelltheorie muss ein Modell drei grundlegende Bedingungen erfüllen. Es muss erstens ein Original repräsentieren. Zweitens muss es eine Reduktion der Teile des Originals auf die notwendigen Teile durchführen, die in der untersuchten Domäne interessant sind. Unwichtige Teile werden vernachlässigt. Drittens steht ein Modell stellvertretend für ein Subjekt, das modelliert wird, um einen bestimmten Zweck zu untersuchen. In der Softwaretechnik bringen Modelle den Vorteil, dass alle Teile eines Originals entfallen, die für die untersuchte Domäne unwichtig sind. Es können zwar aus verschiedenen Sichten mehrere Modelle für dasselbe Original erstellt werden. Aber jedes dieser Modelle ist eine verständlich zugeschnittene Darstellung der gewünschten Domäne. Die Komplexität wird aufgegliedert und ermöglicht es, mit leichtgewichtigeren und somit kostengünstigeren Objekten zu arbeiten. Vorhersagen können schneller durchgeführt werden und der Speicheraufwand sinkt.

Metamodelle Die Definition von Modellen wird ebenfalls durch Modelle, die sogenannten Metamodelle vorgenommen [15]. Sie enthalten auf einem höheren Abstraktionsgrad die Regeln, Werte und Elemente, die ein Modell ausmachen und die zur Erstellung einer Modellinstanz genutzt werden.

Modelltransformation Eine Modelltransformation überführt ein Modell von einer Domäne in eine andere [16]. Die Transformation ist eine Sequenz von Bearbeitungsoperationen, durch die sich das Ausgangsmodell abändern lässt, um zur anderen Repräsentation zu werden[17]. Eine solche Transformation muss syntaktisch und semantisch korrekt sein. Wenn zwei Modelle des gleichen Originals in zwei Domänen genutzt werden, ist es wünschenswert, dass ihre enthaltenen Informationen widerspruchsfrei bleiben. Es muss deshalb eine Verbindung zwischen den Werten beider Modelle bestehen, über die die Modelle synchronisiert werden. Dies ist einerseits durch ein geteiltes Metamodell möglich, oder eine Vermittlerstruktur muss von außen Veränderungen beobachten und bei Aktualisierungen von einem Modell ebenfalls das unveränderte, andere Modell benachrichtigen und aktualisieren[18].

Tripel-Graph-Grammatiken In diesem Kontext existieren sogenannte Tripel-Graph-Grammatiken [19], um bidirektional zwei Domänengraphen über einen Korrespondenzgraphen zu verknüpfen [20]. Die Knoten des Korrespondenzgraphen zeigen auf Knotenmengen der beiden anderen Graphen und bilden Regeln, welche Teile miteinander verknüpft werden [21].

Palladio-Komponentenmodell Palladio ist ein Modellierungsansatz für Systemarchitektur, um Qualitätsvorhersagen über das Design der Softwarekomponenten zu treffen [13]. Das Palladio-Komponentenmodell (PCM) als Domänenspezifische Sprache stellt die Basis zur Spezifizierung und Dokumentation der Modellkomponenten dar. Eine Instanz des PCM besteht aus mehreren Metamodellen, die jeweils eine Sicht auf das Softwaresystem abbilden. Für die Vertraulichkeitsanalyse wird das PCM außerdem um entsprechende Metamodelle erweitert, die Spezifikationen aus der Sicherheitsdomäne enthalten [5]. Im Folgenden werden die für diese Arbeit relevanten Sichten beschrieben.

In einem *Repository*-Modell werden Softwarekomponenten und ihre Schnittstellen beschrieben, sowie das Verhalten der Komponenten in *Service Effect Specifications* (SEFFs). In einem *System*-Modell wird die Softwarearchitektur festgehalten und bildet ab, wie sich die Instanzen der Komponenten verknüpfen.

Im *Resourceenvironment*-Modell werden technische Ressourcen wie z. B. Server mit ihren Eigenschaften beschrieben. Für alle Ressourcen eines bestimmten Teils des Systems vereint ein Container die jeweiligen allokierten Teile, z. B. CPUs, Festplatten und Netzwerkverbindungen. Es werden dort technische Spezifikationen wie Verarbeitungsraten gespeichert. In einem *Allocation*-Modell wird zusätzlich gespeichert, wie die Dienste des Systems auf die Ressourcen verteilt werden.

Um vertraulichkeitsrelevante Eigenschaften zu modellieren, wird das PCM durch ein *Confidentiality*-Modell erweitert [5], in dem DataSets spezifiziert werden. Durch diese DataSets können Daten durch beliebige Kriterien gruppiert werden, z. B. zu Sicherheitsleveln, Rollen, Nutzern oder anderen Eigenschaften. Den Parametern des *Repository*-Modells werden über einen stereotypischen Informationsfluss bestimmte DataSets zugeordnet. Außerdem werden die Akteure (Adversaries) modelliert, die mit dem System interagieren und somit als Angreifer infrage kommen. Ihnen werden DataSets zugeordnet, die sie sehen dürfen. Die Ressourcen des *Resourceenvironment*-Modells werden erweitert, um einen geografischen Ort sowie vorhandene Schutzmechanismen zu annotieren. So kann eine Zuordnung der Akteure zu den Komponenten des Systems über den geografischen Ort erfolgen, über den sie Zugriff erlangen.

2.1.2 Sicherheitseigenschaften

In der Domäne der Softwaresicherheit gibt es viele Begriffe, die Aussagen über den Zustand eines Systems machen wie beispielsweise Verfügbarkeit, Vertraulichkeit, Verlässlichkeit, und Integrität [22]. Da es in dieser Arbeit um den Kontext der Berechtigungsverteilung geht, sind insbesondere Vertraulichkeit und Integrität wichtig. Zuvor werden grundlegende Begrifflichkeiten eingeführt. In Abbildung 2.1 sind drei Sicherheitslevel angegeben, von denen Geheim die höchste und Öffentlich die niedrigste Stufe ist.

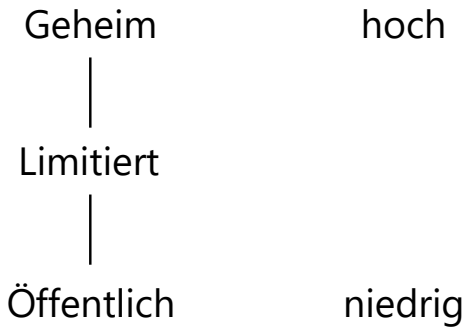


Abbildung 2.1: Beispiel für Sicherheitslevel

Eine Sicherheitsanalyse führt eine Überprüfung eines Systems anhand vorher festgelegter Regeln durch und dient dazu, verletzte Sicherheitsabhängigkeiten zu finden. Je nach Analyseart beschreiben die Regeln, welche Systemteile mit Sicherheitsleveln annotiert werden und was die Level bedeuten. In einer Informationsflussanalyse von Tuma et al. [23] werden beispielsweise Daten und Angreiferzonen mit Leveln markiert. Im Access Analysis-Ansatz Kramers [5] werden Angreifer und auch Informationen wie z. B. Parameter mit Sicherheitsleveln belegt. Seifermann [3] annotiert Vertraulichkeitslevel an Ressource Container im Palladiomodell und Nutzerdaten besitzen ebenfalls Sicherheitslevel. Für Analysen legen Regeln die Sicherheitslevel und deren Abhängigkeiten fest. Informationsflüsse ergeben sich aus Daten und Operationen auf diesen Daten, die durch ein Programm ausgeführt werden. Im Allgemeinen weist eine Informationsflussanalyse den Daten Sicherheitslevel hinzu und spezifiziert, welche Flüsse erlaubt sind und welche nicht. Laut Denning [24] liegt ein invalider Informationsfluss vor, wenn Daten von höhervertraulichen Daten abhängig sind. Die Analyseergebnisse bestehen aus Verletzungen der definierten Flussregeln und deren erzeugenden Konstellationen von Sicherheitsleveln.

Eine Informationsflussanalyse setzt einen Sicherheitsbegriff um. Im Folgenden werden die Begriffe Vertraulichkeit und Integrität beschrieben, die sich konzeptuell in ihren jeweiligen erlaubten Berechtigungsflüssen unterscheiden. Hierfür soll nun angenommen werden, dass Systemnutzer ein Zugriffslevel und Elemente im System ein Klassifikationslevel besitzen. Für beide Parteien beschreibt das Level, welche Daten dem Nutzer oder dem Programmelement sichtbar sind. Ein Nutzer mit einem bestimmten Zugriffslevel darf auf alle Elemente zugreifen, die das gleiche oder ein geringeres Klassifikationslevel haben. Ein System, das Vertraulichkeit erfüllt, darf keine Flüsse von höher vertraulichen Daten zu niedriger vertraulichen Daten haben. Denn sonst dürften Nutzer mit einem geringen Zugriffslevel auf Elemente mit ihrem Klassifikationslevel zugreifen, an dem aber Daten mit hohem Sicherheitslevel liegen. Ein Modell für Vertraulichkeit stellt das Bell-LaPadula-Modell dar [25]. Ein System, das Integrität erfüllt, darf keine Informationsflüsse von niedriger vertraulichen Daten zu höher vertraulichen Daten haben. Denn die hoch vertraulichen Daten sollen unverändert und vollständig bleiben. Hierfür existiert ein Modell von Biba [26].

Für den Informationsfluss zwischen zwei Elementen, die mit Sicherheitsleveln annotiert wurden, folgt daraus, dass er nur valide bezüglich der Vertraulichkeit ist, falls er von niedrig nach hoch stattfindet. Denn sonst könnte ein Angreifer auf Daten zugreifen, für die er keine Rechte besitzt. Ein System ist in diesem Kontext also vertraulich, wenn es keine Interferenzen, also hoch klassifizierte nach niedrig klassifizierten Datenflüsse gibt, wie es Goguen und Meseguer [27] ableiten. Existiert doch ein solcher Fluss von einer Fluss-Quelle zu einer Fluss-Senke, kann eine Sicherheitsanalyse ihn als invalide erkennen. Er kann dann auf zwei Arten in einen validen Fluss zurücktransformiert werden [28]. Entweder wird das Klassifikationslevel des Quellelements herabgestuft, wodurch ausgesagt wird, dass dessen Daten auch für Bereiche mit niedrigerer Klassifikation sichtbar werden. Oder das Sicherheitslevel der Senke wird erhöht, wodurch Nutzer mit den vorherigen Zugriffsleveln keine Zugriffserlaubnis mehr haben. Diese zwei Arten der Modifikation können nach einer Informationsflussanalyse angewandt werden, um das System in einen Zustand zu überführen, in dem die Ergebnisse der Analyse weiter nutzbar sind.

2.1.3 Verknüpfung von Sicherheitseigenschaften

Es ist möglich, Sicherheitslevel zu Mengen zusammenzufassen und Nutzern mehrere verknüpfte Berechtigungen zuzuweisen. In Abbildung 2.2 ist links das CANUKUS-Beispiel von Bell und LaPadula [29] skizziert. Es beschreibt die Klassifizierung von Daten in ihrer Sichtbarkeit für Regierungsbehörden von Kanada (CAN), den Vereinigten Staaten (US) und des Vereinigten Königreichs (UK). Die Verknüpfung der Sicherheitslevel kann auf zwei Arten vorgenommen werden, konjunktiv oder disjunktiv [25]. Durch beide Verknüpfungsarten werden jedoch unterschiedliche Annahmen dargestellt, die im Folgenden beschrieben werden. Im disjunktiven Fall sind Daten höher klassifiziert, wenn sie für weniger Behörden sichtbar sind, d. h. US_only hat ein höheres Sicherheitslevel als UKUS_only. Die disjunktive Verknüpfung spiegelt sich darin wider, dass ein Nutzer mit Berechtigung US auf alle Daten zugreifen darf, die in ihrem Sicherheitslevel US enthalten, also US_only, UKUS_only, CANUS_only und CANUKUS_only. Fließen Informationen von einem verknüpften Sicherheitslevel zu einem einzelnen Sicherheitslevel, fließen sie von niedrig nach hoch und sind somit valide [27]. Invalide Flüsse hingegen sind ein Übergang von einzelnen zu gekoppelten Verbindungen und dürfen in einem sicheren System nicht vorkommen.

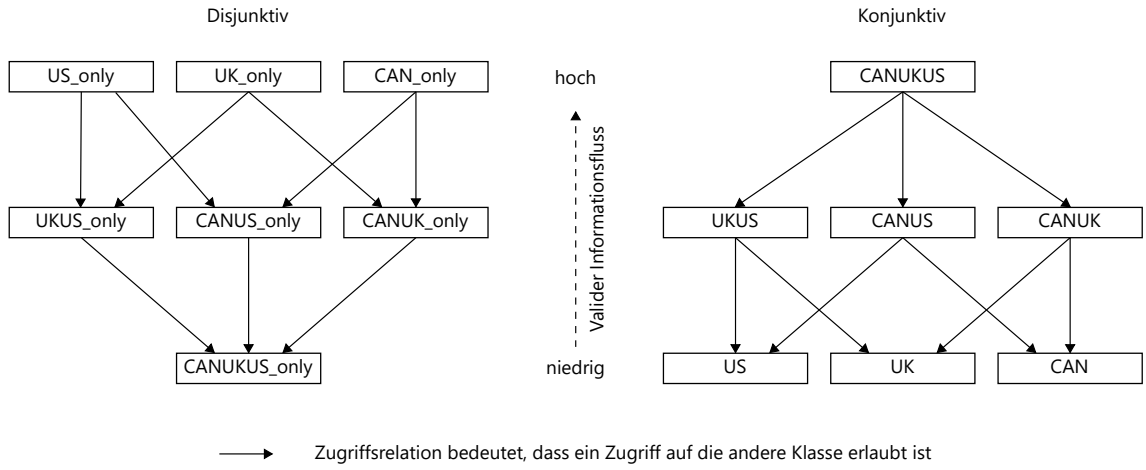


Abbildung 2.2: Disjunktive und konjunktive Verknüpfung am CANUKUS-Beispiel

Eine andere mögliche Sichtweise ist die konjunktive Verknüpfung, wie sie auf der rechten Seite der Abbildung 2.2 dargestellt ist und im ursprünglichen Beispiel von Bell und LaPadula angewandt wird [29]. Ein Mitarbeiter erhält nur Zugriff, wenn der Mitarbeiter alle beinhalteten Berechtigungen für die Daten besitzt, auf die er zugreifen will. Hat er nur US Rechte, darf er nicht auf UKUS zugreifen, denn er bräuchte dafür zusätzlich die Rechte für UK. Aus diesen komplementären Sichtweisen wird deutlich, dass bei der Verknüpfung von Sicherheitsleveln genau spezifiziert werden muss, ob diese eine Konjunktion oder Disjunktion der Teilrechte ist. Denn bei der disjunktiven Verknüpfung sind einzelne Rechte hoch klassifiziert und gekoppelte niedrig. Und bei der konjunktiven Verknüpfung wiederum sind einzelne Rechte niedrig klassifiziert und gekoppelte Rechte hoch. Für eine Analyse hat dies zur Folge, dass Informationsflüsse genau entgegengesetzt als valide und invalide klassifiziert werden.

2.1.4 Taxonomie von Sicherheitsleveln zur Vertraulichkeit

Wie bereits in den vorigen Abschnitten erläutert, kann Vertraulichkeit durch Sicherheitslevel modelliert werden, die an Daten und Akteuren annotiert werden. Abbildung 2.3 bietet einen Überblick, wie sich diese Sicherheitslevel repräsentieren lassen. Im Folgenden wird zu jeder dargestellten Klasse beschrieben, wie die Level aus einer Klasse interpretiert werden und welche Flüsse zwischen zwei Leveln legal und welche illegal bezüglich der Vertraulichkeit sind. Grundlegend werden die Darstellungen in unverknüpfbare und in verknüpfbare Level unterteilt (vgl. Abbildung 2.3). In beiden Darstellungsarten werden im Folgenden weitere Teilklassen beschrieben.

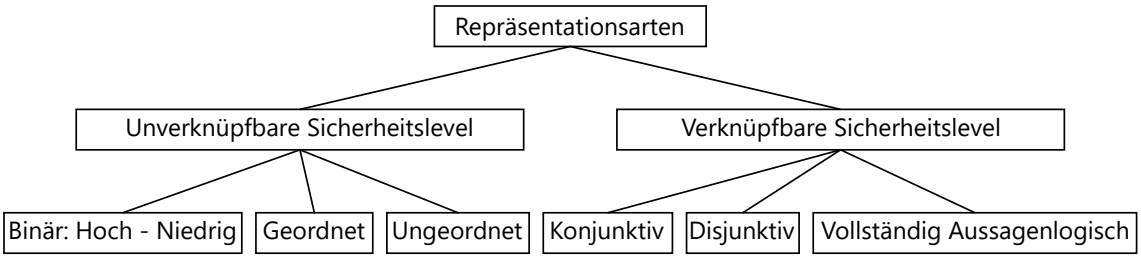


Abbildung 2.3: Übersicht über die Repräsentationsarten von Sicherheitsleveln

2.1.4.1 Unverknüpfbare Sicherheitslevel

Binär: Hoch - Niedrig In der Fallstudie *JPMail* von Tuma et al. [23] werden zwei verschiedene Level benutzt: Hoch (High) und Niedrig (Low). Dies ist der einfachste Fall der Vertraulichkeitsmodellierung, denn es gibt nur diese zwei exklusiven Sicherheitslevel. Hierbei macht es semantisch keinen Sinn, dass die beiden Level verknüpft werden, denn die Zuordnung eines Levels zu einem Datum soll immer so minimal wie möglich sein, damit nur die wirklich nötigen Einschränkungen vorliegen. Bei einem Datum, das sowohl als Hoch und Niedrig klassifiziert werden könnte, muss anhand der modellierten Domäne abgewogen werden, welche der beiden Klassen zutrifft. Besonders eignet sich ein binäres Sicherheitslevel, wenn bei der Untersuchung eines Systems ein primärer Sicherheitsmechanismus im Fokus liegt, der an den Komponenten entweder vorliegt oder nicht. Eine Verletzung der Vertraulichkeit liegt dann vor, wenn ein Fluss von *hoch* nach *niedrig* stattfindet, denn Informationen, die vorher nur mit der Berechtigung *hoch* sichtbar waren, sind nun auch an einer Stelle sichtbar, an der mit der Berechtigung *niedrig* zugegriffen werden kann.

Geordnet Der binäre Fall lässt sich erweitern, wenn es eine strikt geordnete Menge an Klassen gibt, die als Sicherheitslevel genutzt werden. Mit den drei Leveln *Öffentlich*, *Limitiert* und *Geheim* aus Abbildung 2.1 wird das Sicherheitslevel *Öffentlich* als niedrigstes und das Sicherheitslevel *Geheim* als höchstes bezüglich der Vertraulichkeit eingeordnet. Die Sicherheitslevel stellen als geordnete Liste dar, in welche Richtung ein Informationsfluss erlaubt ist. Unerlaubte Flüsse laufen von einer Quelle mit hoch klassifiziertem Level entgegen der Listenordnung zu einer Senke mit niedriger klassifiziertem Level. Eine solche Darstellung

der Sicherheitslevel findet sich durch die drei geordneten Level [User], [User,Airline] und [User,Airline,TravelAgency] auch in der Fallstudie *TravelPlanner* von Katkalov et al. [10].

Ungeordnet Liegen Sicherheitsklassen vor, die keine Hierarchie besitzen und nur zur Trennung verschiedener Berechtigungsbereiche dienen, müssen erlaubte Flüsse als Regeln aufgezählt werden. Dies wird z. B. in der Fallstudie *PrivateTaxi* von Katkalov [10] genutzt, um Level zu annotieren, die sich gegenseitig ausschließen. Sobald ein Informationsfluss auftritt, der in keiner Regel festgehalten ist, stellt er eine Verletzung dar.

2.1.4.2 Verknüpfbare Sicherheitslevel

Konjunktiv Sobald Sicherheitslevel verknüpfbar werden, lassen sich für die Vertraulichkeit komplexere Sicherheitsmechanismen darstellen. Im konjunktiven Fall stellt ein kombiniertes Sicherheitslevel den Fall dar, dass auf den zugehörigen Ort nur zugegriffen werden darf, wenn ein Akteur alle einzelnen Sicherheitslevel besitzt, oder eine Übermenge davon. Eine solche Darstellung der Level findet sich im Bell-LaPadula-Modell [29]. Ein erlaubter Fluss verläuft in dieser Darstellung zwischen zwei Leveln, wenn die Quellenlevel des Flusses eine Untermenge der Senkenlevel sind.

Disjunktiv In Unterabschnitt 2.1.3 wurde mit dem CANUKUS-Beispiel von Bell und LaPadula [29] bereits ein Fall gezeigt, bei dem Sicherheitslevel disjunktiv verknüpft interpretiert werden. Der disjunktive Fall ähnelt dem konjunktiven. Auch hier werden verknüpfte Sicherheitslevel für komplexere Vertraulichkeitsaussagen genutzt. Ein disjunktiv kombiniertes Sicherheitslevel erlaubt aber genau dann Zugriff auf einen Ort oder eine Information, wenn ein Angreifer mindestens eines der enthaltenen einzelnen Level kennt bzw. sobald der Schnitt der Levelmengen von Ort und Angreifer nicht leer ist. Ein erlaubter Fluss verläuft in dieser Darstellung zwischen zwei Leveln, wenn die Quellenlevel des Flusses eine Obermenge der Senkenlevel sind.

Vollständig Aussagenlogisch Im konjunktiven und disjunktiven Fall stehen zur Verknüpfung von Sicherheitsleveln nur die jeweiligen Junktoren Konjunktion und Disjunktion zur Verfügung. Da dies nicht für komplexe aussagenlogische Flussregeln ausreicht, können die verfügbaren Junktoren zu einer funktional vollständigen Junktorenmenge, z. B. zu $\{\vee, \wedge, \neg\}$ erweitert werden. Im Fallbeispiel *PrivateTaxi* von Katkalov [10] werden die Regeln für die erlaubten Flüsse zwischen Leveln z. B. durch aussagenlogische Bedingungen beschrieben, in denen die Konjunktion, die Äquivalenz und die Implikation vorkommt.

2.1.5 Sicherheitsverband

Für die Beschreibung von erlaubten Informationsflussabhängigkeiten und auch anderer Relationen, wie einer Zugreifbarkeitsrelation, bietet sich die mathematische Struktur eines Verbands (engl. Lattice) an [30].

Ein Verband besteht aus einer totalen Ordnungsstruktur, bei der es für zwei Elemente ein Supremum gibt, das das kleinste Element darstellt, das größer oder gleich beiden Unterelementen ist. Ein Supremum wird für die Elemente A und B über die Konjunktion A ∧ B dargestellt [30]. Auch das Infimum ist auf einer solchen Struktur eindeutig. Es ist das größte Element, das über die Ordnungsrelation kleiner oder gleich beiden Überelementen ist. Das Infimum wird für die Elemente A und B über die Disjunktion A ∨ B dargestellt [30]. In Abbildung 2.4 ist ein Verband dargestellt, der für die Elemente A und B das Supremum A ∧ B und Infimum A ∨ B markiert. In der Literatur wird für das Supremum ebenfalls der Operator ⊗ genutzt [24].

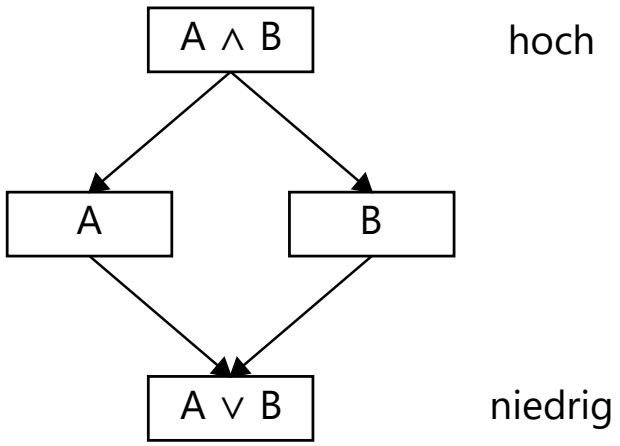


Abbildung 2.4: Beispiel für einen Sicherheitsverband

Um Sicherheitsinformationen auf einen Verband abzubilden, muss definiert werden, welche der Informationen als hoch und welche als niedrig in der Ordnung eingestuft werden. Hierzu muss die Ordnungsrelation ausdrücklich dem Sicherheitsbegriff angepasst werden. Im Folgenden wird anhand eines Beispiels beschrieben, wie das Übertragen eines Sicherheitsbegriffes auf eine Modellierung in einem Verband durchgeführt werden kann.

Es wird beispielhaft der Sicherheitsbegriff auf Architekturlevel aus Härings Ansatz [9] auf einen Verband beschrieben. Der Sicherheitsbegriff ist dort als Vertraulichkeit gewählt und eine Ordnung der Sicherheitslevel soll über die Zugriffsrelation geschehen. Hierbei gibt die Zugriffsrelation an, dass ein Sicherheitslevel genau dann über einem anderen in der Ordnung liegt, wenn es auf dieses andere zugreifen darf. Über die Auswahl von Klassen aus der Potenzmenge von Sicherheitsleveln können diese Klassen beliebige Verknüpfungen der Sicherheitslevel darstellen. Jedoch kann, wie bereits in Unterabschnitt 2.1.3 erklärt, eine Verknüpfung der einzelnen Sicherheitslevel disjunktiv oder konjunktiv definiert werden, je nachdem, wie die Sicherheitsrelation zwischen Teilen und Verknüpfungen im System vorliegen soll. Wird eine konjunktive Verknüpfung gewählt und werden die verknüpften Elemente wie in Abbildung 2.4 in der Relation als hoch eingeordnet, ist es leicht, das Supremum abzulesen, denn das ist über die Konjunktion A ∧ B definiert. Wird eine disjunktive Verknüpfung gewählt, werden verknüpfte Elemente wie in Abbildung 2.4 in der Relation niedrig angeordnet. In diesem Fall lässt sich das Infimum leicht ablesen, denn es ist über die Disjunktion A ∨ B definiert.

Wichtig ist, dass Analysen ihren erwarteten Sicherheitsverband genau definieren. Es muss die Art der Verknüpfung von Sicherheitsleveln definiert sein und die Zuordnung von verknüpften Leveln zu hoch oder niedrig innerhalb der Ordnungsrelation. Je nach Sicherheitsbegriff ist es nach der Durchführung einer Analyse wichtig, ob Suprema oder Infima berechnet werden, um neue Sicherheitslevel abzuleiten. Auch das muss vor Benutzung einer Analyse entsprechend der Verknüpfungsart ausgewählt werden.

2.1.6 Assume-Guarantee Ansatz

Oft besitzen industrielle Softwaresysteme durch ihre hohe Komplexität eine exponentielle Anzahl erreichbarer Zustände. Um trotzdem eine Analyse durchzuführen, ist es notwendig, diese Analyse auf Teilanalysen aufzuteilen. Hierbei muss formal verifiziert werden, dass die Teilanalysen die gleichen Resultate erzeugen wie eine ganzheitliche Programmanalyse.

In diesen Kontext fällt der Ansatz der Kompositionellen Argumentation (engl. Compositional Reasoning) [31]. Es ist eine Technik, um die Argumentation über den globalen Zustandsraum durch eine Argumentation auf lokaler Ebene zu ersetzen. Jede Komponente wird separat analysiert, basierend auf Annahmen über das Verhalten von anderen Komponenten.

Hierunter fällt der Ansatz Assume-Guarantee Reasoning [32] (im Folgenden AGR). Es werden Annahmen über die Umgebung einer Komponente gemacht, wodurch sie einzeln verifiziert werden kann. Danach kann eine so erzeugte Garantie als Annahme für andere Komponenten genutzt werden. Eine Formel im AGR-Paradigma kann als Tripel $\langle A \rangle\ M\ \langle P \rangle$ ausgedrückt werden, wobei M eine Komponente, A eine Annahme über die Umgebung von M, und P eine Eigenschaft ist. Sobald M in einem System liegt, das A erfüllt, muss das System auch P erfüllen. Nur dann ist die Formel wahr. Ein Beispiel einer Beweisregel ist folgende Inferenzregel [33]:

$$\frac{\langle A \rangle\ M_1\ \langle P \rangle \qquad \langle true \rangle\ M_2\ \langle A \rangle}{\langle true \rangle\ M_1\ ||\ M_2\ \langle P \rangle}$$

Um herauszufinden, ob ein System, das aus zwei Teilkomponenten M_1 und M_2 besteht, die Eigenschaft P erfüllt, ohne M_1 und M_2 zu koppeln, wird einerseits verlangt, dass P aus A durch M_1 folgt und andererseits, dass A durch M_2 gegeben ist, also aus $true$ ableitbar ist [34]. Diese Annahmen müssen jedoch manuell erstellt werden, was den eigentlichen Aufwand der Methode ausmacht [35].

Der AGR-Ansatz kann ebenfalls auf Architekturmodelle angewandt werden, wie es Cofer et al. [36] beschreiben. Hierbei wäre P die Garantie einer korrekten Implementierung. Ohne nun die gesamte Implementierung zu testen, könnte über die dargestellte Regel zuerst auf Modellebene gezeigt werden, dass bei der Annahme A, einer korrekten Umgebung, alle Teile einzeln P erfüllen, also korrekt implementiert sind. Ist im Anschluss für alle Teilkomponenten die Annahme A der korrekten Umgebung gegeben, folgt, dass auch P gegeben ist.

2.2 Statische Analysen

Während verschiedener Phasen des Entwicklungsprozesses lassen sich Statische Analysen ausführen. Eine Statische Analyse führt im Gegensatz zu einer Dynamischen Analyse das Programm nicht aus, sondern analysiert das vorliegende System nur anhand seines Modells, seines Quelltextes oder dessen Kompilat, je nachdem, ob sie auf Architektur- oder Quelltextebene stattfindet. Vorteilhaft hieran ist, dass Dynamische Analysen erst auf einem fertigen, in seiner Umgebung aufgesetzten System ausgeführt werden können und Statische Analysen bereits vorher.

Beispiele auf Architekturebene sind die Access Analysis for PCM von Kramer et al. [5] und UMLSec von Jürjens et al. [4]. Die Access Analysis wird in Unterabschnitt 2.2.1 kurz vorgestellt, da sie im iterativen Ansatz als Architekturanalyse für die Überprüfung von Zugriffen genutzt wird.

Auf Quelltextebene werden Analysen in Informationsfluss- und Datenflussanalysen unterteilt. Ein Informationsfluss liegt vor, wenn anhand des Programmablaufs Informationen über vertrauliche Systemeingaben an Parameter oder Variablen gelangen, die weniger vertraulich klassifiziert sind und die sensiblen Informationen preisgeben können. Beispiele für Informationsflussanalysen sind JOANA [37], JRIF [38] und das KeY-Framework [39]. Da JOANA in dieser Arbeit als Informationsflussanalyse genutzt wird, wird das Werkzeug in Unterabschnitt 2.2.2 genauer beschrieben.

Im Gegensatz zu Informationsflüssen reicht es bei einem Datenfluss nicht aus, dass über den Ablauf des Programms ausgesagt werden kann, ob vertrauliche Informationen als Eingabe genutzt werden oder nicht. Für einen Datenfluss müssen die Daten selbst, die als vertrauliche Eingabe klassifiziert sind, bis an eine unvertrauliche Ausgabe weitertransportiert werden. Sobald die Daten im Verlauf einer Programmausführung in andere Formate oder Typen umgewandelt werden, liegen sie nicht mehr im Original vor und ihre Informationen, die in die neuen Objekte geflossen sind, gelten nicht mehr als Datenfluss. Trotzdem können auch aus den neu erzeugten Objekten Informationen über ihre Vertraulichkeit abgeleitet werden. Eine Datenflussanalyse deckt deshalb nur eine Teilmenge der Flüsse ab, die durch eine Informationsflussanalyse gefunden werden.

Als spezielle Form der Datenflussanalysen existieren sogenannte Taint-Flow-Analysen. Bei diesen werden Eingabedaten als vertraulich markiert und bei einer Umformatierung eines Datums auch entstandene, abhängige Objekte als vertraulich markiert, bis es irgendwann zu einer Weitergabe an eine unvertrauliche Ausgabe kommt und eine Verletzung vorliegt. Zusätzlich zu reinen Datenflüssen werden damit auch Flüsse sichtbar, die durch das Umwandeln von Daten von einer normalen Datenflussanalyse nicht mehr gefunden werden können. Die Taint-Flow-Analyse CodeQL [40] wird für diese Masterarbeit genutzt und deshalb in Unterabschnitt 2.2.3 beschrieben. Da eine Unterscheidung von echten Datenflüssen und Taints im Kontext dieser Arbeit nicht weiter relevant ist, werden *Taint Flows* mit in den Begriff der Datenflüsse einbezogen und nicht weiter separat getrennt bezeichnet.

Es gibt auch Ansätze, die statische Analysen auf Architektur- und Quelltextebene verbinden, wie z. B. IFlow [8] oder der Kopplungsansatz Härings [9]. Diese werden in den verwandten Arbeiten in Kapitel 3 diskutiert.

2.2.1 Access Analysis

Die Access Analysis ist eine Architekturanalyse, die das Projekt Confidentiality4CBSE [41] nutzt. Dieses Projekt spezifiziert Vertraulichkeit in komponentenbasierten Systemen. Hierfür wird das Palladio-Komponentenmodell (PCM) [13] erweitert, um ein Modell mit Sicherheitsspezifikationen anzureichern. Die Erweiterung enthält zwei Modelle, ein Vertraulichkeitsmodell und ein Angreifermodell [5]. Das Vertraulichkeitsmodell speichert Datenmengen als Sicherheitslevel. Diese Sicherheitslevel können an Parameter und Orte (Locations) des PCM annotiert werden. Im Angreifermodell werden Akteure definiert, die mit dem System interagieren. Ihnen wird ebenfalls eine Datenmenge zugewiesen, auf die sie erlaubten Zugriff besitzen. Das Ergebnis der Access Analysis ist ein Beweisbaum über die verletzte Vertraulichkeit. In ihm befinden sich Interaktionen aus dem Angreifermodell, die die Sicherheitslevel aus dem Vertraulichkeitsmodell verletzen.

2.2.2 JOANA

JOANA ist eine Informationsflussanalyse auf Java-Quelltext, die den kompilierten Quelltext nach festen Mustern abarbeitet. In diesem Quelltext müssen für die Analyse Einstiegspunkte, Quellen und Senken annotiert werden. Einstiegspunkte sind hierbei Startpunkte der Analyse und besitzen einen Sicherheitsverband mit erlaubten Informationsflüssen. Quellen und Senken besitzen Sicherheitslevel. Während der Analyse wird von einem Einstiegspunkt aus ein Aufrufbaum erstellt, in dem JOANA nach invaliden Informationsflüssen sucht. Invalide Informationsflüsse sind diejenigen Flüsse von einer Quelle zu einer Senke, bei der das Quellensicherheitslevel im Sicherheitsverband über dem Senkensicherheitslevel steht, also ein Fluss von hoch zu niedrig stattfindet.

2.2.3 CodeQL

CodeQL ist eine Analyseplattform zur Automatisierung von Sicherheitsprüfungen. Hierzu werden aus dem vorliegenden Quelltext relationale Repräsentationen, Abhängigkeiten, Quelltextteile und sprachenabhängige Relationsschemas in eine Datenbank extrahiert, der sogenannten CodeQL-database. Diese Einträge werden als Fakten bezeichnet. Über eine objektorientierte Abfragensprache, der *queryLanguage*, kann die Untersuchung von Sicherheitsanforderungen als Abfragen auf dieser Datenbank ausgeführt werden. Basierend auf der Datenbank wird ein Datenflussgraph generiert und eine Abfrage wird lokal auf Methoden und global auf allen Datenflüssen untersucht [40].

In einer Abfrage werden zu untersuchende Abhängigkeiten aus definierten Prädikaten selektiert, wie in Listing 2.1 für einen Ausschnitt einer Abfrage von erlaubten Flüssen dargestellt ist. Diese Prädikate werden während der Ausführung der Abfrage mit Wertetupeln aus der aufgebauten Datenbank als *PathNodes* instanziiert. Wenn diese Tupel die Prädikate erfüllen, können deren Rückgabewerte in weiteren Prädikaten und Formeln getestet werden, bis eine Auswertung des gesamten Abfragebefehls erfolgt ist.

```
from MyTaintFlow::PathNode source, MyTaintFlow::PathNode sink
where MyTaintFlow::flowPath(source, sink) and source != sink and
      not isFlowAllowed(source.getNode(), sink.getNode()) and
      notEqualElements(source.getNode(), sink.getNode())
select sink.getNode(), source, sink, printResult(source, sink)


predicate isFlowAllowed(DataFlow::Node source, DataFlow::Node sink){
   allowedFlows(getSourceLabel(source), getSinkLabel(sink))
}

predicate allowedFlows(SecurityLevel source, SecurityLevel sink){
   source = AirlineSupport() and sink = Support() or
   source = AirlineSupport() and sink = Airline() or
      [...]
   source = AirlineTravelAgencyUser() and sink = AirlineUser()
   or getLevelAsString(source) = getLevelAsString(sink)
   or exists(SecurityLevel l | allowedFlows(source, l) and allowedFlows(l, sink))
   or none()
}
```

Listing 2.1: Quelltextauszug aus einer CodeQL-Abfrage

2.3 Analyse-Framework

Zur automatisierten Ausführung einer statischen Quelltextanalyse wird in dieser Arbeit das Projekt *Architecture-And-StaticCode-Analyses-CouplingFramework* genutzt. Das Projekt beinhaltet ein Framework, das Pfad- und Dateiverwaltung übernimmt und einen gesamten Analysedurchlauf und dessen Rückprojektion der Ergebnisse auf Knopfdruck ermöglicht. Hierzu führt das Framework folgende fünf Schritte hintereinander aus:

- ALIGNMENT: Generiere Quelltext zu einem Architekturmodell
- COMPLETION: Annotiere Sicherheitsinformationen für die Analyse
- ANALYSIS: Führe ein Analysetool/Skript aus, z. B. JOANA, CodeQL
- RESOLUTION: Berechne aus den Analyseergebnissen neue Sicherheitsanforderungen
- INTEGRATION: Projiziere die neuen Sicherheitsaspekte zurück ins Architekturmodell

Im Folgenden wird die Architektur des Frameworks beschrieben, um eine Grundlage für die spätere Implementierung der Erweiterungen des eigenen Ansatzes zu bieten. Eine Übersicht ist als Klassendiagramm in Abbildung 2.5 dargestellt.

Das oberste Element des Frameworks ist die Klasse *AnalysisCouplingFramework*. Über die Methode *run(configuration)* kann eine bestimmte *Configuration* übergeben werden, wodurch für diese eine *Session* erzeugt und gestartet wird. Eine *Configuration* speichert alle notwendigen Daten. Sie besitzt eine *UUID* als Identifizierungsnummer. Sie besitzt weiterhin den nächsten zu startenden *Entrypoint* zu einem der fünf bereits aufgezählten Ausführungsschritte. Als Drittes enthält sie alle Pfade, die in den fünf Schritten benötigt werden. Sie wird in einer Konfigurationsdatei im XML-Format persistiert und kann aus dieser erneut geladen werden. Mit einer *Configuration* lassen sich *Sessions* erzeugen, die für die Durchführung der fünf Schritte verantwortlich sind. Eine *Session* speichert hierfür einen *Entrypoint*, der angibt, welcher der fünf Schritte als Nächstes ausgeführt werden soll. Als Zweites enthält sie eine *ProcessingStrategy*, die die feste Ausführungsreihenfolge der Schritte erzwingt und die Instanzen der Schritte enthält. Die Schritte müssen die abstrakte Klasse *Processingstep* implementieren. In dieser wird die Logik bezüglich des Imports von Daten aus dem vorherigen *ProcessingStep* und des Exports der Ergebnisse zum nachfolgenden *ProcessingStep* gekapselt. Wie Abbildung 2.5 im unteren Bereich darstellt, wird für jeden der fünf Schritte eine eigene abstrakte Klasse vorgeschrieben, die es für eine Analyse zu implementieren gilt. Zusätzlich muss zu jedem *ProcessingStep* ein Adapter geschrieben werden, der auf die Skripte oder Projekte zugeschnitten ist, die im jeweiligen Schritt ausgeführt werden sollen. Diese Adapter müssen das Interface *ExecutableProcessingstepAdapter* implementieren.

Auf der rechten Seite von Abbildung 2.5 ist dargestellt, wie das Framework für eine konkrete statische Analyse genutzt wird, in diesem Falle für JOANA [37]. Zuerst werden im unteren Bereich die fünf *ProcessingSteps* und ihre zugehörigen *Adapter* implementiert. Als Nächstes wird eine *JoanaProcessingStrategy* erstellt, die die eigenen Klassentypen für die Ausführung bereitstellt. Auf oberster Ebene bündelt die *JoanaConfiguration* alle JOANA-spezifischen Abhängigkeiten, die in einer Datei im XML-Format gespeichert werden. Es wird ein *Runner* implementiert, der das *AnalysisCouplingFramework* mit der *JoanaConfiguration* startet, indem er die genannte Methode *run(configuration)* aufruft.

Es wird außerdem über einen Pausierungsmechanismus die Möglichkeit angeboten, einen der fünf Schritte manuell auszuführen. Der Mechanismus muss explizit in die Rückgabe eines *ProcessingSteps* als *WaitForManualActionResult* eingebaut werden. Dieses löst in der internen Ausführung der Session eine Unterbrechung aus und es wird über die Konsole auf eine Freigabe zur Fortsetzung gewartet. Sollte außerdem einer der fünf Schritte für eine statische Analyse nicht notwendig sein, wie es für JOANA der Schritt RESOLUTION ist, kann der entsprechende Adapter durch einen DummyAdapter ersetzt werden, wie es Abbildung 2.5 zeigt.

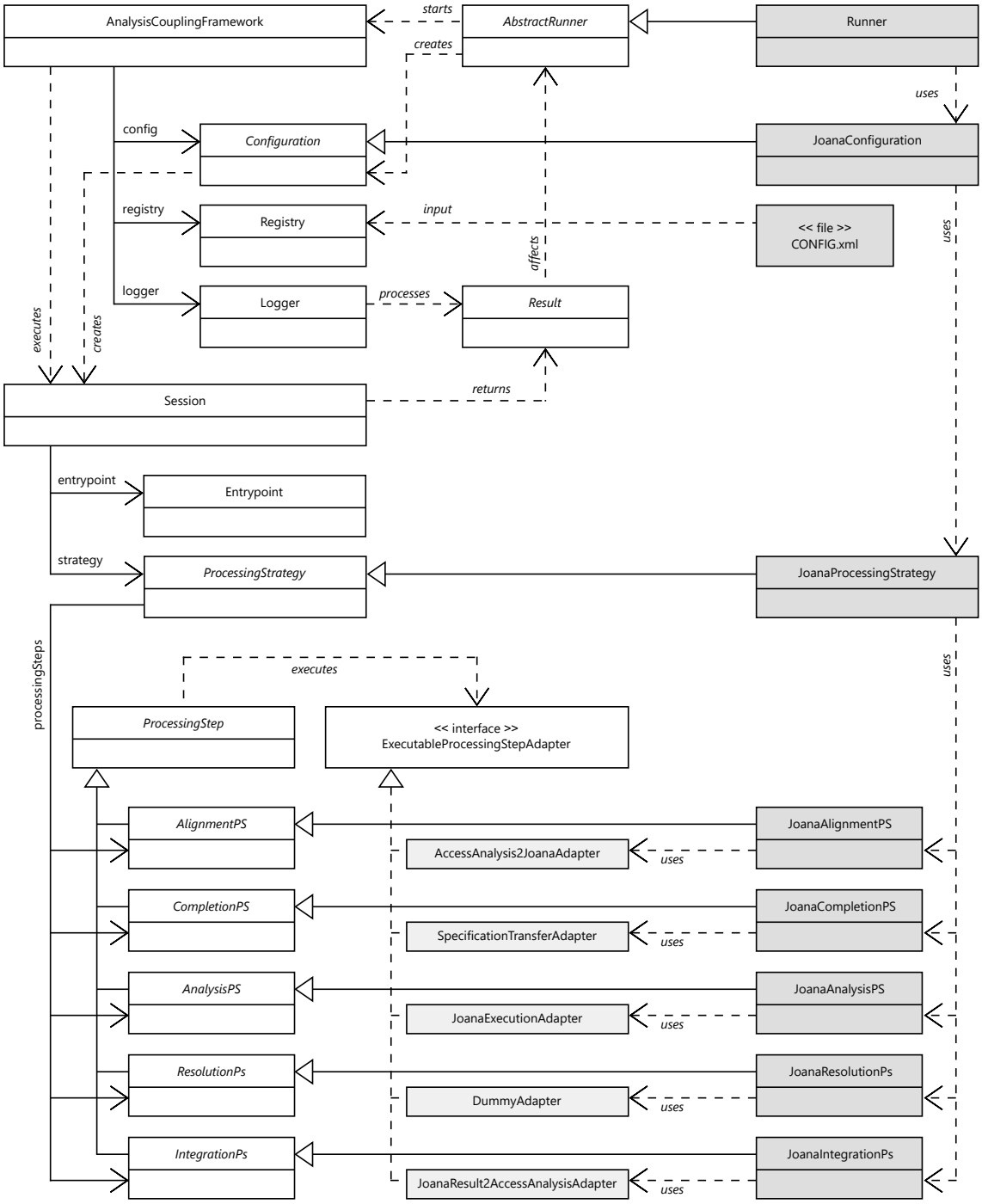


Abbildung 2.5: Klassendiagramm des Analyseframeworks

3 Verwandte Arbeiten

Dieses Kapitel diskutiert verwandte Arbeiten. Als Erstes sind für diese Arbeit Ansätze relevant, die mehrere Sichtweisen auf ein System koppeln und Informationen zwischen diesen Sichten durch Projektionen austauschen. Solche Kopplungsansätze werden in Abschnitt 3.1 beschrieben. Bei der Untersuchung der Sicherheit eines Systems gibt es viele Ansätze, die sich mit der Modellierung und Analyse von Vertraulichkeit beschäftigen. Diese werden in Abschnitt 3.2 beschrieben. Dort wird auch die Einordnung von Analyseansätzen in Informationsflusskontrolle, Datenflusskontrolle und Zugriffskontrolle vorgenommen. Zuletzt werden in Abschnitt 3.3 Arbeiten aus dem Bereich der Sicherheitsverbände beschrieben.

3.1 Kopplungsansätze

Kopplungsansätze existieren in verschiedenen Domänen. Häring [9] zeigt in einem Ansatz der Vertraulichkeitsdomäne zur Kopplung von Architektur- und Quelltextanalyse, dass ein Informationstransfer zwischen den beiden Ebenen möglich ist. In seiner Arbeit wird ein Quelltextmodell aus den Architekturspezifikationen generiert und darauf das Informationsflussanalysewerkzeug JOANA [37] angewandt. Dieses findet invalide Informationsflüsse, die zur Berechnung neuer Sicherheitslevel für die betroffenen Bereiche auf Architekturebene genutzt werden. Durch eine Rückprojektion dieser neuen Sicherheitslevel in das Architekturmodell, erzielt eine Architekturanalyse mehr Aussagen über Vertraulichkeitsverletzungen. Häring erörtert in seiner Arbeit, dass die gefundenen Level für die Rückprojektion nur approximiert werden und nicht optimal sind. Es entstehen dadurch aktualisierte Vertraulichkeitsannotationen, die mehr DataSets enthalten als notwendig und so die Genauigkeit verfälschen. Härings Kopplungsansatz der Quelltext- und Architekturanalyse im Bereich Vertraulichkeit stellt die Grundlage für diese Masterarbeit dar. Von dem eigenen Ansatz grenzt sich jedoch ab, dass Häring die Kopplung nur einmalig ausführt, während der entwickelte iterative Ansatz eine mehrmalige Projektion und Rückprojektion durchführt.

Eine weitere verwandte Arbeit im Bereich der Analysenkopplung ist ein Ansatz von Cortelessa et al. [42]. Sie führen eine modellgetriebene Refactoring-Methode in der Domäne der Systemverfügbarkeit durch. Durch eine Analyse von abgeleiteten Petrinetzen können bekannte Ausfalltoleranzmuster in das Architekturmodell zurückprojiziert werden. Interessant ist für diese Masterarbeit, dass Cortelessa et al. diese Verfügbarkeitsanalyse iterativ umsetzen und solange neue Muster zurückprojizieren, bis das Modell ausreichende Verfügbarkeit erreicht. Der Iterationsmechanismus kann, obwohl er für eine andere Sicherheitsdomäne angewendet wird, mit dem eigenen Iterationsmechanismus verglichen werden.

Zur Abgrenzung fällt die Arbeit Cortelessas et al. jedoch in die Domäne der Ausfallsicherheit (Reliability).

Peldszus entwickelt in einer Arbeit [7] das Werkzeug GRaViTY [43], mit dem es möglich ist, mehrere Modelle aus verschiedenen Abstraktionsschichten automatisch zu synchronisieren. Er zeigt, dass sich Sicherheitsinformationen aus Architektur- und Quelltextebene kombinieren lassen und erzeugt eine gesamtheitliche Repräsentation des Systems. Zwar ist es auch das Ziel dieser Masterarbeit, die Informationen verschiedener Ebenen zu koppeln, aber zur Abgrenzung wird kein gemeinsames, konsistentes Modell angestrebt. Es werden stattdessen Analysen auf beiden Ebenen miteinander verbunden und nur in eine Richtung zurückprojiziert.

Der Ansatz iObserve von Heinrich et al. [44] untersucht im Kontext von Software-Evolution das Aktualisieren von PCM-Elementen auf Architektursicht, sobald sich in der ausgeführten Laufzeitinstanz des Systems neue Erkenntnisse ergeben. Wie auch in dieser Masterarbeit wird eine Rückprojektion über ein Korrespondenzmodell durchgeführt und so eine Kopplung mehrerer Ebenen erzeugt. Diese Rückprojektion ist bei Heinrich et al. jedoch auf das PCM beschränkt und z. B. nicht auf Erweiterungen der Vertraulichkeit.

3.2 Vertraulichkeitsmodellierung und Analyse

Beim Ansatz *IFlow* von Katkalov et al. [8] wird ein UML-Modell mit Sicherheitsannotationen, die den Informationsfluss einschränken, zur Analyse genutzt. Zuerst wird die Architektursicht formal verifiziert. Als Nächstes wird Java-Quelltext in ein Quelltextskelett generiert [45]. Dieses Skelett wird manuell implementiert. Der finale Quelltext kann dann mit Informationsflussanalysen wie *JIF* oder *JOANA* auf eine korrekte Implementierung hin untersucht werden. Eine Rückprojektion von Analyseergebnissen in die Architektursicht liegt nicht vor.

Seifermann et al. [3] entwickeln eine Methode, um in Datenflussdiagrammen sowohl Informationsflusskontrolle als aus Zugriffskontrolle kombiniert auszudrücken. Hierdurch werden Inkonsistenzen vermieden und die Aussagekraft des Modells erhöht. Die Autoren zeigen, wie diese Technik potenzielle Verstöße frühzeitig im Entwicklungsprozess identifiziert, um die Sicherheit und Einhaltung von Datenschutzvorgaben zu verbessern.

Tuma et al. [23] präsentieren eine Methode zur Identifikation von Designfehlern in Software durch die Analyse von Datenflüssen. Die Autoren demonstrieren, wie diese Methode helfen kann, sicherheitsrelevante Schwachstellen bezüglich Integrität und Vertraulichkeit frühzeitig im Entwurfsprozess zu entdecken und zu beheben.

Shostack behandelt in seinem Buch *Threat Modeling: Designing for Security* [46] umfassend die Konzepte und Methoden des Threat Modeling, eine Technik zur Identifizierung und Bewertung potenzieller Bedrohungen in der Softwareentwicklung. Er bietet praktische Anleitungen zur Integration von Sicherheitsüberlegungen in den Entwicklungsprozess und zeigt auf, wie verschiedene Modelle zur Bedrohungsanalyse angewendet werden können, um

Sicherheitslücken frühzeitig zu erkennen. Dieses Werk behandelt auch die Zugriffsanalyse, ist aber zur Abgrenzung von der eigenen Arbeit nur eine theoretische Übersicht über das breite Themengebiet von Bedrohungen für ein System.

Ansätze auf Architekturebene für die Überprüfung der Zugriffskontrolle wurden bereits in Abschnitt 2.2 genannt. Hierunter fallen UMLSec [4] und die Access Analysis von Kramer et al. [5].

In einem Ansatz von Gerking et al. [47] wird Informationsflusssicherheit von Echtzeitsystemen über eine Modellüberprüfung von abgeleiteten Automaten getestet.

Im Bereich der Quelltextanalysen gibt es einerseits Informationsflussanalysen, die untersuchen, ob durch den Programmablauf Rückschlüsse auf vertrauliche Informationen gewonnen werden können. Zu solchen Analysen zählen JOANA [48], JRIF [38] und der *Lattice*-basierte Ansatz *Informationflow Control by Contract* (IFbC) von Runge et al. [49]. Um speziell auch Java Reflections zu untersuchen, existieren Ansätze von Livshits et al. [50] und der Ansatz TamiFlex von Bodden et al. [51]. Das KeY-Framework [39] ist ein vollumfängliches Verifikationswerkzeug auf Quelltextebene, das Yurchenko et al. [52] in einem Ansatz zur Vertraulichkeitsverifikation nutzen. Der Ansatz SUSI von Rasthofer et al. [53] nutzt maschinelles Lernen, um in Android-Quelltext Quellen und Senken von Flüssen zu finden, die für die Untersuchung der Privatsphäre wichtig sind.

Als reine Datenflussanalysen existieren die Analysen Flow Twist von Lerch et al. [54] für große Java-Bibliotheken und die Taintflow-Analyse CodeQL [40], um Flüsse von Nutzereingaben durch das System als sogenannte *Taints* zu identifizieren. Auch der Ansatz FlowDroid von Arzt et al. [55] setzt Taint-Analysen auf Quelltextebene um.

Staicu et al. [56] kommen in einer empirischen Studie zu dem Ergebnis, dass für die meisten Sicherheitsprobleme leichtgewichtige Taint-Analysen ausreichend sind. Die aufgezählten Analyseansätze unterscheiden sich von dieser Masterarbeit, da sie sich mit einzelnen Analysen beschäftigen und sich nicht mit der Kopplung von mehreren Abstraktionsleveln befassen. Als Teilschritte werden solche Analysen in der Arbeit jedoch genutzt. Insbesondere wird für die Evaluation einerseits die schwergewichtige Informationsflussanalyse JOANA genutzt und im Vergleich dazu andererseits die Taint-Analyse CodeQL.

3.3 Sicherheitsverbände für Vertraulichkeit

Die Nutzung von Sicherheitsverbänden (Security Lattices) zur Wahrung der Vertraulichkeit in Informationssystemen ist ein gut erforschtes Gebiet, das wichtige theoretische und praktische Grundlagen bietet. Diese Konzepte sind relevant für die Darstellung von Sicherheitseigenschaften in dieser Arbeit und für die Kopplung von Quelltext- und Architekturanalysen zur Verknüpfung von Vertraulichkeitsergebnissen.

Dennings Werk *A Lattice Model of Secure Information Flow* [24] stellt das *Lattice*-Modell vor, das als Grundlage für die formale Definition und Verwaltung von Zugriffsrechten

und Vertraulichkeitsstufen dient. In ihrem Buch *Cryptography and Data Security* [57] vertieft Denning diese Konzepte und bietet eine umfassende Darstellung der Prinzipien der Datensicherheit, einschließlich der Anwendung von Sicherheitsverbänden.

Das Bell-LaPadula-Modell [25] nutzt ebenfalls ein Verbands-Modell zur Durchsetzung von Vertraulichkeitsrichtlinien. Es beschreibt, wie Systeme sicher gestaltet werden können, um unautorisierten Informationsfluss zu verhindern. Dieses Modell ist besonders einflussreich für die Entwicklung sicherer Softwarearchitekturen.

Sandhu erweitert diese Ideen in seinem Werk *Lattice-Based Access Control Models* [58], indem er verschiedene auf Lattices basierende Zugriffssteuerungsmodelle untersucht und deren Anwendung auf die Vertraulichkeit diskutiert. Diese Modelle bieten wertvolle Einblicke für die Integration von Sicherheitsanforderungen sowohl in der Quelltextanalyse als auch in der Architekturanalyse.

McLean liefert in einem Kommentar [59] eine kritische Analyse des Bell-LaPadula-Modells und diskutiert die Implikationen von Sicherheitsverbänden für die Vertraulichkeit. McLeans Arbeit betont die Bedeutung der theoretischen Fundierung und die Herausforderungen bei der praktischen Anwendung von Sicherheitsverbänden.

Diese grundlegenden Arbeiten bilden das theoretische Fundament für moderne Ansätze zur Verknüpfung von Vertraulichkeitsergebnissen aus Quelltext- und Architekturanalysen. Durch die Kombination dieser beiden Analyseebenen kann eine ganzheitlichere Sicht auf die Vertraulichkeit in Software-Systemen erreicht werden, was zu einer robusteren Sicherheitsarchitektur führt. Im Gegensatz zu diesen beschriebenen theoretischen Werken, werden in der eigenen Arbeit die Konzepte der Verbandsdarstellung von Sicherheitsleveln praktisch angewendet.

4 Fortlaufendes Beispiel

In diesem Kapitel wird ein fortlaufendes Beispiel beschrieben, welches zur Illustration der Konzepte dieser Arbeit dient und ebenfalls in deren Evaluation verwendet wird. Es wird das Reiseplanungssystem *TravelPlanner* von Katkalov et al. [8] verwendet, welches bereits von Seifermann et al. [3] und Häring [9] zur Evaluation herangezogen wurde.

Das System des *TravelPlanners* wird in einer komponentenbasierten Modellierung mit dem Palladio-Komponentenmodell (PCM) (vgl. Unterabschnitt 2.1.1) und mit den Modellerweiterungen von Kramer [5] zur Vertraulichkeitsmodellierung beschrieben. Im Folgenden werden die für diese Arbeit relevanten Sichten beschrieben.

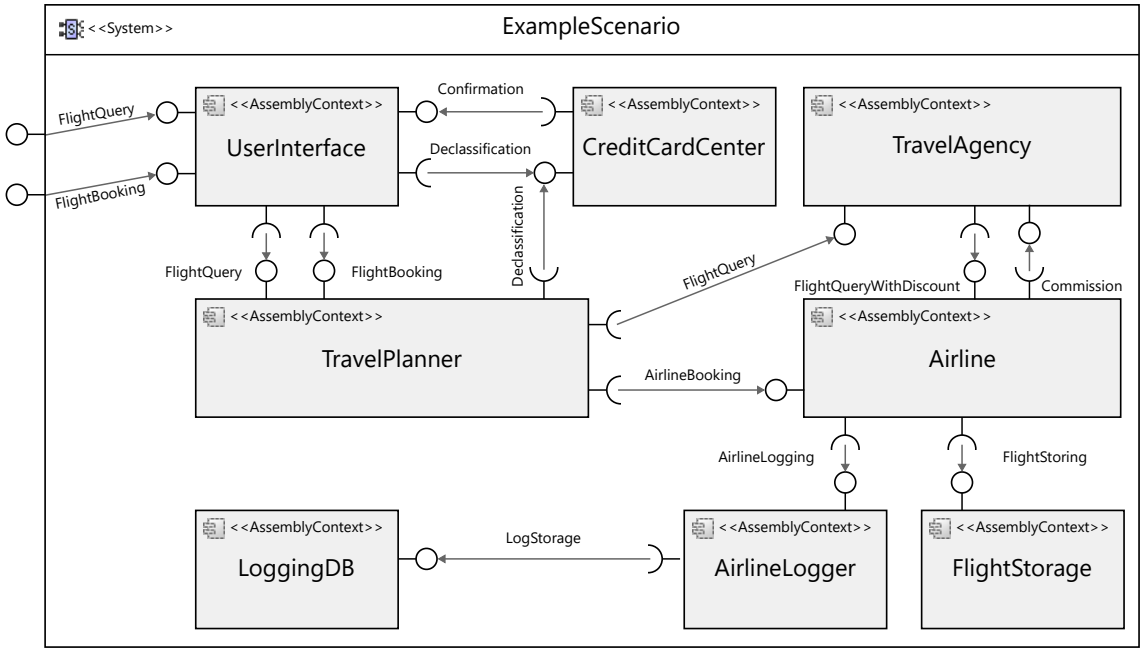


Abbildung 4.1: Systemdiagramm des *TravelPlanners*

Komponentenspezifikation und Architektur Wie in Abbildung 4.1 dargestellt, besteht das *TravelPlanner*-System aus acht Komponenten, die über Schnittstellen miteinander kommunizieren. Ein Nutzer greift über die Komponente *UserInterface* auf das System zu, um einen Flug zu buchen. Die Komponente *TravelPlanner* kommuniziert zuerst mit einer *TravelAgency* um Angebote einzuholen und als Zweites mit der *Airline* um eine Buchung vorzunehmen. Die Komponente *CreditCardCenter* ist dafür zuständig Kreditkarteninformationen für eine Buchung bereitzustellen und diese dafür zu deklassifizieren. Im Schaubild auf unterster

Ebene kommuniziert die *Airline* mit drei Komponenten, die technische Logs anlegen und für die Datenhaltung zuständig sind. Jede Kommunikation zwischen den Komponenten wird über Schnittstellen realisiert, die von einer Komponente angeboten (Provided Interface) und von der anderen benötigt (Required Interface) werden.

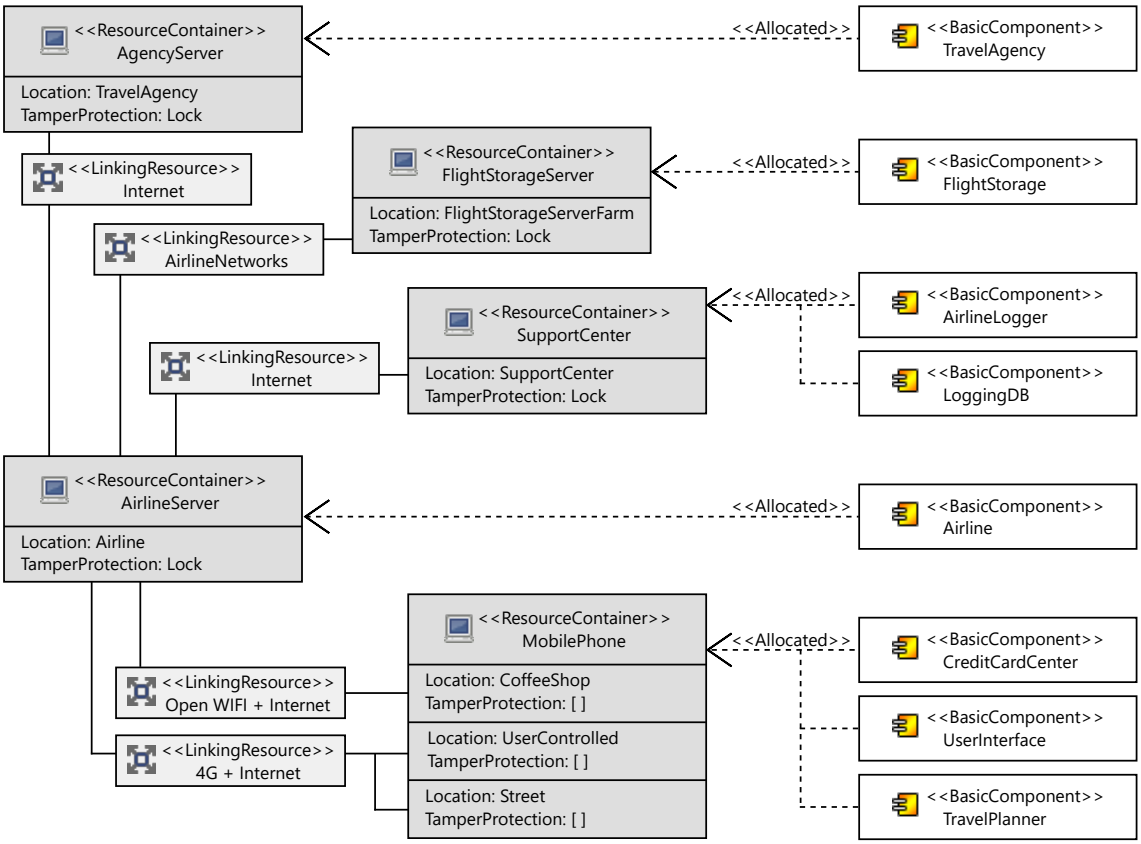


Abbildung 4.2: Ressourcen- und Allokationsdiagramm des *TravelPlanners*

Vorhandene Ressourcen Abbildung 4.2 stellt dar, dass das System verteilt an fünf Standorten liegt. Die dunkelgrauen ResourceContainer stellen die Systemhardware *AgencyServer*, *FlightStorageServer*, *SupportCenter*, *AirlineServer* und *MobilePhone* dar. Der ResourceContainer *MobilePhone* besitzt in der Modellierung drei mögliche Standorte, da ein Nutzer das mobile Gerät an verschiedenen Orten zur Reiseplanung benutzen kann. Die ResourceContainer werden miteinander durch Internetverbindungen verknüpft, die als *LinkingResources* modelliert werden. Auf der rechten Seite in der Abbildung befinden sich die acht Systemkomponenten und es wird über die Allokationspfeile verdeutlicht, auf welchen Ressourcen sie laufen.

Vertraulichkeit des Systems Um Vertraulichkeit zu modellieren, werden DataSets aus dem Confidentiality-Modell für die Darstellung von Sicherheitsleveln genutzt. Außerdem werden die Parteien, die mit dem System interagieren, als Akteure modelliert. Im *TravelPlanner*-Beispiel sind dies *User*, *Airline*, *TravelAgency* und *SupportTechnician*. In Abbildung 4.3 wird gezeigt, welche DataSets die Akteure erlaubterweise sehen dürfen. In der Modellierung sind

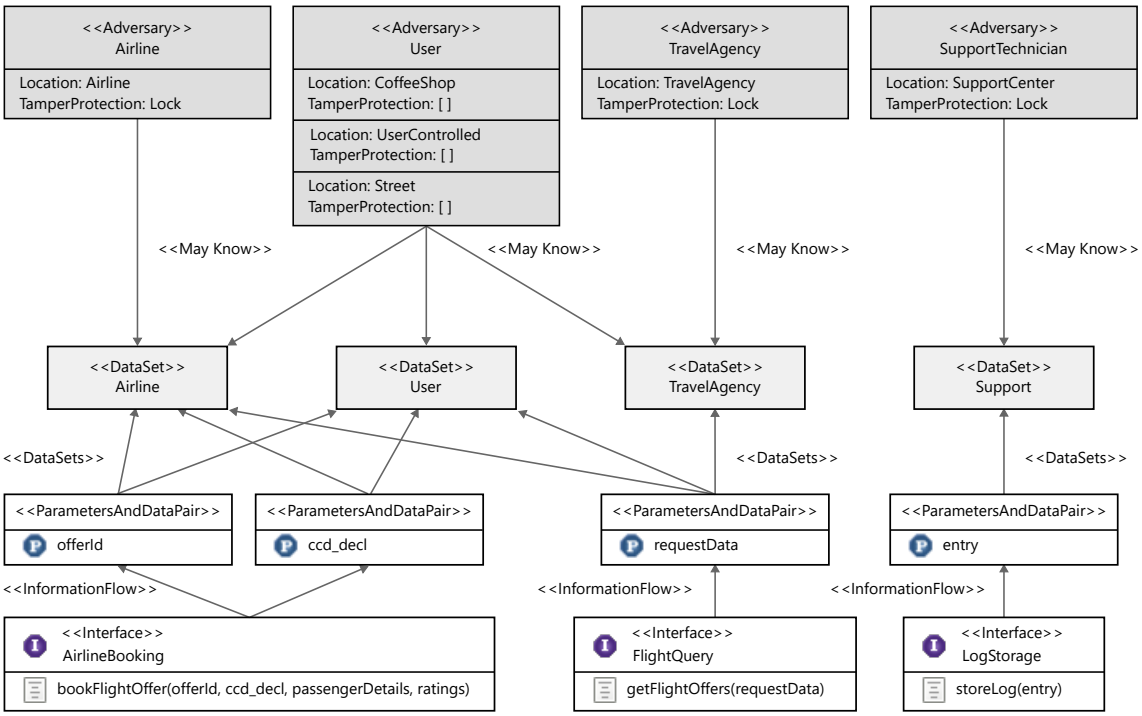


Abbildung 4.3: Zuordnung von Akteuren und Parametern auf DataSets. Die Interfaces sind aus Gründen der Übersichtlichkeit nur für drei Beispiele aufgezählt. Die ParameterAndDataPairs für das Interface *AirlineBooking* sind ebenfalls nicht vollständig.

die DataSets genau nach der Zielgruppe benannt, für die die beinhalteten Daten gedacht sind. Nur der Akteur *User* soll auch DataSets der *Airline* und der *TravelAgency* kennen dürfen, denn deren Informationen sind für die Reiseplanung relevant und sind deshalb auch für einen Nutzer der Reiseplaner-App bestimmt. Wie in Unterabschnitt 2.1.1 beschrieben, lassen sich im Repository-Modell des PCM auch zu Parametern in Methodensignaturen bestimmte DataSets zuweisen. Wie die untere Zeile in Abbildung 4.3 zeigt, erhält z. B. der Parameter *offerId* der Methode *bookFlightOffer()* aus der Schnittstelle *AirlineBooking* die zwei DataSets *Airline* und *User*. Eine vollständige Übersicht aller Parameter und ihrer annotierten DataSets findet sich am Ende dieses Kapitels in Tabelle 4.1. Dort sind die ersten Zeilen dreifach gelistet, da der Parameter *entry* in der korrespondierenden Implementierung überladen wurde und jeweils mit drei verschiedenen Datentypen aufgerufen wird, einmal mit *CreditCardDetails* (CCD), einmal mit *Discountdetails* (DD) und einmal mit Zeichenketten (String). Parameter werden zur Referenzierung über ihre Klasse, ihre Methode und ihren Namen dargestellt.

Erlaubter Zugriff Um für den Verlauf dieser Arbeit das fortlaufende Beispiel auch im Kontext einer Zugriffsanalyse zur Veranschaulichung zu nutzen, werden im Folgenden die Zusammenhänge von erlaubten Zugriffen und den Modellspezifikationen beschrieben.

In Abbildung 4.4 gelangen drei Akteure über ihre modellierte *Location* an den Ort *Support-Center*. An diesem Ort ist der Parameter *entry* sichtbar, dem das DataSet *Support* zugeordnet ist. Diesen Parameter darf ein Akteur nur sehen, wenn ihm ebenfalls das DataSet *Support*

zugewiesen ist. Zusätzlich ist der Ort *SupportCenter* über den Schutzmechanismus (TamperProtection) *Lock* verschlossen. Ein solches Schloss passieren nur diejenigen Akteure, die entweder einen Schlüssel besitzen oder bereit sind, den Schutzmechanismus zu brechen. Beide Fälle zeigen sich dadurch, dass ihnen ebenfalls die TamperProtection *Lock* zugewiesen ist.

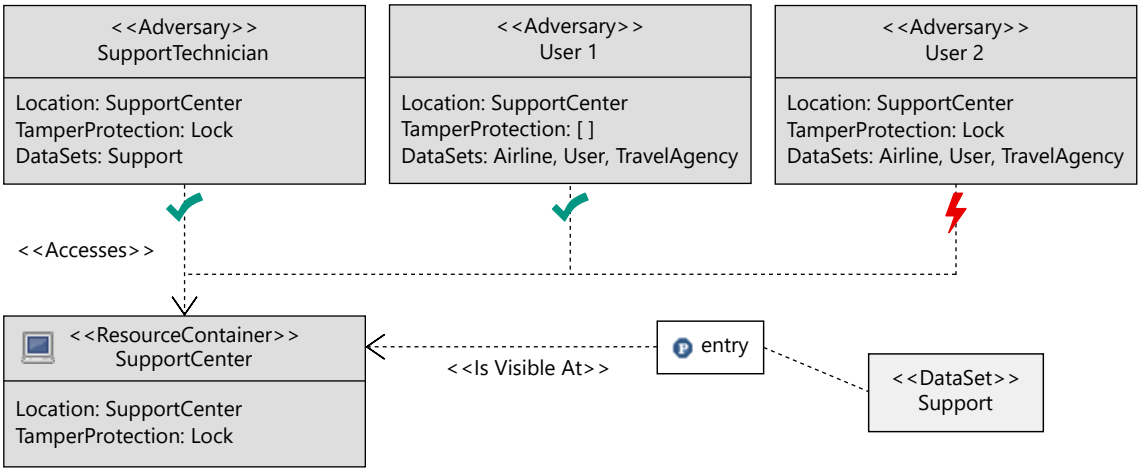


Abbildung 4.4: Einordnung der Zugriffsversuche verschiedener Akteure auf das SupportCenter. Verletzungen sind rot dargestellt, verletzungsfreie Zugriffsversuche grün.

Der erste Akteur *SupportTechnician* besitzt für das *SupportCenter* die notwendige TamperProtection-Annotation und darf das *SupportCenter* betreten. Da er auch den dort anliegenden Parameter *entry* durch seine zugewiesenen DataSets sehen darf, entstehen keine Zugriffverletzungen. Der zweite Akteur *User 1* besitzt nicht die notwendige Annotation, um das *SupportCenter* zu betreten. Dies bedeutet, dass er keinen Schlüssel hat und auch nicht dazu gewillt ist, das Schloss zu brechen. Somit entstehen auch durch ihn keine Zugriffsverletzungen. Der dritte Akteur *User 2* besitzt die *Lock*-Annotation, wodurch er das *SupportCenter* betreten kann. Es wird damit der Fall modelliert, dass ein ortsfremder Akteur durch das Brechen eines Schlosses unberechtigten Zugriff erlangt, da das *SupportCenter* eigentlich nur für Mitarbeiter gedacht ist. Da der Akteur *User 2* in diesem Fall den Parameter *entry* sehen kann, aber seine DataSets nicht das notwendige DataSet *Support* enthalten, liegt eine Verletzung der Vertraulichkeit durch diesen Zugriff vor.

Sobald an einem Ort mehrere Parameter sichtbar werden, müssen diese getrennt auf erlaubte Zugriffe überprüft werden. In Abbildung 4.5 gelangt ein Akteur mit den zugewiesenen DataSets *Support* und *Airline* an einen Ort, an dem zwei Parameter vorliegen. Auf den Parameter *entryCCD* darf er nicht zugreifen, denn dafür wird das DataSet *User* benötigt, welches der Akteur nicht besitzt. Auf den zweiten Parameter *entryDD* darf er zugreifen, denn der Parameter ist unter anderem dem DataSet *Airline* zugeordnet, das auch in den DataSets des Akteurs zu finden ist. Insgesamt kommt es so zu einer Vertraulichkeitsverletzung.

Das dargestellte Zugriffsverhalten aus Abbildung 4.5 stellt den disjunktiven Fall von Sicherheitslevelverknüpfungen dar (vgl. Unterabschnitt 2.1.4). Es reicht aus, mindestens eines der anliegenden Level zu kennen, die in dieser Modellierung als DataSets dargestellt werden, um

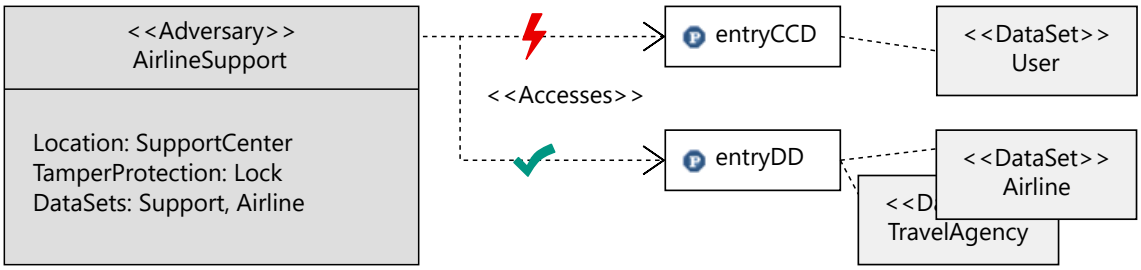


Abbildung 4.5: Disjunktiver Fall: Einordnung von Parameterzugriffen in verletzend (rot) und verletzungsfrei (grün).

einen erlaubten Zugriff auf den Parameter mit diesen Leveln zu erhalten. Da in den Grundlagen dieser Arbeit auch der konjunktive Kontext vorgestellt wurde, wird in Abbildung 4.6 vergleichend dargestellt, welche Verletzungen im konjunktiven Fall entstehen.

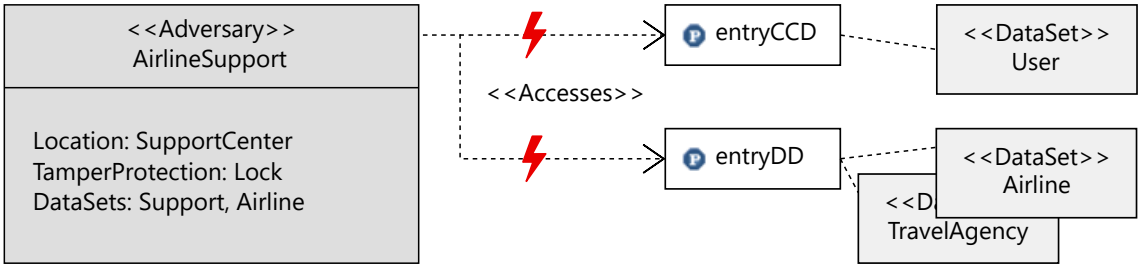


Abbildung 4.6: Konjunktiver Fall: Alle Parameterzugriffe werden als verletzend eingeordnet.

Derselbe Akteur darf im konjunktiven Fall nur auf einen Parameter zugreifen, wenn er alle dort anliegenden Sicherheitslevel selbst besitzt. Für den Zugriff auf den ersten Parameter *entryCCD* fehlt ihm jedoch das DataSet *User* und für den Zugriff auf den zweiten Parameter *entryDD* fehlt ihm das DataSet *TravelAgency*. Aus diesem Grund erzeugen beide Zugriffe eine Vertraulichkeitsverletzung. Es zeigt sich, dass die Interpretation des Modells somit davon abhängt, wie die Verknüpfung der Sicherheitslevel interpretiert wird. Für die Access Analysis von Kramer et al. [5] ist dies z. B. der disjunktive Fall.

Parameter	DataSets
AirlineLogging : log() : $\text{entry}_{\text{CCD}}$	Airline; Support
AirlineLogging : log() : entry_{DD}	Airline; Support
AirlineLogging : log() : $\text{entry}_{\text{String}}$	Airline; Support
LogStorage : storeLog() : $\text{entry}_{\text{CCD}}$	Support
LogStorage : storeLog() : entry_{DD}	Support
LogStorage : storeLog() : $\text{entry}_{\text{String}}$	Support
FlightQuery : getFlightOffers() : requestData	Airline; Travelagency; User
Declassification : releaseCCD() : airlineId	User
DeclassificationConfirmation : confirmRelease() : ccd	User
DeclassificationConfirmation : confirmRelease() : airlineId	User
AirlineBooking : bookFlightOffer() : offerId	Airline; User
AirlineBooking : bookFlightOffer() : ccd_decl	Airline; User
AirlineBooking : bookFlightOffer() : passengerDetails	Airline; User
AirlineBooking : bookFlightOffer() : ratings	Airline; User
FlightQueryWithDiscount : getFlightOffers() : requestData	Airline; Travelagency; User
FlightQueryWithDiscount : getFlightOffers() : discount	Airline; Travelagency
FlightQueryWithDiscount : getFlightOffers() : ratings	Airline; Travelagency
FlightBooking : bookSelected() : flightOffer	Airline; User
FlightStoring : addToFlight() : flightId	Airline
FlightStoring : addToFlight() : details	Airline
FlightStoring : addFlight() : flight	Airline
FlightStoring : retrieveFlights() : airlineId	Airline

Tabelle 4.1: Parameter und ihre zugewiesenen DataSets

5 Iterativer Analyseansatz

Bevor der eigene Ansatz im Detail beschrieben wird, wird ein Überblick über den Ablauf einer Analysenkopplung gegeben. Im Anschluss folgt in Abschnitt 5.1 eine Problemanalyse des ganzheitlichen Ansatzes Härings [9], der eine Kopplung von Architektur und Quelltextebene nur einmalig ausführt. Es werden Ziele abgeleitet, die durch eine iterative Ausführung dieser Kopplung erreicht werden sollen. Im Anschluss wird in Abschnitt 5.2 insbesondere der Aspekt der Genauigkeitsverbesserung erläutert. Es folgt die Vorstellung und Bewertung von verschiedenen Iterationsarten und ihren enthaltenen Mechanismen in Abschnitt 5.3. Zuletzt wird für die Ziele, die bei der Problemanalyse aufgestellt wurden, die Auswahl einer konkreten Iterationsart in Abschnitt 5.4 beschrieben.

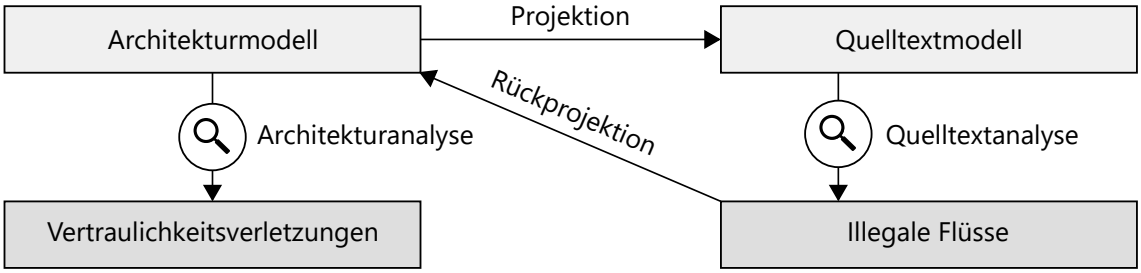


Abbildung 5.1: Ablauf der Analysenkopplung Härings auf Architektur- und Quelltextebene

Aufbauend auf dem Kopplungsansatz Härings [9] soll die Verbindung der Quelltextsicht mit der Architektursicht einen Mehrwert für Vertraulichkeitsaussagen über ein System erzeugen.

Die Kopplung führt hierzu eine Projektion der Vertraulichkeitsinformationen aus einem Architekturmodell des Systems in ein Quelltextmodell des Systems durch, wie in Abbildung 5.1 dargestellt wird. Das Quelltextmodell wird durch eine Informationsflussanalyse auf Flüsse untersucht, die inkonsistent zu den angereicherten Vertraulichkeitsinformationen sind. Die gefundenen inkonsistenten Flüsse werden als illegal markiert und zu einer Anpassung für die Vertraulichkeitsinformationen verarbeitet. Das Ergebnis sind Neuzuweisungen von Vertraulichkeitseigenschaften bestimmter Elemente im System.

Die Kopplung führt anschließend eine Rückprojektion des Ergebnisses in die Architektursicht aus. Hierzu wird ein Korrespondenzmodell verwendet, das ein Element des Quelltextmodells mit einem Element aus der Architektur verbindet. Die berechneten Neuzuweisungen von Vertraulichkeitseigenschaften werden in die Architektur übernommen und können dort über eine Zugriffsanalyse wie die Access Analysis von Kramer [5] ausgewertet werden.

Vertraulichkeitsverletzungen, die bereits durch die Architekturmodellierung auftreten, müssen nicht durch einen Kopplungsansatz aufgedeckt werden, denn sie lassen sich auch ohne Kopplung nur mit der Architekturanalyse finden. Aus diesem Grund wird das ursprüngliche Architekturmodell bezüglich seiner Vertraulichkeitsinformationen konsistent modelliert. Das bedeutet, dass Akteure im Architekturmodell im Ausgangszustand keine Verletzungen bei einer Zugriffsanalyse erzeugen. Durch die Rückprojektion werden die Vertraulichkeitseigenschaften des Architekturmodells verändert, wodurch potentielle Inkonsistenzen entstehen. Diese sind jetzt ausschließlich auf die Kopplung zurückzuführen, da das Modell vorher konsistent war. Verletzungen entstehen, wenn es Akteure gibt, die durch die Inkonsistenzen die Vertraulichkeit beeinträchtigen. Eine Zugriffsanalyse nach der Kopplung deckt solche Verletzungen auf. Die Verbindung beider Sichten erzeugt für die Vertraulichkeitsaussagen somit einen Mehrwert.

5.1 Problemanalyse

Folgende Probleme wurden im ganzheitlichen Ansatz Härings [9] erkannt:

- **(Skalierbarkeit):** Für das Übertragen von erlaubten Flüssen zwischen Sicherheitsleveln zum Quelltextmodell nutzt Häring einen naiven Ansatz zur Verbandsgenerierung. Über die Potenzmenge, die aus einzelnen Eingabeleveln aufgestellt wird, entstehen sehr viele kombinierte Level, die alle in einem Sicherheitsverband zusammengefasst werden. Um die Kanten und Level in einem solchen Verband zu reduzieren, wird in Kapitel 6 dieser Arbeit eine Optimierung der Verbandsgenerierung beschrieben. Es wurde hierbei das Verfahren der Beschränkten Transitiven Reduktion entwickelt, um den Verbandsgraphen auf die Level und ihre Relationen zu beschränken, die im Modell ausgeprägt sind. Sie soll die Skalierbarkeit des Ansatzes verbessern (Ziel 1).
- **(Genauigkeit):** Das zweite Problem seines Ansatzes ist, dass es in einer einzigen Ausführung der Quelltextanalyse vorkommen kann, dass mehrere illegale Flüsse zu einer gleichen Senke fließen. Da seine Quelltextanalyse nur einmalig ausgeführt wird, müssen solche Kollisionen aufgelöst und zu einem einzigen Ergebnis zusammengeführt werden. Härings genutzter Approximationsalgorithmus für dieses Zusammenführen erzielt keine volle Abdeckung der Verletzungen, die die illegalen Flüsse in der Architektur erzeugen sollten, da durch den von ihm genutzten Verknüpfungsoperator eine Unterabschätzung in Kauf genommen wird. Dies wird in Abschnitt 5.2 ausführlich hergeleitet. Es ergibt sich, dass ein iterativer Ansatz derartige Kollisionen vermeiden kann, indem in jeder Iteration nur ein einzelnes Level und dessen Flüsse betrachtet werden. Dadurch kann eine gesamte Abdeckung der erwarteten Verletzungen erreicht und die Genauigkeit des Ansatzes verbessert werden (Ziel 2).
- **(Performanz):** Das dritte Problem seines Ansatzes ist, dass die Analyse der Flüsse von Quellen zu Senken in einem einzigen Analysedurchlauf ausgeführt wird. Dabei entstehen große Dateien als Eingabe für die Quelltextanalyse, da alle Annotationen für Quellen und Senkenkombinationen abgedeckt werden müssen. Es kommt zu einem

hohen Arbeitsspeicherverbrauch, sobald die Quelltextanalyse alle Annotationen auf einmal nutzt. Um die Auslastung zu reduzieren, wird ein iterativer Ansatz als Möglichkeit vorgestellt, die Menge an zu untersuchenden Flüssen auf Iterationen aufzuteilen. Diese Iterationen werden einzeln ausgeführt und sind leichtgewichtiger als die gesamtheitlich ausgeführte Quelltextanalyse im nicht-iterativen Ansatz. Die Partitionierung kann die Gesamtlaufzeit und den maximalen Speicherverbrauch verringern. So soll die Performanz des Ansatzes erhöht werden (Ziel 3).

5.2 Genauigkeitsverbesserung der Rückprojektion

In Unterabschnitt 2.1.4 wurden verschiedene Repräsentationen für Sicherheitslevel unterteilt. Im Folgenden wird für die verschiedenen Darstellungen der Sicherheitslevel beschrieben, wie eine Rückprojektion von illegalen Informationsflüssen durchgeführt werden muss, um die Ergebnisse der Quelltextanalyse in die Architektur zu übertragen.

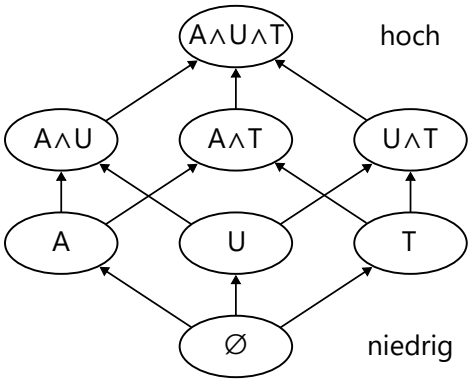
5.2.1 Unverknüpfbare Sicherheitslevel

Bei unverknüpfbaren Sicherheitsleveln gibt es die Klassen *Binär*, *Geordnet* und *Ungeordnet*. In Unterabschnitt 2.1.4 wurde jeweils bereits beschrieben, wann in den Klassen ein illegaler Fluss von einer Quelle zu einer Senke vorliegt. Um einen solchen illegalen Fluss, den die Quelltextanalyse findet, auch ins Architekturmodell zurückzuprojizieren, muss die Informationsflusssenke auf das Sicherheitslevel der Quelle gesetzt werden. In der Architektur liegt dann für jeden Akteur, der Zugriff auf die Senke hat, ein aktualisiertes Sicherheitslevel an dieser Senke vor, das inkonsistent zum Ursprungsmodell ist. Der Akteur, der zuvor nur Informationen mit dem Level der Senke sehen konnte, sieht jetzt Informationen mit dem aktualisierten Sicherheitslevel, für die er eventuell nicht zugriffsberechtigt ist. Dadurch kann eine Vertraulichkeitsverletzung erzeugt werden.

Kommt es zu mehrfachen illegalen Flüssen mit verschiedenen Quellenleveln, muss entschieden werden, mit welchem der Quellenlevel eine Aktualisierung des Senkenlevel durchgeführt wird. Im Falle der Klasse *Binär*, mit den Leveln *High* und *Low* tritt ein solcher Fall nicht auf, da es nur zwei verschiedene Klassen gibt und so alle illegalen Flüsse das Quellenlevel *High* haben. Im Falle der Klasse *Geordnet* wird für die Rückprojektion bei mehreren verschiedenen Quellenleveln dasjenige ausgewählt, das in der Ordnung am höchsten vertraulich eingestuft ist. Denn dadurch werden nach Aktualisierung des Architekturmodells die meisten Verletzungen sichtbar. Für die Klasse *Ungeordnet* kann keines der Quellenlevel alleine für die Aktualisierung ausgewählt werden, aber es ist auch nicht möglich die Level zu verknüpfen, da die Klasse aus der Kategorie *Unverknüpfbar* stammt. In der ganzheitlichen Kopplung der Analysen kann nur eines der Level zurückprojiziert werden und es gehen Verletzungen verloren, die durch die anderen Level sichtbar werden könnten. Der iterative Ansatz kann im Vergleich dazu in mehreren Iterationen jeweils eines der Level zurückprojizieren und deckt damit alle potentiellen Verletzungen ab.

5.2.2 Verknüpfbare Sicherheitslevel

Konjunktiv Im konjunktiven Fall stellt ein kombiniertes Sicherheitslevel den Fall dar, dass auf ein Element nur zugegriffen werden darf, wenn ein Akteur alle einzelnen Sicherheitslevel des Elements besitzt oder eine Übermenge davon. Einzelne Sicherheitslevel sind in einem konjunktiven Verband als niedrig und kombinierte Level als höher vertraulich einzuordnen, wie es in Unterabschnitt 2.1.4 beschrieben und links in Abbildung 5.2 dargestellt ist. Ein erlaubter Informationsfluss verläuft deshalb von einzelnen zu kombinierten Leveln. Sobald ein illegaler Informationsfluss entgegen der Verbandsrichtung stattfindet, kann über den minimalsten, gemeinsam erreichbaren Knoten in der Graphstruktur das Supremum der zusammenfließenden Level berechnet werden (vgl. Unterabschnitt 2.1.5). Dieses Supremum kann in die Architektur übertragen werden, um zu kennzeichnen, dass der Zugriff auf ein Element nun als Berechtigung mehr Sicherheitslevel fordert. Dies führt bei gleichbleibenden Berechtigungen der Akteure zu Verletzungen, die so in der Architekturebene sichtbar gemacht werden können. Die konjunktive Verknüpfung macht auch das Berechnen des Supremums möglich, wenn mehrere illegale Flüsse an das gleiche Systemelement gelangen. Denn die Level dieser Flüsse lassen sich als mehrstellige Konjunktion ebenfalls zu einem Supremum vereinen.



Illegaler Fluss	Supremum
A → U	A ∧ U = AU
AU → A	AU ∧ A = AU
AU → T	AU ∧ T = AUT
T → AU	T ∧ AU = AUT
A ↘ / AU ↗ U	A ∧ AU ∧ U = AU

Abbildung 5.2: Links: Ein konjunktiver Verband der Einzellevel Airline (A), User (U) und Travel-Agency (T). Das Symbol ∅ steht für die leere Levelmenge. Rechts: Illegale Flüsse im konjunktiven Fall und das zugehörige Supremum der Level.

Die rechte Seite von Abbildung 5.2 stellt dar, wie das Sicherheitslevel der Senke in der Architektur bei verschiedenen illegalen Flüssen angepasst werden muss. Durch einen illegalen Fluss liegen an der Senke mehrere konjunktive Sicherheitslevel vor, die alle über eine mehrstellige Konjunktion zu einem neuen Sicherheitslevel verknüpft werden, wie es der Supremumsoperator vorgibt. Bei einem Fluss von [User] nach [Airline] muss das Level der Senke des Flusses z. B. auf [User;Airline] gesetzt werden.

Disjunktiv Ein disjunktiv kombiniertes Sicherheitslevel erlaubt genau dann Zugriff auf einen Ort oder eine Information, wenn ein Angreifer mindestens eines der enthaltenen einzelnen Level kennt bzw. sobald der Schnitt der Levelmengen von Ort und Angreifer nicht leer ist. Während erlaubte Flüsse im konjunktiven Fall von der Untermenge zur Obermenge verliefen, laufen erlaubte Flüsse im disjunktiven Fall von der Obermenge zur

Untermenge, wie es der Verband in Abbildung 5.3 darstellt. Denn die Obermenge aus verknüpften Sicherheitsleveln ist niedriger vertraulich als die Untermenge aus einzelnen Sicherheitsleveln, da mehr Parteien berechtigt sind, darauf zuzugreifen. Das CANUKUS-Beispiel in Unterabschnitt 2.1.3 veranschaulicht dies. Im Unterschied zur Konjunktion ist die Projektion von Verletzungen, die durch illegale Flüsse entstehen, zurück in die Architektur nicht direkt über ein Supremum oder Infimum ablesbar, wie im Folgenden abgeleitet wird.

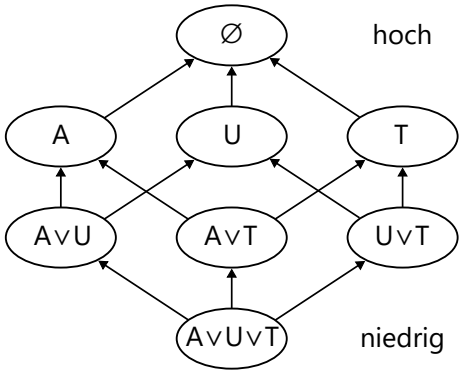


Abbildung 5.3: Ein disjunktiver Verband der Einzellevel Airline (A), User (U) und TravelAgency (T)

Hierfür veranschaulicht Abbildung 5.4 die Bedeutung eines illegalen Informationsflusses. Vor dem Fluss ist nur einer der sieben Akteure unberechtigt, auf die Senke zuzugreifen. Nach dem Fluss sind drei der sieben Akteure unberechtigt, auf die gemeinsame Informationsmenge zuzugreifen. Denn am Ausgangsort (der Senke) liegen nach dem Fluss sowohl Daten mit dem ursprünglichen kombinierten Sicherheitslevel vor als auch Daten mit dem kombinierten Sicherheitslevel des Zuflusses. Aussagenlogisch bedeutet dies, dass auf die kombinierte Datenmenge nur derjenige zugreifen darf, der sowohl auf das Ursprungslevel als auch auf das Zuflusslevel zugreifen darf. Die beiden Levelmengen müssen demnach über eine Konjunktion (nicht eine Disjunktion) verknüpft werden, wie es im zweiten Teil der Grafik als neues Level der Senke dargestellt ist.

Diese Verknüpfung muss aber als Disjunktion repräsentiert werden, da im disjunktiven Verband einzig die Disjunktion zur Verknüpfung von Leveln erlaubt ist. Eine aussagenlogische Umformung ergibt im Beispiel:

$$\begin{aligned} Senkenlevel_{neu} &= Quellenlevel_{alt} \wedge Senkenlevel_{alt} = A \wedge (A \vee U) \\ &= (A \wedge A) \vee (A \wedge U) \\ &= A \vee (A \wedge U) \\ &= A \end{aligned}$$

Durch Ersetzen des alten Senkenlevels $(A \vee U)$ mit dem neuen Senkenlevel A werden alle Verletzungen in die Architektur übertragen. In diesem Beispiel ist die Darstellung des neuen Levels nur möglich, weil es eine Darstellung des aussagenlogischen Terms gibt, der als mehrstellige Disjunktion repräsentierbar ist. Außerdem dürfen nur positive

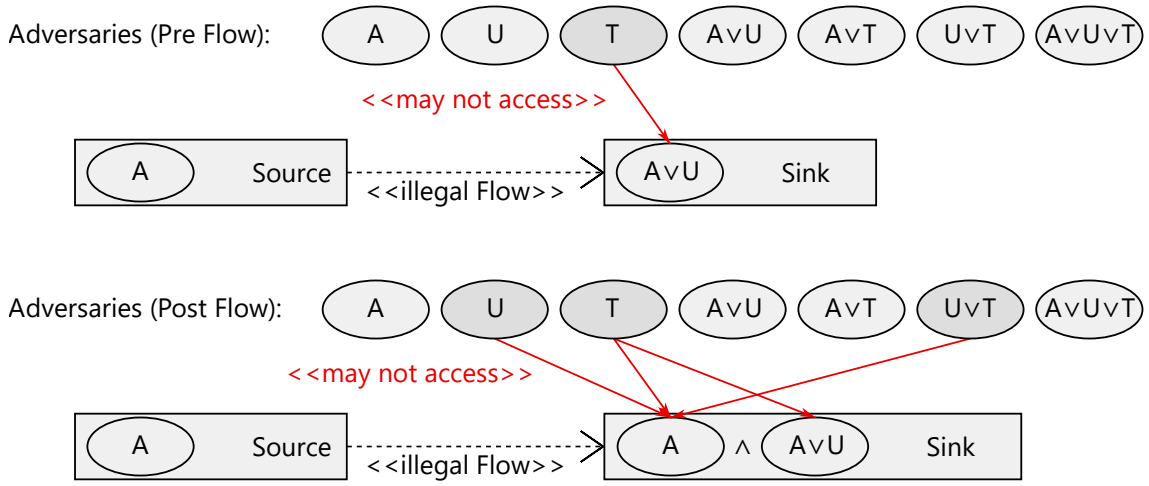


Abbildung 5.4: Verletzungen der Vertraulichkeit beim Zugriff auf das Element Senke vor und nach einem illegalen Informationsfluss. Der illegale Fluss verläuft von einer Quelle mit Level A zu einer Senke mit Level $(A \vee U)$. Akteure werden in berechtigt (grau) und unberechtigt (dunkelgrau) unterteilt.

Literale vorkommen, d. h. es muss auf die Negation verzichtet werden. Die Negation eines einfachen Sicherheitslevels $\neg A$ bedeutet semantisch, dass kein Akteur, der Level A besitzt, auf eine derart annotierte Systemstelle zugreifen darf. Da dazu aber alle Akteure aufgezählt werden müssten, die eine Levelmenge ohne A besitzen, ist eine Negation nicht als ein einziges Level darstellbar und deshalb nicht in der disjunktiven Darstellung erlaubt. Auch die Definition und Nutzung eines neuen Sicherheitslevels mit dem Namen $\neg A$, das genau diese Konstellation repräsentiert, ist nicht erlaubt, da es in diesem Falle die Möglichkeit gäbe, A mit $\neg A$ zu verknüpfen, was semantisch keinen Sinn macht. Im Allgemeinen ist es somit nicht möglich, die benötigte Levelkombination von Quelle und Senke disjunktiv darzustellen, da keine funktional vollständige Junktorenmenge zur Verfügung steht. Z. B. ist es beim illegalen Fluss $A \rightarrow U$ nicht möglich, das konjunktive Supremum $A \wedge U$ als Disjunktion darzustellen. Ein weiteres Gegenbeispiel ist der illegale Fluss $(A \vee U) \rightarrow (A \vee T)$ mit dem Supremum $(A \vee U) \wedge (A \vee T) = A \vee (U \wedge T)$, das sich zwar vereinfachen lässt, aber nicht zu einer vollständigen Disjunktion wird.

Da das Architekturmodell vor der Rückprojektion als konsistent und deshalb verletzungsfrei angenommen wird, wie zu Beginn des Kapitels beschrieben, kann abgeleitet werden, dass das Ursprungslevel keine Verletzung erzeugt hat. Jeder modellierte Akteur hat somit die Berechtigung für den Zugriff auf das Ursprungslevel. Alle Akteure, die schon vor dem Fluss nicht auf das Ursprungslevel zugreifen dürfen, existieren folglich im Modell nicht. Somit bleibt aus der Konjunktion von Ursprungslevel und Zuflusslevel nur noch das Zuflusslevel als Kandidat für eine Architekturrückprojektion und einen Informationsgewinn über neue Verletzungen.

Liegt ein einzelner illegaler Informationsfluss von einer Quelle zu einer Senke vor, ist es somit möglich, die zweistellige Konjunktion beider Levelmengen darüber abzudecken, dass das Architekturmodell im Ausgangszustand bereits den ersten Teil der Konjunktion als

verletzungsfrei belegt. Durch eine anschließende Rückprojektion des Quelllevels wird der zweite Teil der Konjunktion untersucht. Insgesamt ist es so über die zeitlich getrennten Modellinstanzen möglich, alle Aussagen über Verletzungen zu gewinnen. Problematisch wird es, wenn mehrere Quellen unterschiedliche Levelmengen an die gleiche Senke fließen lassen. Denn in einer solchen mehrstelligen Konjunktion entfällt kein weiterer Teil außer dem Ursprungslevel, sodass die Konjunktion bestehen bleibt und sich nicht über eine Disjunktion darstellen lässt. Als Beispiel müsste für die illegalen Flüsse von $A \rightarrow U$ und $T \rightarrow U$ das Level $(A \wedge T)$ rückprojiziert werden, das sich aber nicht als reine Disjunktion darstellen lässt.

Quelle_1	Quelle_2	Senke	SUP	$\cap$	(p; r)	QD	(p; r)
A	-	U	A	$\varnothing$	(0,50; 1)	A	(1; 1)
A	-	$A \vee U$	A	A	(1; 1)	A	(1; 1)
A	-	$U \vee T$	A	$\varnothing$	(0,50; 1)	A	(1; 1)
$A \vee U$	-	$A \vee T$	$A \vee U$	A	(0,50; 1)	$A \vee U$	(1; 1)
A	U	T	$A \wedge U$	$\varnothing$	(0,75; 1)	$A \vee U$	(1; 0,33)
A	U	$A \vee U$	$A \wedge U$	$\varnothing$	(0,67; 1)	$A \vee U$	(0; 0)
A	U	$A \vee T$	$A \wedge U$	$\varnothing$	(0,67; 1)	$A \vee U$	(1; 0,25)
A	$A \vee U$	$A \vee U \vee T$	A	A	(1; 1)	$A \vee U$	(1; 0,33)
A	$U \vee T$	$A \vee U \vee T$	$A \wedge (U \vee T)$	$\varnothing$	(0,57; 1)	$A \vee U \vee T$	(0; 0)
$A \vee U$	$A \vee T$	$A \vee U \vee T$	$(A \vee U) \wedge (A \vee T)$	A	(0,67; 1)	$A \vee U \vee T$	(0; 0)

Tabelle 5.1: Abschätzungen illegaler Flüsse durch die drei Operatoren Quellensupremum (SUP), Mengenschnitt $\cap$ und Quellendisjunktion (QD). Das Quellensupremum entspricht dem konjunktiven Supremum, jedoch wird das Level der Senke vernachlässigt. Für die fehlerbehafteten Operatoren werden Präzision (Precision) und Sensitivität (Recall) aus dem Vergleich erzeugter neuer Verletzungen durch die sieben möglichen Akteure mit dem erwarteten Ergebnis als Tupel angegeben. Die Werte sind auf zwei Nachkommastellen gerundet. Die Auswahl der dargestellten Flüsse bildet die notwendigen Flussklassen des Verbands aus Abbildung 5.3 ab, mit denen alle illegalen Flüsse durch Umbenennung der Literale erzeugbar sind.

In einem solchen Fall veranschaulicht Tabelle 5.1 mögliche Ansätze, das Sicherheitslevel für die Rückprojektion zu approximieren. Neben dem konjunktiven Supremum, das den korrekten Term der Kombination beschreibt, wird zusätzlich der Mengenschnitt $\cap$ als Operator vorgeschlagen. Dieser beschreibt den ersten gemeinsam erreichbaren Knoten im Graphen des disjunktiven Verbandes ähnlich dem Supremum im konjunktiven Fall. Es werden durch diese Kombination nur noch Akteure zugelassen, die mindestens ein Level besitzen, das zuvor in beiden zu kombinierenden Levelmengen enthalten war. Dies ist eine Überabschätzung, da alle Akteure wegfallen, die zuvor über zwei separate, nicht im Schnitt liegende Teillevel sowohl auf die Quelle als auch die Senke zugreifen konnten. Solche Akteure erzeugen falsch Positive Verletzungen in der Architektur. Ein weiterer Operator zur Kombination ist die vollständige Disjunktion aller einzelnen Level, die aus den Quellen stammen. Akteure werden in dieser Kombination genau dann zugelassen, wenn sie zuvor Zugriff auf mindestens eine der Quellen hatten. Dieser Operator liefert bei illegalen Flüssen mit nur einer Quelle einen vollwertigen Informationszuwachs über Verletzungen für die

Architektur, da das Senkenlevel im Ursprungsmodell bereits abgedeckt ist. Sobald mehrere Quellen in Flüssen zu einer Senke beteiligt sind, liegt eine Unterabschätzung vor. Ein Akteur, der zuvor nur auf eine Quelle erlaubten Zugriff hatte, wird nicht als verletzend erkannt, obwohl nun Daten für ihn sichtbar werden, die aus Quellen stammen, deren Level ihn zuvor abhielten. Solche Akteure stellen falsch Negative (nicht gefundene Verletzungen) in der Architekturanalyse dar.

Um die Genauigkeit der Operatoren einzuschätzen, werden die Metriken Sensitivität und Präzision herangezogen. Die Sensitivität stellt die Abdeckung der Verletzungen dar und zeigt im Falle des Quellendisjunktionsoperators die Unterabschätzung in drei der Zeilen, in denen sie nur 0,33 erreicht. Die Präzision stellt dar, wie viele *Falsche Alarme* ausgelöst werden und zeigt im Falle des Schnittoperators $\cap$ die Überabschätzung, da in den meisten Zeilen keine Präzision von 1,0 erreicht wird.

Zur Verdeutlichung, wie Präzision und Sensitivität in Tabelle 5.1 berechnet wurden, dient Tabelle 5.2. Über die Auflistung aller Zugriffe der Akteure und erzeugten neuen Verletzungen können die Operatoren mit dem erwarteten Ergebnis verglichen werden. Das erwartete Ergebnis ist durch den Supremumsoperator $\cap$ gegeben. Die Präzision p kann mit den gelisteten Werten über das Verhältnis von korrekt Positiven (TP: true positive) Klassifikationen gegenüber der Summe aus korrekt Positiven und falsch Positiven (FP: false positive) berechnet werden: $p = \frac{TP}{TP+FP}$. Im Beispiel des fünften illegalen Flusses ($A \rightarrow T; U \rightarrow T$) aus Tabelle 5.1 ergibt sich für den Schnittoperator $p_{\cap} = \frac{3}{3+1} = 0{,}75$ und für den Operator der Quellendisjunktion $p_{QD} = \frac{1}{1+0} = 1{,}00$. Die Sensitivität r wird über das Verhältnis an korrekt Positiven gegenüber der Summe an korrekt Positiven und falsch Negativen (FN: false negative) berechnet: $r = \frac{TP}{TP+FN}$. Im Beispiel ergibt sich für den Schnittoperator $r_{\cap} = \frac{3}{3+0} = 1{,}00$ und für den Operator der Quellendisjunktion $r_{QD} = \frac{1}{1+2} \approx 0{,}33$.

Akteure:	A	U	T	$A \vee U$	$A \vee T$	$U \vee T$	$A \vee U \vee T$
Zugriff auf vorherige Senke T:	-	-	✓	-	✓	✓	✓
Zugriff auf SUP = $A \wedge U$:	-	-	-	✓	-	-	✓
Zugriff auf $\cap = \varnothing$:	-	-	-	-	-	-	-
Zugriff auf QD = $A \vee U$:	✓	✓	-	✓	✓	✓	✓
Neue Verletzungen durch SUP:	-	-	X	-	X	X	-
Neue Verletzungen durch $\cap$:	-	-	X (TP)	-	X (TP)	X (TP)	X (FP)
Neue Verletzungen durch QD:	-	-	X (TP)	-	- (FN)	- (FN)	-

Tabelle 5.2: Beispielauflistung der erlaubten Zugriffe und neuen Verletzungen für den fünften illegalen Fluss ($A \rightarrow T; U \rightarrow T$) aus Tabelle 5.1. Die Abkürzungen der drei Operatoren sind aus Tabelle 5.1 übernommen. Eine neue Verletzung entsteht genau dann, wenn der Zugriff auf die vorherige Senke erlaubt war und durch den Operator nicht mehr.

Härings Ansatz [9] berechnet die Rückprojektion über den Quellendisjunktionsoperator und führt somit eine Unterapproximation durch, wie gezeigt wurde. Im iterativen Ansatz können die mehrstelligen Konjunktionen des Supremums in getrennte Modellinstanzen zurückprojiziert werden. Durch die gesammelten Verletzungen aller neuen Modelle werden die illegalen Flüsse vollständig abgedeckt, wie es der Supremumsoperator ausdrückt.

Vollständig Aussagenlogisch Im konjunktiven und disjunktiven Fall stehen zur Verknüpfung von Sicherheitsleveln nur die jeweiligen Junktoren Konjunktion und Disjunktion zur Verfügung. Wie im disjunktiven Fall beschrieben wurde, kann dies zu Problemen bei der Berechnung eines Supremums führen, wenn eine mehrstellige Konjunktion als Disjunktion ausgedrückt werden soll. Erweitert man die verfügbaren Junktoren zu einer funktional vollständigen Junktorenmenge, z. B. zu $\{\vee, \wedge, \neg\}$, lassen sich alle aussagenlogischen Terme und Verknüpfungen von Leveln darin darstellen. Ein solches aussagenlogisches Sicherheitslevel gewährt einem Akteur genau dann Zugriff, wenn alle Literale im aussagenlogischen Term mit den Wahrheitswerten instanziiert werden, die beim Akteur vorliegen und der Term wahr ist. Ein Akteur besitzt eine Liste der positiven Literale, die zu dem logischen Wahrheitswert $1 = wahr$ ausgewertet werden; alle anderen Literale werden mit $0 = falsch$ ausgewertet.

Da es bei n einzelnen Leveln unendlich viele verschiedene aussagenlogische Terme zur Verknüpfung gibt, müssen kombinierte Level in Äquivalenzklassen (ÄK) ihrer Terme aufgeteilt werden. Eine ÄK besteht aus allen Termen, die für alle Akteure und deren Literalwerte auf denselben Wahrheitswert ausgewertet werden. Da es 2^n verschiedene Akteure gibt, gibt es 2^{2^n} verschiedene ÄKs. Ein erlaubter Fluss findet genau dann zwischen einer Quelle und Senke statt, wenn beide Terme der kombinierten Sicherheitslevel von Quelle und Senke aus derselben Äquivalenzklasse stammen. Ein illegaler Fluss liegt vor, wenn sich beide Terme aussagenlogisch nicht ineinander überführen lassen. Eine Architekturmodellierung und auch Analysen auf Architektur- sowie Quelltextebene müssten angepasst werden, um die Nutzung solcher kombinierter Level zu unterstützen. Da die Äquivalenzüberprüfung zweier Terme mit dem Erfüllbarkeitsproblem der Aussagenlogik zusammenhängt, ist die Nutzung aussagenlogisch verknüpfter Sicherheitslevel jedoch nicht sinnvoll.

5.3 Vorstellung der Iterationsarten

Im iterativen Analyseansatz ist es das Ziel, die Kopplung von Architektur- und Quelltextebene wie auch im nicht-iterativen Ansatz Härings [9] für eine Kombination der Vertraulichkeitsaussagen der beiden Ebenen zu nutzen. Es werden Informationsflüsse der Quelltextebene analysiert und Flüsse gefunden, die dem Vertraulichkeitsbegriff der Architektur widersprechen. Diese illegalen Flüsse werden verarbeitet und in die Architektur zurückprojiziert. Bevor nun jedoch die Architekturanalyse ausgeführt wird, um Vertraulichkeitsverletzungen aufzudecken, werden im iterativen Ansatz erst weitere Zyklen des Projektions- und Rückprojektionskreislaufs ausgeführt. Diese Durchläufe des Projektionszyklus werden Iterationen genannt. In einer einzelnen Iteration wird nur ein bestimmter Teil des Systems, der Modellelemente oder anderer aufteilbarer Elemente, z. B. aus der generierten Spezifikation, analysiert. Die Art der Elemente, durch die die verschiedenen Iterationen unterschieden werden, wird als Iterationskonstrukt definiert. Das Iterationskonstrukt und seine verschiedenen Arten werden in Unterabschnitt 5.3.1 beschrieben. Um Iterationen auszuführen, werden zusätzlich zum Iterationskonstrukt die Mechanismen Iterator und

Abbruchbedingung benötigt. Diese werden in Unterabschnitt 5.3.2 und Unterabschnitt 5.3.3 beschrieben.

Für eine iterative Ausführung des Projektionszyklus gibt es zwei Dimensionen, aus denen sich die Iterationsart zusammensetzt.

- **(Eingabemodell) parallel oder sequentiell:** Nutzen die Iterationen alle als Eingabe das ursprüngliche Architekturmodell oder jeweils die Rückprojektion der Vorgängeriteration?
- **(Ausgabemodell) gemeinsames oder getrenntes Ergebnis:** Erfolgt die Rückprojektion in ein gemeinsames oder in getrennte Ergebnismodelle?

Im Folgenden werden die vier entstehenden Iterationsarten zu jeder Kombinationsmöglichkeit der Dimensionswerte mit ihren Einsatzgebieten, Vor- und Nachteilen beschrieben. Eine grafische Darstellung bietet Abbildung 5.5.

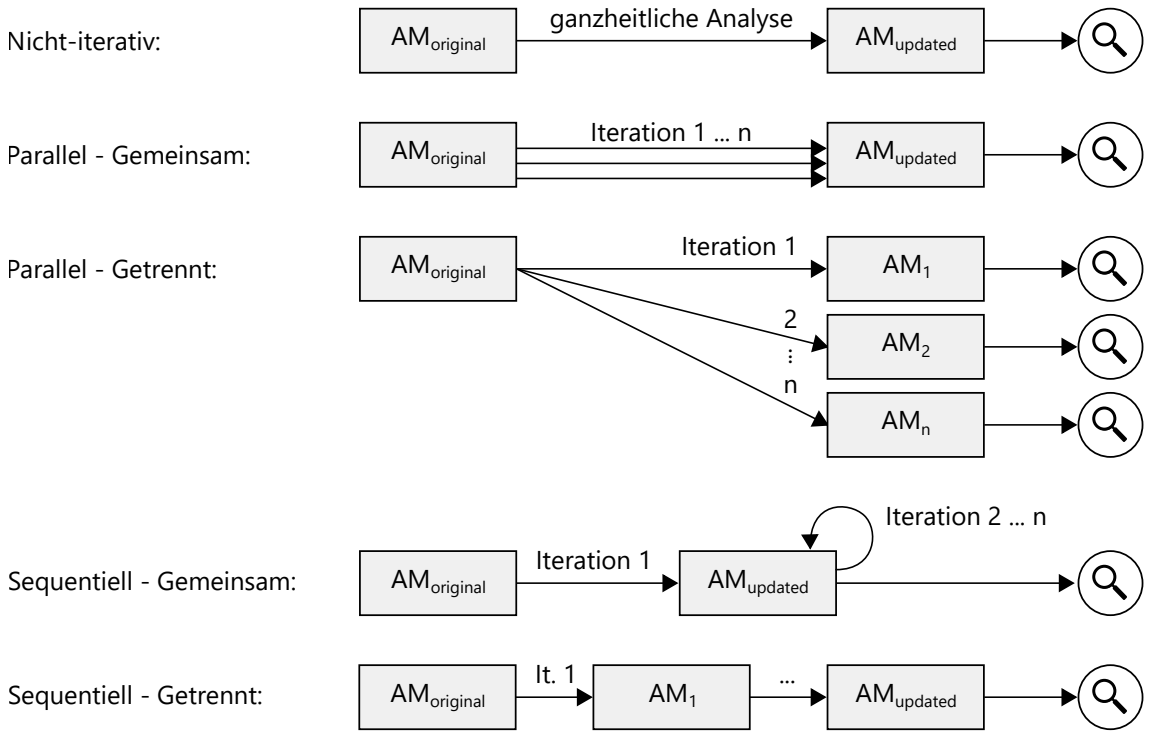


Abbildung 5.5: Ablauf der vier möglichen Iterationsarten im Vergleich zur nicht-iterativen Analysenkopplung. Die Instanzen des Architekturmodells werden mit AM abgekürzt und die anschließende Untersuchung durch die Architekturanalyse mit dem Lupensymbol dargestellt.

Parallel - Gemeinsames Ergebnis In dieser Iterationsart nutzt jeder Projektionszyklus der Iterationen das ursprüngliche Architekturmodell als Eingabe. Die Iterationen führen ihre Teilanalysen aus und projizieren das Ergebnis in ein gemeinsames Ergebnismodell. Dieses wird im Anschluss durch die Architekturanalyse überprüft. Wie in Abbildung 5.5 deutlich wird, unterscheiden sich die instanziierten Modelle in ihrer Anzahl nicht von der ganzheitlichen Analyse.

In dieser Iterationsart können einzelne Teilanalysen die Eingabe an die Quelltextanalyse partitionieren. Durch kleinere Mengen an zu untersuchenden Elementen kann Arbeitsspeicherplatz und Laufzeit gespart werden (Ziel 3). Hierbei ist wichtig, ob die Iterationen zeitlich parallel ausgeführt werden oder zeitlich hintereinander. Bei einer zeitlich parallelen Ausführung wird nur die Laufzeit verringert; der Arbeitsspeicherverbrauch summiert sich über die Teilanalysen. Im umgekehrten Fall wird durch die Hintereinanderausführung kein Laufzeitgewinn erzielt, jedoch Arbeitsspeicher eingespart. Es muss somit in jeder konkreten Konfiguration abgewogen werden, welche Performanzmetrik der limitierende Faktor und deshalb wichtiger ist.

Die Genauigkeit (Ziel 2) wird gegenüber dem nicht-iterativen Ansatz nicht erhöht, da die Rückprojektionen ebenfalls in ein gemeinsames Ergebnismodell zusammengefügt werden müssen. Bei Kollisionen kann wie bei der ganzheitlichen Analyse nur ein Approximationsoperator genutzt werden, z. B. die Quellendisjunktion. Außerdem muss für die Erkennung von Kollisionen das bereits vorhandene Ergebnismodell geladen werden. Es wird somit ein zusätzlicher Schritt im Projektionszyklus während der *Resolution*-Phase der Analyseschritte nötig.

Parallel - Getrenntes Ergebnis In dieser Iterationsart nutzt jede Iteration unabhängig das ursprüngliche Architekturmodell und projiziert die Ergebnisse in ein eigenes, getrenntes Ergebnismodell (vgl. Abbildung 5.5). Im Anschluss wird für jedes Ergebnismodell eine eigene Architekturanalyse durchgeführt.

Der Vorteil dieser Iterationsart ist, dass durch die getrennte Rückprojektion Kollisionen vermieden werden können. Dazu müssen die Eingabemodelle der Iterationen so aufgeteilt werden, dass in einer Iterationen nur ein einzelnes Level untersucht wird. Wie in Abschnitt 5.2 hergeleitet, entstehen dadurch nicht mehrere illegale Flüsse mit unterschiedlichen Quellenleveln. Eine Aktualisierung der Vertraulichkeitseigenschaften ist so durch das einzelne Level durchführbar. Damit kann der Supremumsoperator umgesetzt werden und die Genauigkeit erhöht sich (Ziel 2), da die Abdeckung aller möglichen Verletzungen gewährleistet ist. Die Ausführung der Architekturanalysen auf den getrennten Ergebnismodellen macht diese sichtbar. Bezüglich der Performanz (Ziel 3) kann durch das Partitionieren der Eingabe, durch kleinere Elementmengen, die es zu untersuchen gilt, eine Reduktion des Arbeitsspeicherverbrauchs und der Laufzeit erzielt werden.

Nachteil dieser Iterationsart ist, dass die Kopplung mit der Architekturanalyse für jedes Ergebnismodell einzeln ausgeführt wird. Somit entsteht ein Mehraufwand, der die Performanz (Ziel 3) wiederum negativ betrifft. Es entsteht außerdem ein Zusatzaufwand im Speicherverbrauch, da es mehr Ergebnismodelle gibt, als im nicht-iterativen Ansatz.

Sequentiell - Gemeinsames Ergebnis In dieser Iterationsart wird die erste Iteration auf dem ursprünglichen Architekturmodell ausgeführt. Jede weitere Iteration nutzt das gemeinsame Ergebnismodell, das durch die Vorgänger-Iteration potentiell bereits aktualisiert wurde, als Eingabemodell für die eigene Ausführung. Erst nach Beenden der letzten Iteration, wird die Architekturanalyse durchgeführt.

Vorteil dieser Iterationsart ist die Übertragung von Aktualisierungen in die Eingabe der anderen Iterationen. Hierdurch besteht die Möglichkeit, dass Iterationen auch Konstrukte partitionieren können, deren Flüsse eigentlich über diese Partitionsgrenzen abgeschnitten werden. Dies ist z. B. der Fall, wenn Systemkomponenten auf Iterationen aufgeteilt werden sollen und in jeder Iteration eine einzelne Komponente untersucht wird. Damit keine Flüsse übersehen werden, die zwischen den Komponenten stattfinden, werden weitere Iterationen an den iterativen Ausführungs-Stack angehängt, die das System global betrachten und nur Flüsse zwischen Komponenten untersuchen. Die Ergebnisse innerhalb der Komponenten liegen dann schon vor und werden in den angehängten Iterationen, die das System global betrachten, mit einbezogen. Wichtig ist hierbei, dass durch die verschiedenen Abstraktionsebenen (Intra-Komponenten und Inter-Komponenten) durch jede Iteration Rückprojektionen entstehen können, die wiederum die jeweils andere Abstraktionsebene betrifft. Es müssen deshalb solange abwechselnde Iterationen im Stack hinzugefügt werden, bis sich das Ergebnismodell stabilisiert hat. Das beschriebene Vorgehen eignet sich dazu, sehr große Systeme analysierbar zu machen, die nicht als komplettes Modell geladen werden können. Die Interaktionen werden dann auf die Abstraktionsebenen Intra-Komponenten und Inter-Komponenten aufgeteilt und in entsprechenden Iterationen abwechselnd ausgeführt. Dieses Vorgehen dient dem Ziel der Skalierbarkeit (Ziel 1).

Die Laufzeit wird durch die erneuten Ausführungsdurchgänge bis hin zur Ergebnisstabilisierung erhöht. Dies beeinflusst die Performanz (Ziel 3) negativ. Auch die Genauigkeit (Ziel 2) wird nicht verbessert. Dies liegt einerseits daran, dass nur ein gemeinsames Ergebnismodell vorliegt und so Kollisionen von rückprojizierten Leveln entstehen können. Dadurch müssen die Ergebnisse kombiniert werden, z. B. durch den Quellendisjunktionsoperator, der jedoch eine Unterabschätzung der Vertraulichkeitsverletzungen bedeutet. Andererseits kann es zu unterschiedlichen Ergebnissen kommen, je nachdem in welcher Reihenfolge die Iterationen im Stack liegen. Dies wird als Reihenfolgeeffekt definiert. Er kann immer dann auftreten, wenn Iterationen von einem Vorgängerergebnis abhängig sind und ihre Ausführungsreihenfolge nicht eindeutig bestimmbar ist. In Abbildung 5.6 existieren im ersten Beispiel drei Modellelemente. Diese haben die Level [Airline;TravelAgency], [Airline;User] und [TravelAgency]. Flüsse sind als Pfeile dargestellt und sind illegal, sobald die Senke nicht vollständig in der Quelle enthalten ist. Für einen illegalen Fluss wird das Quellenlevel an die Senke rückprojiziert. Im Beispiel wird einmal mit der Iteration gestartet, die alle Flüsse von Quellenelement i aus untersucht und einmal mit der Iteration, die alle Flüsse von Quellenelement ii aus untersucht. Das Ergebnismodell ist für beide Ausführungsreihenfolgen unterschiedlich. Auch im zweiten Beispiel entstehen je nach Reihenfolge unterschiedliche Ergebnismodelle. Diejenige Iteration, die zuletzt ausgeführt wird, überschreibt hierbei das Ergebnis der anderen. Es ist deshalb wichtig, die Ausführungsreihenfolge deterministisch festzulegen.

Sequentiell - Getrenntes Ergebnis In dieser Iterationsart nutzt jede Iteration das Ergebnismodell der Vorgängeriteration. Das Ergebnismodell der letzten Iteration wird anschließend mit der Architekturanalyse untersucht. Diese Iterationsart ähnelt der zuvor beschriebenen Iterationsart (Sequentiell - Gemeinsam) bis auf den Unterschied, dass viele Zwischenmodelle anfallen. Da nach dem Einlesen durch die Nachfolgeiteration kein weiterer Nutzen mehr

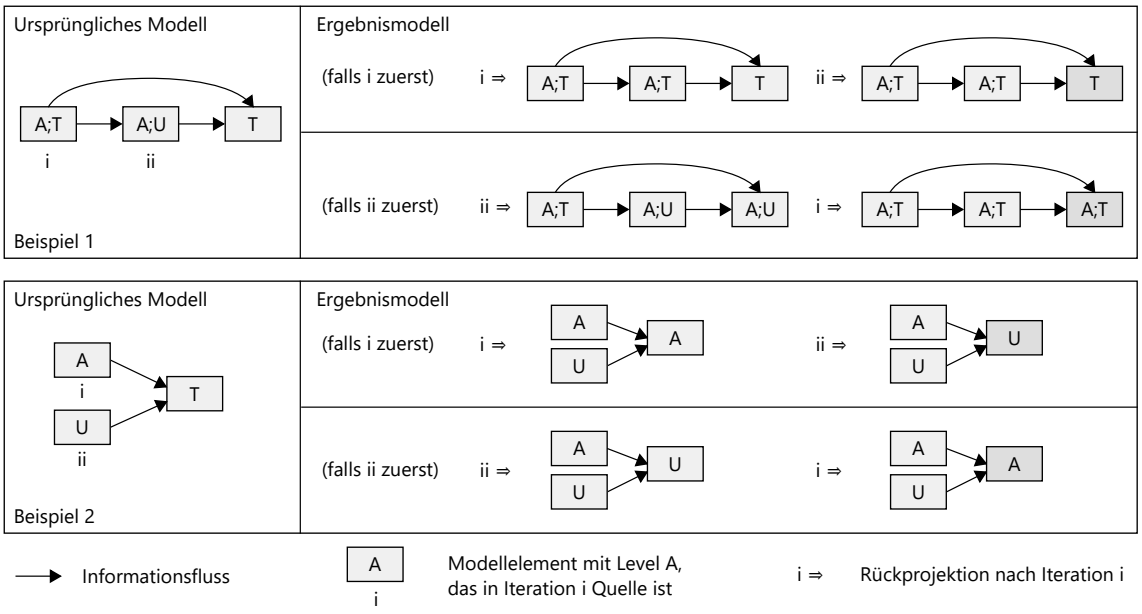


Abbildung 5.6: Reihenfolgeeffekte bei sequentieller Iterationsausführung. Durch unterschiedliche Abfolge der Iterationsschritte ergeben sich in den zwei Beispielen jeweils zwei verschiedene Ergebnismodelle. Unterschiede sind dunkelgrau markiert.

aus diesen Zwischenmodellen gezogen werden kann, können diese gelöscht werden. Der Ansatz wird dadurch äquivalent zur Iterationsart (Sequentiell - Gemeinsam), da nur noch das Ergebnismodell der letzten Iteration vorliegt.

5.3.1 Iterationskonstrukt

Das Iterationskonstrukt wird definiert als eine Grundmenge an Elementen, die sich aufteilen lässt und deren Elemente unabhängig analysierbar sind. Diese Elemente werden auf die Iterationen verteilt, um als separate Eingabe für die Quelltextanalyse zu dienen. Folgende Kategorien von partitionierbaren Elementen existieren für die Wahl des Iterationskonstrukts:

- **Systemelemente**, wie z. B. Teilkomponenten, Komponenten, Systeme und verteilte Systeme. Diese sind nach zunehmenden Abstraktionsgrad aufgezählt. Diese Klasse wird in Unterabschnitt 5.3.1.1 vorgestellt.
- **Spezifikationselemente**, wie z. B. Annotationen von Quellen und Senken an Parametern, Methoden und Klassen. Diese Klasse wird in Unterabschnitt 5.3.1.2 vorgestellt.
- **Sicherheitslevel**, wie z. B. alle ausgeprägten kombinierten Level. Diese Klasse wird in Unterabschnitt 5.3.1.3 vorgestellt.

Es lässt sich auch wie bei einer verschachtelten Schleife über mehrere Iterationskonstrukte gleichzeitig iterieren. Dies ist in manchen Fällen sogar notwendig, um verschiedene

Abstraktionsgrade miteinander zu verbinden wie z. B. Inter-Komponentenflüsse mit Intra-Komponentenflüssen. Eine Kombination von Iterationskonstrukten wird in Unterabschnitt 5.3.1.4 beschrieben.

5.3.1.1 Partitionierung über Systemelemente und deren Abstraktionsgrad

Als Grundmenge wird für dieses Iterationskonstrukt ein bestimmter Abstraktionsgrad des Systemmodells ausgewählt, z. B. die *Komponenten*. Es wird jeder Iteration genau eine einzelne Komponente zugewiesen und nur deren interne Flüsse analysiert. Nach Durchlauf aller Iterationen wurden alle Komponenten lokal aktualisiert (Intra-Komponente). Sollen sich die Komponenten zusätzlich gegenseitig beeinflussen, wird eine Iteration auf dem nächst höheren Abstraktionsgrad benötigt. Hierfür werden nur die nach außen hin sichtbaren Schnittstellen der Komponenten und ihre gegenseitigen Aufrufe als Quelltextmodell dargestellt (Inter-Komponenten). Durch eine Iteration dieses nächst höheren Abstraktionsgrades, werden die Komponenten durch Flüsse von den umliegenden, verbundenen Komponenten aktualisiert. Das Iterationskonstrukt der nächst höheren Abstraktionsebene wäre z. B. ein modelliertes *System*. Dadurch, dass auf Inter-Komponentensicht eine Aktualisierung nur an externen Schnittstellen der Komponenten durchgeführt werden, muss erneut eine Ausführung Iterationen auf Intra-Komponentensicht gestartet werden. Dieses Wechseln der Abstraktionsgrade muss solange ausgeführt werden, bis sich das gesamte Ergebnismodell stabilisiert hat.

Für dieses Iterationskonstrukt ist es notwendig, dass die Abstraktionsgrade miteinander kommunizieren können. Somit ist als Iterationsart die sequentielle Form erforderlich, wofür nur die Klasse (Sequentiell - Gemeinsam) in Frage kommt. Vorteil dieses Iterationskonstruktes ist bezüglich der Performanz (Ziel 3) das Aufschneiden des Gesamtsystems in leichtgewichtigere Systeme, wodurch der Arbeitsspeicheraufwand reduziert werden kann. Es können so außerdem Systeme untersucht werden, die zu groß für eine ganzheitliche Analyse sind, wodurch die Skalierbarkeit (Ziel 1) erhöht wird.

Nachteil ist, dass gewartet werden muss, bis sich das System stabilisiert hat. Die erhöhte Laufzeit ist ein Problem für die Performanz (Ziel 3) und es kann durch Zyklen in den Komponenteninteraktionen dazu kommen, dass sich Aktualisierungen im Kreis propagieren und es nie zu einer Stabilisierung kommt. Dies ist der Fall, wenn der Zyklus mindestens einmal den Abstraktionsgrad wechselt und er nicht vollständig in nur einer Ebene sichtbar wird. Eine zyklische Neuzuweisung von Sicherheitsleveln wird als Oszillation bezeichnet und in Abbildung 7.4 anschaulich dargestellt.

5.3.1.2 Partitionierung über Spezifikationselemente

Eine Quelltextanalyse untersucht Flüsse, die zwischen einer Quelle und einer Senke stattfinden. Eine ganzheitliche Analyse deckt die Menge aller solcher Flüsse ab. Bei einer iterativen Analyse kann aber über die Auswahl an Quellen und Senken in jeder Iteration eine explizite

Teilmenge der Flüsse untersucht werden. Hierfür werden Annotationen der Quelltextanalyse genutzt, die Modellelemente wie Parameter als Quelle oder Senke markieren. Diese Annotationen werden während der Hinprojektion aus der Architekturspezifikation generiert. Diese generierten Annotationen, können für jede Iteration auf einen bestimmten Teil reduziert werden, um nur bestimmte Quellen oder Senken zu betrachten.

Im Folgenden wird der Vorteil des Partitionierens der Quellenannotationen beschrieben. Werden die Flussquellen auf Iterationen aufgeteilt und jeweils nur eine Quelle pro Iteration analysiert, gibt es bei illegalen Flüssen nur ein Level, das für eine Aktualisierung der Vertraulichkeitsinformationen zurückprojiziert werden muss. Es entstehen keine Kollisionen beim Aktualisieren, die ein Kombinieren von Sicherheitsleveln erfordern. Somit wird kein Verknüpfungsoperator benötigt, der das Ergebnis unterapproximiert. Es wird deshalb die volle Abdeckung der Verletzungen innerhalb einer Iteration erreicht. Für das Ziel der Genauigkeit (Ziel 2) empfiehlt es sich daher, als Iterationsart (Parallel - Getrennt) zu nutzen, da nur dort die Trennung der Rückprojektionen erhalten bleibt und jedes Ergebnismodell einzeln überprüft wird. Bezüglich der Performanz (Ziel 3) bietet die Quellenpartitionierung als Iterationskonstrukt den Vorteil, dass sich Auslastung des Arbeitsspeichers reduziert, da die Quelltextanalyse nur eine Eingabequelle pro Iteration erhält und analysieren muss.

Wenn über Senken iteriert wird, wird in einer einzelnen Iteration genau eine Senke betrachtet. Wenn eine Informationsflussanalyse über einen Systemabhängigkeitsgraphen (SDG) arbeitet, wird dieser aber von den Startpunkten aus aufgestellt, die als Quellen annotiert sind. In ihm wird durch Kanten dargestellt, welche Teile des Quelltexts sich gegenseitig beeinflussen. Es wird in jeder Iteration der vollständige SDG von den Quellen aufgebaut, bis in einem Knoten die Senken auftauchen. Das bedeutet, dass in jeder Iteration alle Operationen der Quelltextanalyse gleich sind und nur das Ergebnis beschnitten wird. So entsteht ein hoher Performanzverlust durch die Mehrfachausführung (entgegen Ziel 3), obwohl das Ergebnis auch in einem Durchlauf erzielt werden könnte. Aus diesem Grund sind die Senken als Iterationskonstrukt ungeeignet.

5.3.1.3 Partitionierung über Sicherheitslevel

In diesem Iterationskonstrukt werden als Grundmenge der Elemente die ausgeprägten Sicherheitslevel des Modells genutzt. In einer bestimmten Iteration wird nur eines dieser Level ausgewählt und nur diejenigen Flüsse untersucht, in denen Informationen weitergegeben werden, die dieses Level enthalten. Die Auswahl solcher Flüsse kann z. B. über die Quellenannotationen der Spezifikation und deren annotierter Level durchgeführt werden.

Vorteil dieses Iterationskonstrukts ist, dass wie in Unterabschnitt 5.3.1.2 für die Quellenpartitionierung keine Kollisionen von Aktualisierungen auftreten, da nur ein einziges Sicherheitslevel für Flussquellen verfügbar ist und deshalb nur dieses in der Rückprojektion von illegalen Flüssen auftritt. Zusätzlich werden alle Quellen in einer Iteration zusammengefasst, die dasselbe Level haben. Das bedeutet, dass insgesamt weniger Iterationen ausgeführt werden müssen als bei der reinen Quellenpartitionierung. Bezüglich der Performanz (Ziel 3) ist dies ein Vorteil. Gleichzeitig wird beim Nutzen der Iterationsart (Parallel - Getrennt) aber

die Genauigkeit wie durch die Quellenpartitionierung optimiert. Das Iterationskonstrukt der Quelllevelpartitionierung stellt somit eine generelle Verbesserung der reinen Quellenpartitionierung dar. Das Iterieren über Senkenlevel wird aus dem gleichen Grund nicht empfohlen, aus dem bereits das Iterieren über Senken als ungeeignet eingestuft wurde (vgl. Unterabschnitt 5.3.1.2).

5.3.1.4 Kombination von Iterationskonstrukten

Iterationskonstrukte können miteinander kombiniert werden. Dabei wird für eine spezielle Iteration eine Auswahl der Elemente aus jedem der Iterationskonstrukte getroffen, die jeweils eine Dimension darstellen und nur das analysiert, was in den Schnitten aller dieser Dimensionen enthalten ist.

Überträgt man das Beispiel mehrerer Komponenten, die innerhalb eines Systems liegen, auf eine Kombination von Iterationskonstrukten, ist eines dieser Iterationskonstrukte der *Abstraktionsgrad*: entweder Intra- oder Inter-Komponentenebene. Wenn für eine Iteration die Intra-Komponentenebene ausgewählt wird, muss das Iterationskonstrukt *Komponentenpartition* als weiteres Konstrukt zur Auswahl einer konkreten Komponente für diese Iteration bereitstehen. Wenn für die Iteration hingegen die Inter-Komponentenebene ausgewählt wird, muss kein weiteres Konstrukt bereitstehen. Stattdessen muss in diesem Fall ein gänzlich anderes Modell für die Analyse bereitgestellt werden, das auf höherem Abstraktionsgrad nur die Interaktionen der Komponenten repräsentiert, nicht aber ihre internen Abläufe.

Die Verschaltung bei der Kombination derartiger Iterationskonstrukte wird in Unterabschnitt 5.3.2 als Mechanismus Iterator beschrieben.

Die Kombination von Iterationskonstrukten, die aus einem gleichen Abstraktionsgrad stammen, ermöglicht eine feingranularere Aufteilung der Elemente auf Iterationen. Werden z. B. die beiden Iterationskonstrukte *Quellenpartitionierung* und *Senkenpartitionierung* genutzt, wird pro Iteration nur genau ein Fluss analysiert, da nur eine Quelle und eine Senke durch die Kombination ausgewählt wird. So kann eine totale Auffächerung aller Flüsse auf einzelne Iterationen durchgeführt werden.

5.3.2 Iterator

Der Iterator ist ein Zeiger, mit dem alle Teilmengen des Iterationskonstrukts durchlaufen werden können. In einer konkreten Iteration zeigt er auf eine Teilmenge, deren Elemente sich in der Eigenschaft gleichen, die das Iterationskonstrukt vorgibt. Hierfür müssen die Teilmengen indiziert werden und dürfen sich nicht verändern, da sonst die vollständige Abdeckung der Elemente durch die Iterationen nicht gewährleistet ist.

Für zwei kombinierte Iterationskonstrukte muss der Iterator zwei Zeiger besitzen, die auf die aktuell ausgewählten Teilmengen im jeweiligen Konstrukt zeigen. Wie in Unterabschnitt 5.3.1.4 beschrieben, wird in manchen Fällen, in denen sich der Abstraktionsgrad zwischen

Iterationen ändert, sogar eine Fallunterscheidung für das innere Iterationskonstrukt der verschachtelten Schleife nötig. Eine Inkrementierung der Zeiger muss solange nur für das innere Iterationskonstrukt durchgeführt werden, bis es vollständig durchlaufen wurde. Erst dann wird einmalig der Zeiger des äußeren Konstrukts inkrementiert und das innere Konstrukt durch einen höheren Abstraktionsgrad abgelöst.

5.3.3 Abbruchbedingung

Eine Abbruchbedingung ist in iterativen Ansätzen wichtig, um zu garantieren, dass die Ausführung terminiert. Im Rahmen dieser Arbeit werden zwei mögliche Abbruchbedingungen betrachtet: Als Erstes die vollständige Abdeckung des Iterationskonstrukts und somit aller Flüsse von den Quellen zu den Senken, als Zweites die Stabilisierung des Ergebnismodells. Im ersten Fall werden alle Flüsse genau einmal analysiert. Im zweiten Fall kommt es solange zur erneuten Ausführung aller Iterationen, bis keine Veränderungen mehr während der Rückprojektion festgestellt werden. Ob es einen stabilen Endzustand für das Modell gibt, hängt jedoch von der Iterationsart und dem Iterationskonstrukt ab.

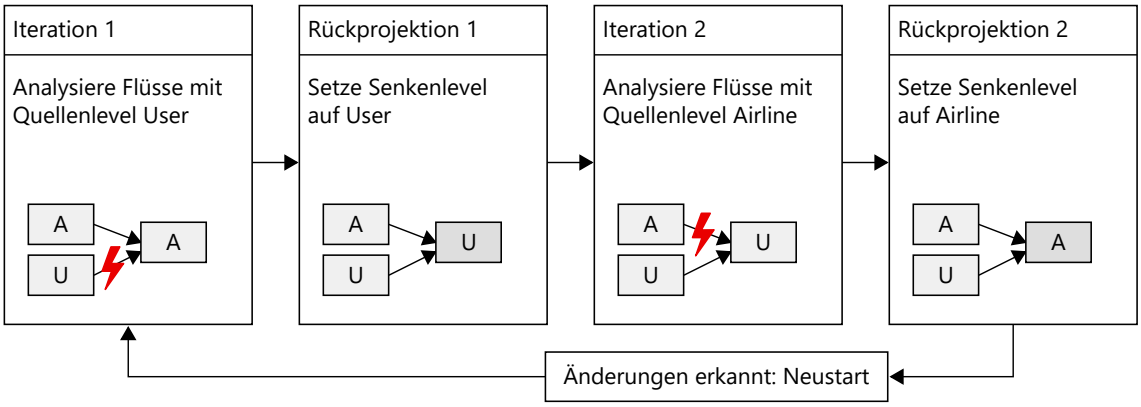


Abbildung 5.7: Oszillation: Zyklische Neuzuweisung des Sicherheitslevel an der Senke

In Abbildung 5.7 wird dargestellt, dass bei einer sequentiellen Iterationsart und dem Iterationskonstrukt Quellenlevel eine alternierende Neuzuweisung des Senkenlevels auftritt und so eine Terminierung der Iterationsdurchläufe ausbleibt, weil in jedem Durchlauf Änderungen registriert werden.

Wenn als Abbruchbedingung eine Stabilisierung des Modells gewählt wird, kann dies entweder durch einen Beobachter am Ergebnismodell oder ein *Flag* während der Rückprojektion durchgeführt werden. Dies ist Entscheidung der Implementierung und wird für diese Arbeit in Unterabschnitt 7.2.2 beschrieben.

Bei der Auswahl einer Iterationsart für den Ansatz muss die Korrektheit und Eignung einer gewählten Abbruchbedingung somit überprüft werden. Bei einer Iteration über Komponenten und verschachtelten Iterationskonstrukten ist als Abbruchbedingung die *Stabilisierung* zu wählen. In Fällen, in denen eine einmalige Betrachtung jedes Flusses ausreicht, ist die Abbruchbedingung *Vollständige Abdeckung* zu wählen.

5.4 Auswahl einer konkreten Iterationsart für diese Arbeit

Es hat sich gezeigt, dass für die Ziele der Skalierbarkeit (Ziel 1), Ergebnisgenauigkeit (Ziel 2) und des Performanzgewinns (Ziel 3) jeweils andere Iterationsarten und Konstrukte am besten geeignet sind. Aufgrund zeitlicher Limitierungen dieser Masterarbeit mussten die Ziele priorisiert werden, um auszuwählen, welche Konfiguration umgesetzt werden sollte. Das Ziel der Genauigkeitsverbesserung wurde hierbei als größter Beitrag für die Forschung erachtet. Es wurde der Kontext der disjunktiv verknüpften Sicherheitslevel gewählt, da dort ein großes Verbesserungspotential der Genauigkeit erwartet wird, wie in Abschnitt 5.2 hergeleitet. Zusätzlich dazu wird für den Ansatz auch die Verbandsgenerierung optimiert, um die Skalierbarkeit des Ansatzes sicherzustellen. Dies wird in Kapitel 6 beschrieben. Die Performanzverbesserung ist zweitrangig, aber wird für die gewählte Konfiguration dennoch evaluiert.

Die Diskussion der Vor- und Nachteile der Iterationsarten und Iterationskonstrukte ergibt für das Ziel der Genauigkeit folgende beste Konfigurationsalternative:

- Iterationsart: (Parallel - Getrenntes Ergebnis)
- Iterationskonstrukt: Partitionierung der Quellenlevel
- Abbruchbedingung: Vollständige Konstruktabdeckung
- Iterator: Trivial über einzelnes Konstrukt

Für die Kopplung wird als Quelltextanalyse die Informationsanalyse JOANA [37] und die Datenflussanalyse CodeQL [40] genutzt. Zur Überprüfung der Vertraulichkeitsverletzungen auf Architekturebene wird die Access Analysis von Kramer [5] gewählt. Es wird zu jeder iterativen Konfiguration eine nicht-iterative Konfiguration für den direkten Vergleich erstellt, wodurch sich folgende vier Konfigurationen ergeben und in die Evaluation in Kapitel 8 einfließen:

- **(JOANA-non-it)**: Nicht-iterativer Ansatz mit Quellendisjunktionsoperator. Repräsentativ für den Ansatz Härings [9]. Kopplung von JOANA und der Access Analysis.
- **(JOANA-it)**: Iterativer Ansatz, der JOANA und die Access Analysis koppelt. Repräsentiert den Supremumsoperator.
- **(CodeQL-non-it)**: Nicht-iterativer Ansatz mit Quellendisjunktionsoperator. Kopplung von CodeQL und der Access Analysis.
- **(CodeQL-it)**: Iterativer Ansatz, der CodeQL und die Access Analysis koppelt. Repräsentiert Supremumsoperator.

In der Implementierung, die in Kapitel 7 beschrieben wird, werden alle Mechanismen dieses Konzeptionskapitels vollumfänglich im Analyseframework bereitgestellt, wodurch auch andere Konfigurationen in zukünftigen Arbeiten instanziierbar wären.

6 Optimierung der Sicherheitsverbände

Sicherheitsverbände, die in Unterabschnitt 2.1.5 definiert wurden, spezifizieren im Kontext dieser Arbeit paarweise, ob zwischen zwei Sicherheitsleveln ein erlaubter Fluss stattfinden darf und wie dieser orientiert ist. In diesem Kapitel wird beschrieben, wie die Sicherheitsverbände, die für den Informationstransfer an die Quelltextanalysen benutzt werden, optimiert werden können. Hierzu wird in Abschnitt 6.1 erläutert, welche Einsparungen durch das Nutzen der transitiven Reduktion [60] statt der Potenzmenge bei der Anzahl an notwendigen Relationskanten zwischen Sicherheitsleveln entstehen. Abschnitt 6.2 geht darauf ein, dass es ausreicht, im Sicherheitsverband nur genau die im Modell konkret ausgeprägten kombinierten Sicherheitslevel zu inkludieren. Zuletzt wird in Abschnitt 6.3 beschrieben, wie sich verschiedene Repräsentationen von Sicherheitsleveln auf die Verbandsgenerierung auswirken.

6.1 Transitive Reduktion

Sicherheitslevel können in der Architektur Eigenschaften von Modellelementen beschreiben. Im Kontext von Kramers Access Analysis werden z. B. DataSets definiert, die bestimmte Daten zusammenfassen und den erlaubten Zugriff auf diese Daten repräsentieren. Auf einen Parameter, dem das DataSet [User] zugewiesen ist, darf nur von einem Akteur zugegriffen werden, wenn dieser das DataSet kennen darf. Der Akteur Airline aus dem fortlaufenden Beispiel darf nur DataSet [Airline] kennen und hat somit keinen erlaubten Zugriff. Im Kontext der Sicherheitslevel lassen sich die DataSets als einzelne Level bezeichnen, die disjunktiv zu komplexeren Zugriffsberechtigungen verknüpft werden können, z. B. [Airline;Support]. Ob die Kombination von Sicherheitsleveln disjunktiv oder konjunktiv ist, ist von der Domäne des modellierten Systems abhängig. Sie entscheidet, ob die kombinierten Level höher oder niedriger vertraulich sind als einzelne Level, wie in Unterabschnitt 2.1.3 beschrieben. Wichtig ist im Folgenden erst einmal nur, dass es eine Teilmengenrelation zwischen kombinierten Leveln geben kann, genau dann, wenn alle einzelnen Level in einem anderen kombinierten Level enthalten sind.

Zu Beginn werden die genutzten Begriffe und die Notation definiert. Es sei $L = \{l_1, \ldots, l_n\}$ die Menge an einzelnen Sicherheitsleveln, die im System auftreten und $n = |L|$ die Anzahl an einzelnen Sicherheitsleveln. Es sei $X = P(L)$ die Potenzmenge der einzelnen Sicherheitslevel und somit die Menge aller kombinierten Sicherheitslevel. Sie entspricht aller möglichen Kombinationen von einzelnen Sicherheitsleveln. Es sei $N = |X| = |P(L)|$ die Anzahl an kombinierten Sicherheitsleveln. Dies ist die Größe der Potenzmenge und lässt sich über

$N = 2^n$ berechnen [61]. Es sei X_k die Menge aller möglichen kombinierten Sicherheitslevel, die genau k einzelne Level als Elemente enthalten. Hierbei ist k die Kardinalität der Menge X_k. Das Sicherheitslevel [Airline;User;TravelAgency] hat z. B. die Kardinalität 3. Es sei $N_k = |X_k|$ die Anzahl kombinierter Sicherheitslevel in X_k. Zur Berechnung wird der Binomialkoeffizient genutzt:

$$N_k = \binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

Veranschaulicht werden kann die Potenzmenge über eine Binärrepräsentation der einzelnen Sicherheitslevel. Jedes Sicherheitslevel wird auf eine der Binärstellen abgebildet, die die Indizes 1 bis n haben. Eine Eins in Index i steht hierbei dafür, dass das einzelne Sicherheitslevel zu Index i in der konkreten Kombination vorkommt, eine Null steht dafür, dass es nicht vorkommt.

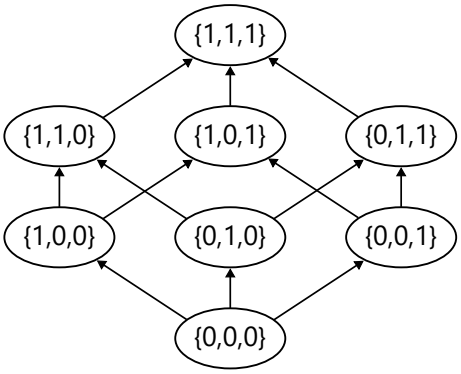


Abbildung 6.1: Die Potenzmenge einer dreielementigen Menge, dargestellt als Hassediagramm

Wie in Abbildung 6.1 als Hassediagramm dargestellt, ist die Teilmengenrelation eine Halbordnung auf der Potenzmenge, bei der das kombinierte Level x_1 in der Binärrepräsentation genau dann eine Teilmenge von einem anderen kombinierten Level x_2 ist, wenn es nur Bitflips der Binärstellen von Null zu Eins, aber keine von Eins zu Null gibt.

Für die Erstellung eines Sicherheitsverbandes muss spezifiziert werden, welche Flüsse zwischen zwei kombinierten Leveln erlaubt sind. Ein trivialer Ansatz ist es, über einer Liste aller kombinierten Level paarweise Vergleiche durchzuführen und die Relationskante zwischen je zwei Elementen entsprechend der Teilmengenrelation zu orientieren. Hierzu macht es Sinn, die Liste der Elemente vorher nach der Anzahl k enthaltener Einzellevel zu ordnen. Denn zwischen zwei verschiedenen kombinierten Leveln x_1 und x_2, die die gleiche Kardinalität k besitzen, kann die Halbordnungsrelation der Teilmenge nicht ausgewertet werden, da keines der beiden eine Teilmenge des anderen ist, wie in Abbildung 6.1 deutlich wird. Die Einschränkung der Vergleichsdurchführung auf kombinierte Level mit verschiedenem k spart Vergleichsoperationen.

In Härings Ansatz wird der Sicherheitsverband auf eine solche triviale Weise berechnet. Betrachtet man ein Beispiel mit $n = 8$ einzelnen Sicherheitsleveln, ergibt dies $N = 256$

kombinierte Level und über die Gaußsche Summenformel eine Anzahl von 32.640 Vergleichsoperationen. Die Potenzmengenberechnung und ein Verband über alle kombinierten Sicherheitslevel skaliert schlecht und sollte deshalb vermieden werden.

Da die genutzten Analysen transitive Abhängigkeiten zwischen erlaubten Flüssen implizit annehmen, müssen zur Spezifikation der erlaubten Flüsse $A \rightarrow AB$, $AB \rightarrow ABC$ und $A \rightarrow ABC$ nicht alle drei in einem Verband angegeben werden. Es reicht aus, $A \rightarrow AB$ und $AB \rightarrow ABC$ zu spezifizieren, denn der dritte erlaubte Fluss ist in diesen beiden bereits transitiv enthalten. Die transitive Hülle bleibt in beiden Fällen gleich. Die minimale Relation, die bezüglich einer Relation dieselbe transitive Hülle hat, nennt sich transitive Reduktion. Im Fall des Sicherheitsverbandes genügt es die transitive Reduktion als Verband zu spezifizieren, um alle erlaubten Flüsse zu erfassen. Für gerichtete, endliche, azyklische Graphen existiert sie und ist eindeutig [60].

Im Folgenden wird über das Herleiten expliziter Formeln gezeigt, wie stark die Anzahl der Verbandskanten durch eine transitive Reduktion verringert wird. Hierfür sei F die Anzahl der erlaubten Flüsse bzw. Kanten der Verbandsrelation und F_k die Anzahl der erlaubten Flüsse, die insgesamt von Elementen aus X_k ausgehen.

Drei Gleichungen werden für nachfolgende Umformungen benutzt und deshalb zuvor eingeführt:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} \cdot a^{n-k} \cdot b^k \qquad \text{Binomische Formel} \qquad (6.1)$$

$$\sum_{k=0}^{n} N_k = \sum_{k=0}^{n} \binom{n}{k} = 2^n \qquad \text{Summenformel für Binomialkoeffizienten} \qquad (6.2)$$

$$N_k = \binom{n}{k} = \binom{n}{n-k} = N_{n-k} \qquad \text{Symmetrie der Binomialkoeffizienten} \qquad (6.3)$$

Zuerst wird die Gesamtanzahl aller Verbandskanten über die Summe der Kanten berechnet, die aus den kombinierten Leveln mit jeweils gleichem k stammen:

$$F_{gesamt} = \sum_{k=0}^{n} F_k = \sum_{k=0}^{n} \left(N_k \cdot \sum_{j=k+1}^{n} F_{k,j} \right) \qquad \text{Kanten im vollständigen Verband} \qquad (6.4)$$

Hierbei ist N_k die Anzahl der Elemente in X_k. $F_{k,j}$ ist die Anzahl der erlaubten Flüsse, die in genau einem Element von X_k beginnen und in allen Elementen von X_j münden, die der Teilmengenrelation entsprechen. Weil $(n-k)$ die Anzahl Nullen in einem kombinierten Level aus X_k ist, die noch zu einer Eins werden können und $(j-k)$ die Anzahl an hinzukommenden Einsen ist, sodass X_j erreicht wird, ergibt sich Formel 6.5. Anschaulich ausgedrückt müssen $(j-k)$ Stellen aus $(n-k)$ möglichen Stellen für einen Bitflip gewählt werden.

$$F_{k,j} = \binom{n-k}{j-k} \tag{6.5}$$

Sei nun $m = (n-k)$ eine Substitution der Anzahl an freien Nullen in einem kombinierten Level aus X_k. Dann kann die Summe der $F_{k,j}$ mit Formel 6.5 zu folgendem Term umformuliert werden:

$$\begin{aligned}
\sum_{j=k+1}^{n} F_{k,j} &= \sum_{j=k+1}^{n} \binom{n-k}{j-k} && \text{mit Indexverschiebung um } k: \\
&= \sum_{j=k+1-k}^{n-k} \binom{n-k}{(j+k)-k} && \text{mit } m = (n-k): \\
&= \sum_{j=1}^{m} \binom{m}{j} && \text{und mit Gleichung 6.2 folgt:} \\
&= 2^m - \binom{m}{0} \\
&= 2^m - 1 \\
&= 2^{n-k} - 1
\end{aligned}$$

Einsetzen in Formel 6.4 ergibt:

$$\begin{aligned}
F_{gesamt} &= \sum_{k=0}^{n} \left(N_k \cdot \left(2^{n-k} - 1 \right) \right) \\
&= \sum_{k=0}^{n} \left(\binom{n}{k} \cdot 2^{n-k} - \binom{n}{k} \right) \\
&= \sum_{k=0}^{n} \left(\binom{n}{k} \cdot 2^{n-k} \cdot \underbrace{1^k}_{=1} \right) - \sum_{k=0}^{n} \binom{n}{k} && \text{und mit Gleichung 6.1 folgt:} \\
&= (2+1)^n - \sum_{k=0}^{n} \binom{n}{k} && \text{und mit Gleichung 6.2 folgt:} \\
&= (2+1)^n - 2^n \\
&= 3^n - 2^n
\end{aligned}$$

In einem vollständigen Verband beträgt die Anzahl erlaubter Flüsse somit $3^n - 2^n$, was nach Landau-Notation in der exponentiellen Größenordnung $O(3^n)$ liegt.

Als Zweites wird die Anzahl an Kanten in der transitiven Reduktion berechnet.

$$F_{reduziert} = \sum_{k=0}^{n} F_k = \sum_{k=0}^{n} N_k \cdot (n-k) \qquad \text{Kanten in der transitiven Reduktion} \quad (6.6)$$

Hierfür werden im Vergleich zu der totalen Formel 6.4 nur noch Kanten betrachtet, die aus Elementen von X_k mit nur einem einzigen BitFlip in die direkt folgenden Elemente von X_{k+1} gehen. $(n-k)$ ist die Anzahl an Nullen in einem kombinierten Level aus X_k, die für das nächstgrößere k zu einer Eins werden können. Durch Umformung und Einsetzen ergibt sich:

$$\begin{aligned}
F_{reduziert} &= \sum_{k=0}^{n} (N_k \cdot (n-k)) \\
&= \frac{1}{2} \sum_{k=0}^{n} (2 \cdot N_k \cdot (n-k)) \\
&= \frac{1}{2} \sum_{k=0}^{n} (N_k \cdot (n-k) + N_k \cdot (n-k)) && \text{und mit Gleichung 6.3 folgt:} \\
&= \frac{1}{2} \sum_{k=0}^{n} \left(N_k \cdot (n-k) + \underbrace{N_{n-k} \cdot (n-k)}_{2.Summand} \right) && \text{Umsortieren nach } k = (n-k): \\
&= \frac{1}{2} \sum_{k=0}^{n} (N_k \cdot (n-k) + N_k \cdot (k)) \\
&= \frac{1}{2} \sum_{k=0}^{n} (N_k \cdot (n-k+k)) \\
&= \frac{1}{2} \sum_{k=0}^{n} (N_k \cdot n) && \text{und mit Gleichung 6.2 folgt:} \\
&= 2^n \cdot \frac{n}{2}
\end{aligned}$$

In einem transitiv reduzierten Verband beträgt die Anzahl erlaubter Flüsse somit $2^n \cdot \frac{n}{2}$, was nach Landau-Notation [62] in der Größenordnung $O(2^n \cdot n)$ liegt. Die exponentielle Basis wird gegenüber der totalen Abschätzung um 1 reduziert.

Verbandsstruktur	Formel Kantenanzahl	n=1	n=2	n=3	n=4	n=8	n=16
Vollständiger Verband	$F = 3^n - 2^n$	1	5	19	65	6.305	42.981.185
Transitive Reduktion	$F = 2^n \cdot \frac{n}{2}$	1	4	12	32	1.024	524.288

Tabelle 6.1: Kantenanzahl eines vollständigen Verbands verglichen mit der transitiven Reduktion

Tabelle 6.1 vergleicht die Entwicklung der Anzahl der Verbandskanten für beispielhafte Einzellevelanzahl n und veranschaulicht die Ersparnis bei der Nutzung einer transitiven Reduktion. Die Reduktion der zu speichernden Kantenanzahl mit einem transitiv reduzierten Verband ist für die Nutzung im Ansatz ein erster Schritt, um die Skalierbarkeit zu erhöhen.

6.2 Beschränkung auf existierende Level

Als Nächstes kann der Sicherheitsverband auf diejenigen Sicherheitslevel und kombinierten Sicherheitslevel beschränkt werden, die im eigentlichen Modell konkret ausgeprägt sind. Denn alle anderen kombinierten Level, die nicht im System auftreten, werden in der Analyse nie beim Überprüfen einer Flussregel genutzt. Auch die Kanten im Sicherheitsverband, die erlaubte Flüsse darstellen, müssen nie als Regeln überprüft werden, weil es die Elemente nicht gibt, die sich auf einen solchen Fluss beziehen. Im Folgenden wird nun ein Vorgehen beschrieben, das die transitive Reduktion eines Sicherheitsverbandes aufstellt, in dem nur vorliegende, konkrete, kombinierte Level inkludiert werden. Damit wird die Berechnung der Potenzmenge vollständig vermieden.

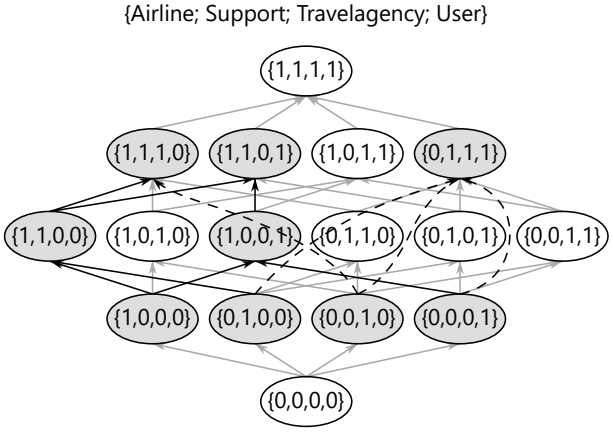


Abbildung 6.2: Kanten eines Sicherheitsverbands, bei dem nur die grau markierten 9 Level ausgeprägt sind. Diese kombinierten, existierenden Level sind: Airline, Support, Travelagency, User, AirlineSupport, AirlineUser, AirlineSupportTravelagency, AirlineSupportUser, SupportTravelagencyUser. Gestrichelte Kanten stellen Relationen dar, die mindestens eine Kardinalität überspringen und deshalb in der ursprünglichen transitiven Reduktion noch nicht enthalten waren.

In Abbildung 6.2 wird anhand eines Beispiels mit 9 ausgeprägten kombinierten Leveln verdeutlicht, wie sich der ursprüngliche Potenzmengenverband und der grau markierte, auf existierende Level beschränkte Verband unterscheiden. Die Kantenanzahl des Sicherheitsverbandes reduziert sich in diesem Beispiel durch die Levelbeschränkung von 32 auf 11. Unter der Annahme, dass bei großen n viele der kombinierten Sicherheitslevel in einem realen System nie ausgeprägt werden und der Verbandsgraph der Potenzmenge in diesem Fall sehr dünn besetzt ist, wächst die Kantenanzahl im beschränkten Fall nur noch quadratisch mit der Anzahl an ausgeprägten Leveln statt exponentiell, wie es bei der Summe an Binomialkoeffizienten in Abschnitt 6.1 gezeigt wurde.

Zur Berechnung des beschränkten Verbandes wird folgendes Vorgehen entworfen:

1. Die ausgeprägten Level werden nach ihren Kardinalitäten in die Mengen X_k einsortiert.
2. Iteriere über die Mengen X_k von oben nach unten. Das alleinige Superelement in X_n hat keine Nachfolger in der Teilmengenrelation, weil es kein X_{n+1} gibt. Deshalb kann bei X_{n-1} begonnen werden.
3. Sei nun $k = i$ in einer beliebigen Iteration aus Schritt 2. Für jedes Element e in X_i berechne alle direkten Nachfolger. Hierfür betrachte aufsteigend die Mengen X_{i+1} bis X_n und vergleiche deren Elemente mit e. Ein direkter Nachfolger e' liegt vor, wenn gilt, dass e eine Teilmenge von e' bezüglich der enthaltenen Einzellevel ist. Bei der Betrachtung von X_{i+2} und höheren Kardinalitätsmengen wird zusätzlich eine zweite Bedingung nötig. Ein Nachfolger e' ist nur noch direkter Nachfolger, wenn er nicht bereits durch einen zuvor gefundenen direkten Nachfolger erreichbar ist.
4. Lege für jedes Element jeweils zu allen seinen direkten Nachfolgern eine Kante an. Die Orientierung der Kante entspricht der Relationsrichtung und ist vom Sicherheitsbegriff abhängig. Deren Einordnung folgt im nächsten Abschnitt.
5. Gib alle berechneten Kanten zurück. Dies ist die beschränkte transitive Reduktion der ausgeprägten Sicherheitslevel.

Verglichen mit den Ergebnissen aus dem vorherigen Abschnitt, stellt sich die Frage nach der Zeitkomplexität und der Speicherkomplexität des Ansatzes. Die Laufzeit der Verbandsberechnung mit dem beschriebenen Vorgehen ist bezüglich der Eingabegröße an n ausgeprägten kombinierten Leveln quadratisch, wie nachfolgend aufgelistet wird. Das Einsortieren der Elemente in ihre Kardinalitätsmengen hat eine Zeitkomplexität von $O(n)$. Jedes Element mit allen jeweils größeren Elementen zu vergleichen und eine Teilmengenprüfung durchzuführen, erfordert $\frac{n\cdot(n+1)}{2}$ Schritte, wenn die Teilmengenprüfung als konstant angenommen wird. Da die Teilmengenprüfung über die feste Anzahl an einzelnen Leveln skaliert und diese Anzahl im Allgemeinen deutlich geringer als die Anzahl an ausgeprägten Leveln ist, ist diese Abschätzung zulässig. Zuletzt werden die Kanten aus den direkten Nachfolgern erzeugt. Um diese zu traversieren, werden ebenfalls maximal $\frac{n\cdot(n+1)}{2}$ Schritte benötigt, da dies die obere Grenze der Anzahl an direkten Nachfolgern ist. Somit liegt die Zeitkomplexität nach der Landau-Notation in $O(n^2)$. Die Speicherkomplexität wird durch die Anzahl an nötigen Verbandskanten festgelegt. Diese liegt im schlechtesten Fall bei $\frac{n\cdot(n+1)}{2}$, wenn jedes ausgeprägte Level zu allen größeren Leveln eine Kante besitzt und ist somit wie die Zeitkomplexität quadratisch in $O(n^2)$.

Im trivialen Ansatz wird zuerst die Potenzmenge der Level gebildet und darauf die Teilmengenrelation zur Kantenauswahl genutzt. In dieser Kantenauswahl werden paarweise die n Level verglichen, wodurch eine Laufzeit in der Größenordnung von $\frac{n\cdot(n+1)}{2}$ Schritten entsteht. Hierbei ist n aber exponentiell von der Anzahl an einzelnen Leveln abhängig und somit deutlich größer als im neuen Ansatz. Zusätzlich entsteht für die Berechnung der Potenzmengenlevel ein Zusatzaufwand an Operationen in der Größenordnung von 2^n. Die Speicherkomplexität hängt ebenfalls exponentiell von n ab.

6.3 Umsetzung für verschiedene Levelrepräsentationen

In diesem Abschnitt wird für verschiedene Repräsentationen der Level (vgl. Unterabschnitt 2.1.4) beschrieben, wie ein Verband in dieser Art aufgestellt wird.

6.3.1 Unverknüpfbare Sicherheitslevel

Binär: Hoch - Niedrig Wenn es nur zwei Level gibt, zwischen denen eine Vertraulichkeitsrelation vorliegt, werden genau zwei Sicherheitslevel (High, Low) instanziiert. Ein Verband zur Speicherung der erlaubten Flüsse besitzt als einzige Kante den Fluss von Low nach High. Reflexive Kanten werden in Verbänden nicht angegeben, da sie implizit angenommen werden. Denn Informationsflüsse zwischen Bereichen, die das gleiche Sicherheitslevel besitzen, stellen keine Verletzung dar.

Geordnet Sobald eine geordnete Relation ohne Verzweigungen zwischen den Leveln vorliegt, muss ein Verband erstellt werden, der einer verketteten Liste gleicht. Ein Verband zur Speicherung der erlaubten Flüsse besitzt jeweils als Kanten die Flüsse zwischen zwei aufeinanderfolgenden Elementen. Eine Liste dieser Flüsse muss nicht über die Beschränkte Transitive Reduktion generiert werden.

Ungeordnet Liegen Sicherheitsklassen vor, die keine Hierarchie besitzen und nur zur Trennung verschiedener Berechtigungsbereiche dienen, lässt sich kein Verband bilden. Erlaubte Flüsse müssen in diesem Fall als Regeln aufgezählt werden.

6.3.2 Verknüpfbare Sicherheitslevel

Konjunktiv Ein konjunktiver Verband kann über das Verfahren der Beschränkten Transitiven Reduktion erstellt werden, wie es in Abschnitt 6.2 beschrieben wurde. Hierzu werden zuerst die Kanten zwischen den ausgeprägten kombinierten Sicherheitsleveln über die Teilmengenrelation erstellt. Danach wird ihre Orientierung im Verband so gespeichert, dass ein erlaubter Fluss von der Untermenge zur Obermenge verläuft. Der so erstellte Verband ist nicht mehr von der Potenzmenge an möglichen kombinierten Leveln abhängig. Einzelne Sicherheitslevel sind in einem konjunktiven Verband als niedrig und kombinierte Level als höher vertraulich einzuordnen. Ein erlaubter Informationsfluss verläuft somit korrekterweise von Niedrig nach Hoch.

Disjunktiv Ein disjunktiver Verband der erlaubten Flüsse kann ebenso wie ein konjunktiver Verband über das vorgestellte Verfahren der Beschränkten Transitiven Reduktion erstellt werden. Der Unterschied ist einzig die Orientierung der Teilmengenrelationskanten. Während erlaubte Flüsse im konjunktiven Fall von der Untermenge zur Obermenge verliefen, laufen erlaubte Flüsse im disjunktiven Fall von der Obermenge zur Untermenge. Denn die Obermenge aus verknüpften Sicherheitsleveln ist weniger vertraulich als die Untermenge aus einzelnen Sicherheitsleveln, da mehr Parteien berechtigt sind, darauf zuzugreifen.

7 Implementierung des iterativen Analyseansatzes

Dieses Kapitel beschäftigt sich mit der Implementierung, die im Rahmen dieses Ansatzes durchgeführt wurde. Die Implementierung unterteilt sich in drei Bereiche. Als Erstes wurde die Verbandsoptimierung angepasst. Hierfür wurde ein Algorithmus zum Erstellen der Beschränkten Transitiven Reduktion eines Verbandes implementiert, wie er in der Theorie in Abschnitt 6.2 vorgeschlagen wurde. Dies wird in Abschnitt 7.1 beschrieben. Als Zweites wird beschrieben, wie das bestehende Analyseframework, das in Abschnitt 2.3 vorgestellt wurde, erweitert wird, um die nötigen Mechanismen für eine iterative Ausführung des Ansatzes bereitzustellen. Diese Mechanismen wurden im Theorieteil aus Abschnitt 5.3 als Iterationskonstrukt, Iterator und Abbruchbedingung definiert. Ihre Umsetzung wird in Abschnitt 7.2 beschrieben. Als Drittes folgt die Implementierung von expliziten iterativen Konfigurationen für die beiden Analysen JOANA und CodeQL. Hierfür ist es nötig, die *ProcessingSteps* und *Adapter* anzupassen, die die Ausführung der statischen Analysen durch das Framework steuern. Außerdem müssen sowohl in der Hinprojektion als auch der Rückprojektion die zugrundeliegenden Analyseschritte angepasst werden, da an diesen Stellen im iterativen Ansatz z. B. andere Verknüpfungsoperatoren notwendig sind. Dies wird in Abschnitt 7.3 ausgeführt. In jedem Fall werden Entwurfsentscheidungen begründet und es wird darauf geachtet, eine erweiterbare Architektur zu entwerfen, damit sich der iterative Ansatz in Zukunft für weitere Analysen nutzen lässt.

7.1 Umsetzung der optimierten Sicherheitsverbände

Um den Sicherheitsverband auf diejenigen Sicherheitslevel und kombinierten Sicherheitslevel zu beschränken, die im eigentlichen Modell konkret ausgeprägt sind, wird dieser Verband nicht mehr über die Potenzmenge der Einzellevel erstellt. Stattdessen werden die ausgeprägten Level der Reihe nach eingelesen und in eine Datenstruktur eingetragen. Nur zwischen diesen eingelesenen Elementen werden Kanten erzeugt, die aus dem Graphen der transitiven Reduktion stammen. So ergibt sich insgesamt ein Sicherheitsverband, der eine Beschränkte Transitive Reduktion eines totalen Sicherheitsverbands ist.

Als Datenstruktur zur Repräsentation von kombinierten Leveln wird aus zwei Gründen die Struktur BitSet gewählt. Erstens belegt ein BitSet nur minimalen Speicherplatz, da jedes einzelne, enthaltene Level über eine 0 oder 1 dargestellt wird, je nachdem ob es im kombinierten Level enthalten ist oder nicht. Hierfür muss zuvor eine globale Registrierung der Indizes aus

der Binärrepräsentation erfolgen, welcher Index welches Einzellevel darstellt. Zweitens ist ein Vorteil der Datenstruktur BitSet, dass die Teilmengenüberprüfung der Teillevel zwischen zwei kombinierten Leveln, die als BitSet repräsentiert werden, sehr einfach ist. Hierfür muss nur überprüft werden, ob das *logische OR* beider BitSets gleich der potentiellen Obermenge der beiden BitSets ist. Denn eine Obermenge hat in der Binärdarstellung an mindestens allen Indizes eine 1 stehen, an denen auch die Repräsentation der Untermenge eine 1 stehen hat. Werden die Darstellungen verodert und das Ergebnis weicht von der angenommenen Obermenge ab, gibt es mindestens eine 1, die nicht in dieser enthalten war, wodurch sie keine Obermenge ist.

Zunächst wird die Klasse *CombinedLevel* erstellt, dargestellt in Abbildung 7.1, die zusätzlich zu einem BitSet zur Repräsentation auch ein HashSet an direkten Nachfolgern enthält. Über dieses werden die Methoden getDirectsuccessors() und getAllSuccessors() angeboten. Letztere Methode führt intern die eigenen direkten Nachfolger mit den rekursiv erhaltenen Nachfolgern dieser zusammen. Beide Methoden sind für den Algorithmus aus Abschnitt 6.2 nötig.

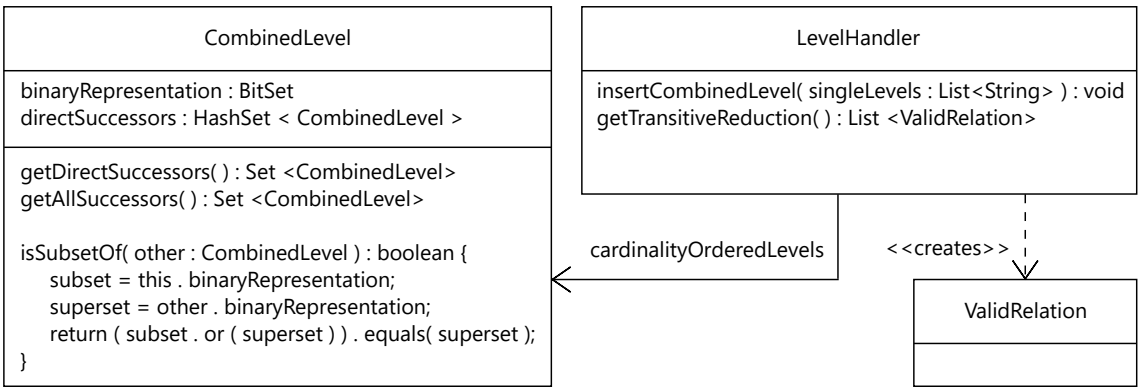


Abbildung 7.1: Klassendiagramm des Pakets *transitivereduction* zur Verbandsgenerierung

Als Nächstes wird die Klasse *LevelHandler* erstellt, ebenfalls dargestellt in Abbildung 7.1, die die eingelesenen ausgeprägten Level nach Kardinalität geordnet speichert und eine Referenzliste zwischen Indizes und deren Bezeichnern anlegt. Die Methode *insertCombinedLevel()* filtert bereits bei der Eingabe Dopplungen in den ausgeprägten Leveln, da jedes Level nur einmal im Verband auftreten soll. Das Ordnen der Level nach Kardinalitäten ist der erste Schritt im Vorgehen der Beschränkten Transitiven Reduktion. Die weiteren Schritte im Vorgehen werden beim Aufruf der Methode getTransitiveReduction() ausgeführt, die in Listing 7.1 als Quelltextausschnitt gezeigt wird. Zuerst werden die direkten Nachfolger für jedes *CombinedLevel* berechnet. Es wird mit den Leveln in der größten Kardinalität gestartet und dann abgestiegen. So besitzen für kleinere Kardinalitäten die darüber liegenden Level bereits eine Zuweisung ihrer direkten Nachfolger. Für die Berechnung der direkten Nachfolger wird in der Methode *findDirectSuccessorsOfCombinedLevel()* für ein bestimmtes Level in den Kardinalitäten ab der eigenen Startkardinalität aufgestiegen. Über die lokale Variable *alreadyReachable* wird für dieses Level aufgezeichnet, welche anderen Level bereits durch einen gefundenen direkten Nachfolger erreichbar sind. Sobald eines dieser anderen Level in der Suche an die Reihe kommt, wird es nicht als eigener direkter Nachfolger eingetragen, da

es in dieser Liste bereits enthalten ist. Ein untersuchtes Level, das in dieser Liste noch nicht enthalten ist und dessen BitSet eine Obermenge des eigenen BitSets ist, wird hingegen als eigener direkter Nachfolger hinzugefügt. Außerdem werden alle seine Nachfolger über die Methode *getAllSuccessors()* bezogen und der Erreichbarkeitsliste hinzugefügt. Nachdem für alle *CombinedLevel* die direkten Nachfolger berechnet und in ihnen abgespeichert wurden, werden die dadurch entstandenen Verbandskanten als *ValidRelations* zurückgegeben. Im Algorithmus sind diese Kanten so orientiert, dass sie von einer Untermenge an kombinierten Leveln zu einer Obermenge an kombinierten Leveln zeigen, was der disjunktiven Verknüpfung entspricht. Dreht man jede Kante jedoch um, erhält man einen konjunktiven Verband. Der Ansatz eignet sich somit für die Generierung beider Verbandsarten.

```
public List<ValidRelation> getTransitiveReduction() {
   List<ValidRelation> relations = new ArrayList<ValidRelation>();

   // successor calculation, iterate downwards
   // the superset element has no successors --> start with cardinality-1
   for (int i = this.numberOfSingleLevels - 1; i >= 0; i--) {
      if (cardinalityOrderedLevels.containsKey(i)) {
         for (CombinedLevel cLevel : cardinalityOrderedLevels.get(i)) {
            this.findDirectSuccessorsOfCombinedLevel(cLevel); }}}

   // edge creation
   for (int i = 0; i < this.numberOfSingleLevels; i++) {
      if (cardinalityOrderedLevels.containsKey(i)) {
         for (CombinedLevel from : cardinalityOrderedLevels.get(i)) {
            if (!from.getDirectSuccessors().isEmpty()) {
               for(CombinedLevel to : from.getDirectSuccessors()) {
                  relations.add(new ValidRelation(from, to)); }}}}
   }
   return relations;
}

private void findDirectSuccessorsOfCombinedLevel(CombinedLevel level) {
   Set<CombinedLevel> alreadyReachable = new HashSet<CombinedLevel>();
   int startingCardinality = level.getCardinality() + 1;

   for (int j = startingCardinality; j <= this.numberOfSingleLevels; j++) {
      if (cardinalityOrderedLevels.containsKey(j)) {
         for (CombinedLevel next : cardinalityOrderedLevels.get(j)) {
            if ((!alreadyReachable.contains(next))
                        && level.isSubsetOf(next)) {
               level.addSuccessor(next);
               alreadyReachable.add(next);
               alreadyReachable.addAll(next.getAllSuccessors()); }}}}
   }
}
```

Listing 7.1: Quelltextauszug aus der Klasse *LevelHandler* zur Transitiven Reduktionsberechnung.

7.2 Erweiterung des Analyse-Frameworks

Damit das Analyse-Framework, das in Abschnitt 2.3 vorgestellt wurde, mehrere Iterationen hintereinander ausführen kann, müssen einige Erweiterungen durchgeführt werden. Bevor diese jedoch beschrieben werden, werden zuerst die Mechanismen Iterationskonstrukt, Iterator und Abbruchbedingung aus Abschnitt 5.3 auf Klassen und deren Funktionalität übertragen.

7.2.1 Umsetzung des Iterationskonstrukts und Iterators durch Partitioner

Als Erstes wird beschrieben, wie das Konzept des Iterationskonstrukts implementiert wurde. Das Iterationskonstrukt ist definiert als eine Grundmenge an Elementen, die sich aufteilen lässt und deren Elemente unabhängig voneinander analysierbar sind.

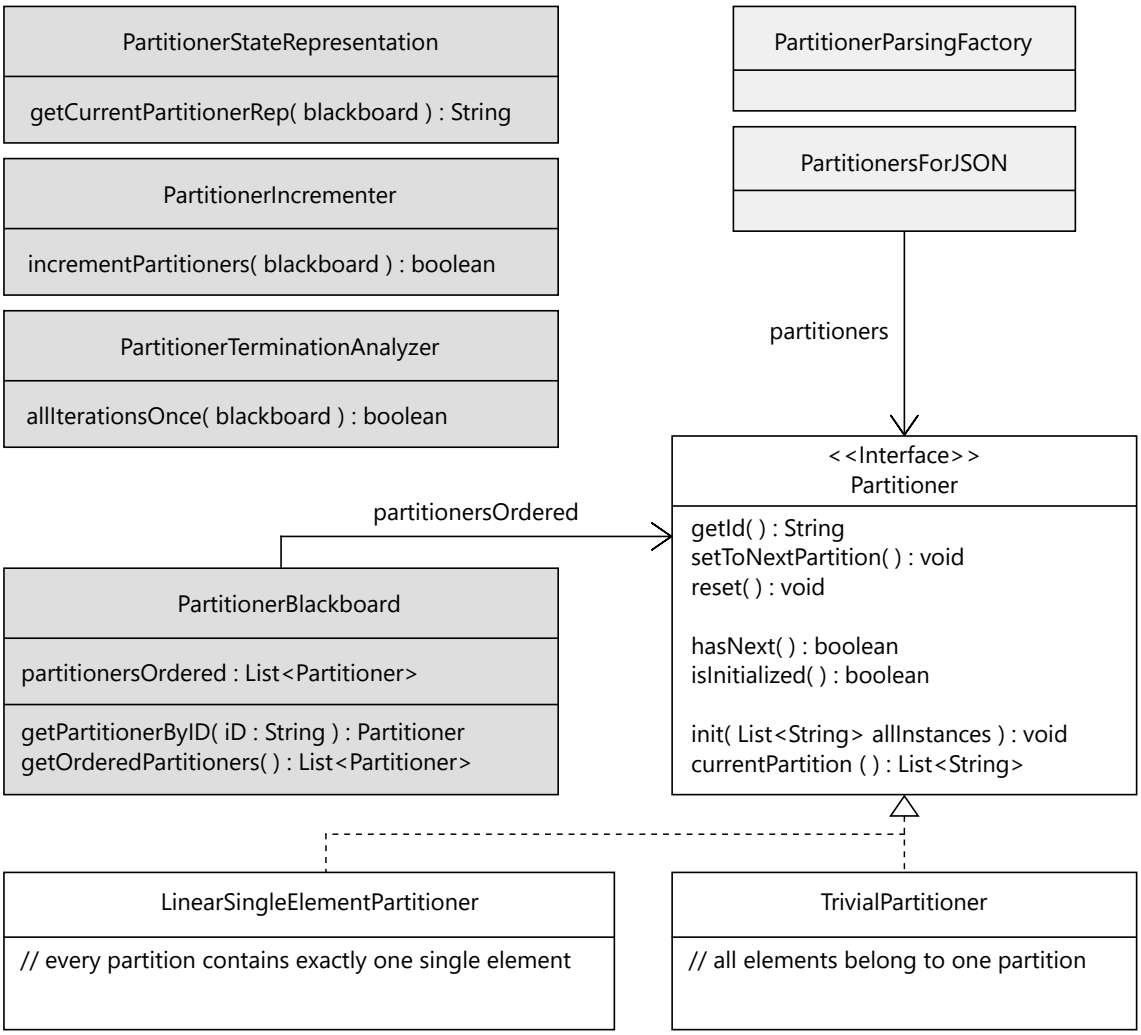


Abbildung 7.2: Übersicht der Klassen und Schnittstellen des Partitioner-Pakets

Die Schnittstelle *Partitioner* wird erstellt, dargestellt auf der rechten Seite der Abbildung 7.2, in der alle Methoden angeboten werden, um einerseits Instanzen als Grundmenge zu initialisieren und andererseits um Partitionen davon zu erhalten. Diese Schnittstelle implementieren die beiden Klassen *LinearSingleElementPartitioner* und *TrivialPartitioner*. Der *LinearSingleElementPartitioner* unterteilt die initialisierte Grundmenge an Elementen in seine Einzelteile, sodass jede Partition genau ein Element enthält. Damit auch im iterativen Framework weiterhin eine nicht-iterative Ausführung möglich bleibt, wird zusätzlich der *TrivialPartitioner* bereitgestellt, welcher die Grundmenge in einer einzigen Partition ablegt und so die Abarbeitung aller Elemente in einer einzigen Iteration zulässt.

Der *Partitioner* setzt intern den Mechanismus des Iterators um, indem ein Zeiger auf die aktuelle Partition gespeichert wird. Über die Methode *setToNextPartition()* lässt sich dieser Zeiger inkrementieren und über die Methode *reset()* wird sowohl die Grundmenge geleert als auch der Zeiger zurückgesetzt. Durch das Inkrementieren wird die nächste Partition ausgewählt und bei einem erneuten Aufruf der Methode *currentPartition()* diese zurückgegeben.

Damit der aktuelle Iterationsstand zwischen Programmunterbrechungen persistiert werden kann, wird die Speicherung der *Partitioner* als JSON-Datei angeboten. Hierfür werden alle instanziierten *Partitioner* in der Klasse *PartitionersForJSON* gesammelt und diese dann durch die *PartitionerParsingFactory* zu JSON-Objekten serialisiert. Die *PartitionerParsingFactory* ist auch für die Speicherung und das Laden solcher serialisierter Objekte verantwortlich. Bei einem Neustart des Programms können geladene JSON-Objekte deserialisiert und die entsprechenden *Partitioner* wieder weiter genutzt werden.

Die linke Seite der Abbildung 7.2 beschreibt, wie die *Partitioner* im Framework zur Verfügung gestellt werden. Es wird das Entwurfsmuster Blackboard gewählt, um eine Trennung der Funktionalität für Datenhaltung einerseits und für Operationen auf den Daten andererseits zu erzeugen und so dem Entwurfsprinzip der *Separation of Concerns* zu entsprechen. Hierfür werden die *Partitioner* auf einem globalen *PartitionerBlackboard* abgelegt, wo über ihre ID auf einzelne *Partitioner* zugegriffen werden kann. Es werden drei Funktionsklassen implementiert, die Operationen auf dem Blackboard ausführen. Die Klasse *PartitionerStateRepresentation* bietet eine Methode an, um den Zustand aller *Partitioner* als eindeutige Zeichenkette auszugeben, die zur Referenzierung für Iterationen notwendig ist. Die Klasse *PartitionerIncrementer* übernimmt die Logik, die *Partitioner* geordnet zu inkrementieren. Da im Falle von mehreren *Partitionern* alle möglichen Kombinationen ihrer Partitionen abgedeckt werden sollen, wird immer nur ein *Partitioner* solange inkrementiert, bis alle seine Partitionen einmal durchlaufen wurden. Erst wenn dieser innerste *Partitioner* wieder auf die Ursprungspartition zeigt, wird der nächste *Partitioner* in der geordneten Liste um eine Partition erhöht. Das Konzept gleicht so einer verschachtelten Schleife. Die dritte Funktionsklasse ist der *PartitionerTerminationAnalyzer*. Dieser überprüft, ob alle Iterationen einmal abgedeckt wurden, indem über die Methode *isInitialized()* abgefragt wird, ob die *Partitioner* zurückgesetzt wurden. Der Zustand, in dem alle *Partitioner* uninitialisiert sind, tritt nur vor der allerersten Iteration und nach der letzten Iteration auf und kann deshalb als solche Bedingung genutzt werden.

Als Nächstes wird das Benutzen eines *Partitioners* beschrieben. Während des Generierungsschritts des Frameworks werden für die Quelltextanalyse Artefakte erzeugt, wie z. B.

Annotationen für Flussquellen. Diese sollen in verschiedene Iterationen aufgeteilt werden. Wie Abbildung 7.3 darstellt, werden hierfür als Erstes alle möglichen Elemente generiert, im Beispiel der Abbildung alle Flussquellen. Es wird ein Aufruf zur Beschneidung dieser Gesamtmenge in den eigentlichen Programmablauf zwischengeschaltet. Der *Partitioner*, der speziell der Aufteilung der Flussquellen zugeordnet ist, wird aus dem globalen Blackboard über seine ID bezogen. Befindet sich die Ausführung in der ersten Iteration, ist dieser *Partitioner* noch nicht initialisiert und erhält alle möglichen Namen der Elemente zur Initialisierung. Diese Namen müssen eindeutig sein. Die Wahl des *String* als Typ zur Referenzierung liegt daran, dass für die Serialisierung der *Partitioner* keine komplexen Objekte infrage kommen. Außerdem sollen die *Partitioner* eine leichtgewichtige Datenhaltung erreichen. Die Zeichenketten zur Identifizierung der Elemente können z. B. *UUIDs* sein oder für Sicherheitslevel z. B. deren eindeutig verkettete Repräsentation im Format [Airline;User;TravelAgency].

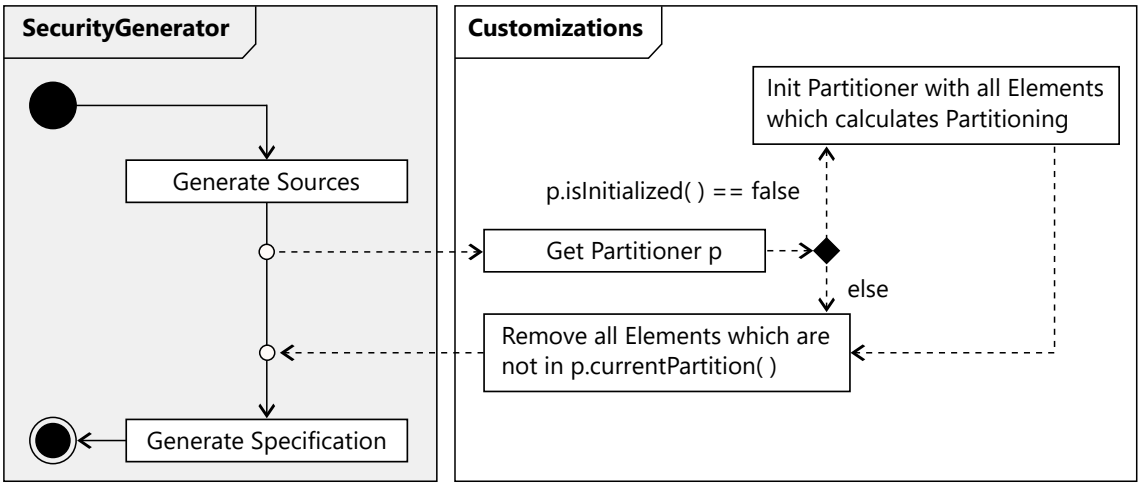


Abbildung 7.3: Zwischenschaltung der Nutzung eines Partitioners im SecurityGenerator

Liegt entweder bereits eine initialisierte Grundmenge vor, oder wurde sie frisch initialisiert, wird im nächsten Schritt die Beschneidung vorgenommen. Hierfür wird die aktuelle Partition abgefragt und alle generierten Elemente wieder gelöscht, deren ID nicht in dieser Partition enthalten ist. Anschließend wird die normale Programmausführung fortgesetzt. An welchen Stellen im Gesamtablauf des Frameworks die Partitioner inkrementiert werden, folgt nach Beschreibung der Abbruchbedingungen.

7.2.2 Umsetzung der Abbruchbedingung als Besuchermuster

Eine Abbruchbedingung terminiert die Ausführung. Wie in Unterabschnitt 5.3.3 beschrieben, werden zwei Abbruchbedingungen implementiert: Als Erstes die vollständige Abdeckung des Iterationskonstrukts und als Zweites die Stabilisierung des Ergebnismodells. Es wird zuerst die abstrakte Oberklasse *AbstractTerminationCondition* mit der Methode *isFulfilled()* bereitgestellt. In dieser Methode sollen Unterklassen über das Besucher-Entwurfsmuster eine Überprüfung an einer Framework-Komponente durchführen, die ihrem entsprechenden

Zweck dient. Als Erstes wird mit der Unterklasse *EveryIterationOnceTerminationCondition* der Zweck der vollständigen Abdeckung implementiert.

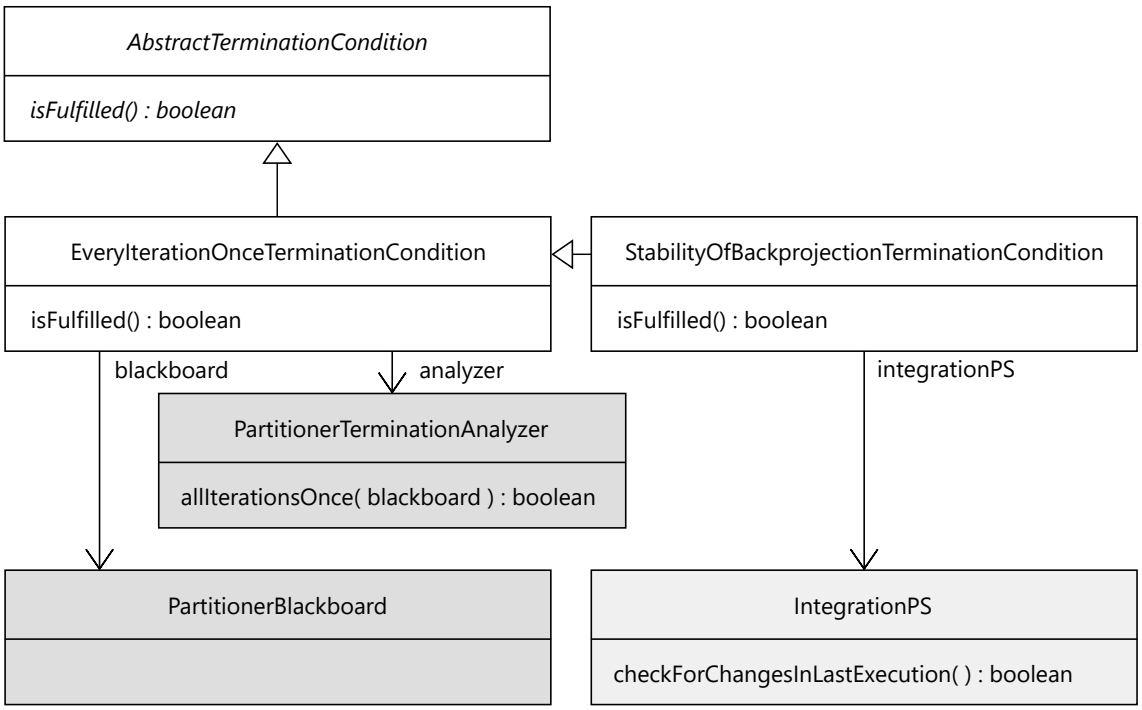


Abbildung 7.4: Übersicht der Klassen des Pakets *terminationcondition* für Abbruchbedingungen

Diese Unterklasse besitzt dafür eine Referenz auf das globale *PartitionerBlackboard*, um darauf zu testen, ob alle Iterationen vollständig durchlaufen wurden. Den Test führt ein eigener *PartitionerTerminationAnalyzer* aus, der als Funktionsklasse in Unterabschnitt 7.2.1 beschrieben wurde.

Als zweite Unterklasse wird mit der *StabilityOfBackprojectionTerminationCondition* die Abbruchbedingung für Modellstabilität bereitgestellt. Sie besitzt für das Besuchermuster eine Referenz auf den *IntegrationPS*. Dieser *ProcessingStep* ist derjenige, der bei der Ausführung die Ergebnisse in das Architekturmodell zurückprojiziert und aufzeichnen kann, ob es bei seiner letzten Ausführung zu Veränderungen kam. Wenn dies der Fall ist, wird beim Aufruf der Methode *checkForChangesInLastExecution()* der Wert *true* zurückgegeben. Da trotzdem zuerst eine vollständige Ausführung aller Iterationen abgewartet werden muss, erbt diese Abbruchbedingung nicht direkt von der abstrakten Klasse *AbstractTerminationCondition*, sondern von deren Unterklasse *EveryIterationOnceTerminationCondition*. So kann sie beide Bedingungen miteinander verknüpfen.

7.2.3 Erweiterung des Frameworks durch iterative Komponenten

Um die Mechanismen zur iterativen Durchführung zu nutzen, werden Erweiterungsklassen in einer iterativen Version des Frameworks implementiert. Abbildung 7.5 stellt die Abläufe und Interaktionen zwischen den Komponenten während der iterativen Ausführung dar.

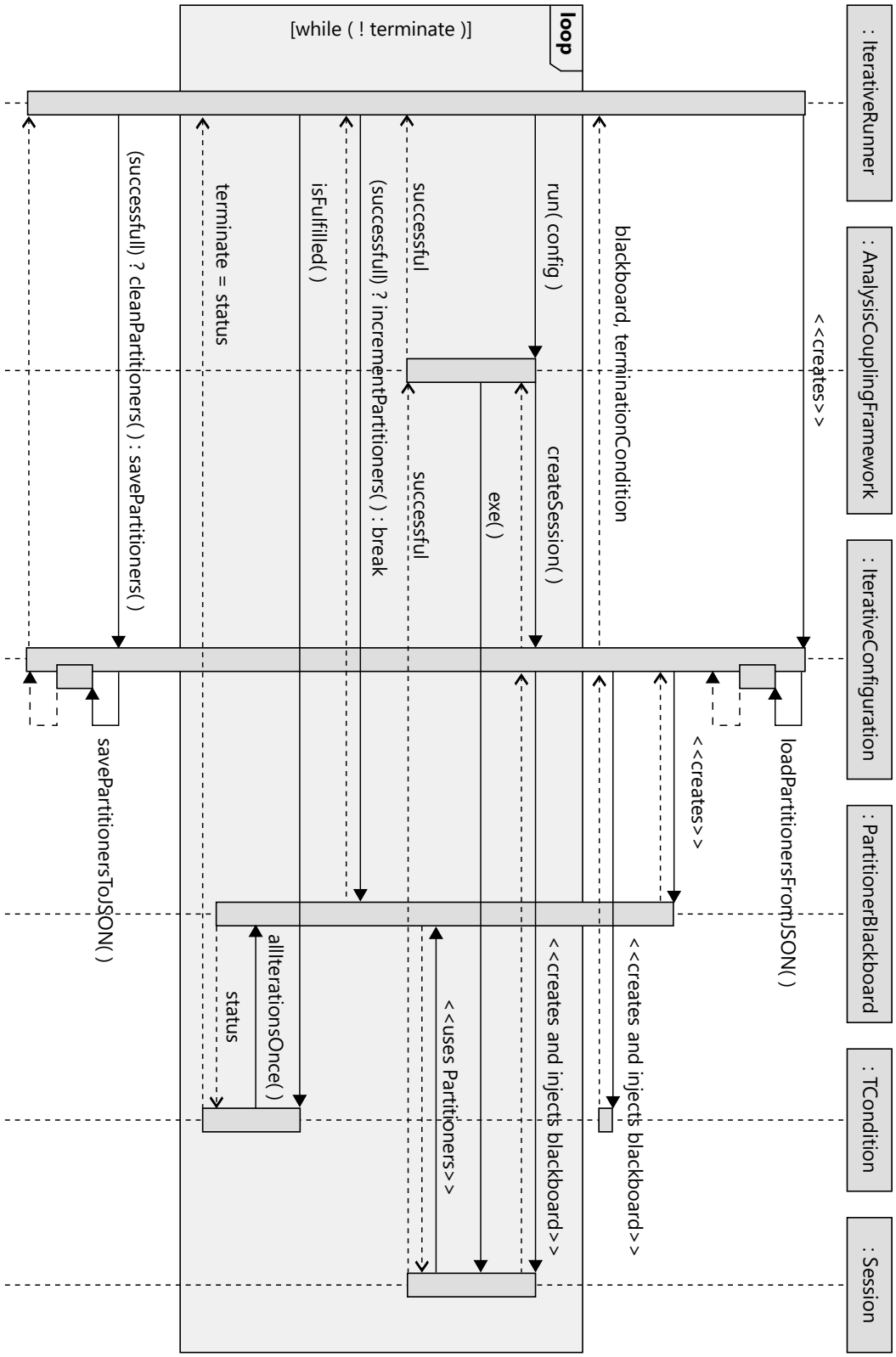


Abbildung 7.5: UML-Sequenzdiagramm der Ausführung des iterativen Frameworks

- **IterativeRunner:** Diese Klasse erweitert den *AbstractRunner* und ruft ebenfalls das Framework mit einer *Configuration* auf, in diesem Fall jedoch mit der Unterklasse *IterativeConfiguration*.

  Für jede einzelne Iteration, die in Abbildung 7.5 als Schleife gekennzeichnet ist, wird über das *AnalysisCouplingFramework* eine neue *Session* erstellt und diese ausgeführt. Wenn sie *successful* zurückgibt, wird das *PartitionerBlackboard* inkrementiert und die nächste *Session* gestartet. Im Fall (*successful* == *false*) wird die Ausführung abgebrochen und die *Partitioner* werden in die JSON-Datei zurückgeschrieben. Am Ende jedes Schleifendurchlaufs wird die Abbruchbedingung aus der *TerminationCondition* überprüft. Wird der Schleifendurchlauf durch die Abbruchbedingung und nicht einen Fehler der Ausführung beendet, werden die *Partitioner* zurückgesetzt und die gespeicherte JSON-Datei bereinigt, sodass beim nächsten Start wieder von vorne begonnen wird.

- **IterativeConfiguration:** Diese Klasse erweitert die Klasse *Configuration* mit der Logik, zusätzlich zu der XML-Datei auch eine JSON-Datei zu laden und zu speichern, in der die *Partitioner* zwischen zwei Iterationen im Fehlerfall persistiert werden. (De-) Serialisierung der *Partitioner* wird in der *PartitionerParsingFactory* durchgeführt. Der Typ eines *Partitioners* wird in der Deserialisierung über die Annotationen *@JsonSubTypes* wieder auf die richtige Instanz zugeordnet. Alle Implementierungen müssen als eine solche Annotation im Interface *Partitioner* registriert werden.

- **PartitionerBlackboard:** Diese Klasse wurde bereits in Abbildung 7.3 beschrieben. Sie dient als globale Ablagefläche der geladenen *Partitioner*-Instanzen. Zu Beginn werden diese Partitioner von der IterativeConfiguration eingelesen, instanziiert und anschließend auf das Blackboard gelegt. Das Blackboard wird dann in diejenigen *Sessions* und deren *ProcessingSteps* injiziert, die die Partitionierung betreffen und mit den *Partitionern* arbeiten (z. B. Step 1: ALIGNMENT). Außerdem löst der *IterativeRunner* über eine Referenz auf das Blackboard die Inkrementierung der *Partitioner* aus.

- **TerminationCondition:** Diese Klasse wurde bereits in Unterabschnitt 7.2.2 beschrieben und stellt die Abbruchbedingung für den Schleifendurchlauf dar. Sie erhält durch ihren Konstruktor das *PartitonerBlackboard* und führt jedes Mal, wenn sie über den Aufruf isFulfilled() getestet werden soll, eine Prüfung des Blackboards durch, um ihren Wahrheitswert zu aktualisieren.

Damit die *ProcessingSteps* innerhalb einer *Session* das Blackboard benutzen können, muss es über die *IterativeConfiguration* beim Erstellen einer *ProcessingStrategy* in die *ProcessingSteps* injiziert werden. Neben der Schnittstelle *ExecutableProcessingStepAdapter*, die in Abschnitt 2.3 zu den Grundlagen des Frameworks beschrieben wurde, wird zusätzlich die abstrakte Klasse *IterativeExecutableProcessingStepAdapter* für Adapter erstellt, um die Referenz auf das Blackboard einzuführen. In Abbildung 7.6 wird am Beispiel für JOANA verglichen, welche Klassen im nicht-iterativen Fall (hellgrau) und welche im iterativen Fall (dunkelgrau) in der *ProcessingStrategy* benötigt werden. Da das Blackboard im iterativen Fall im *ALIGNMENT*-Schritt gebraucht wird, nutzt die Klasse *IterativeJoanaAlignmentPS* einen *IterativeExecutableProcessingStepAdapter* statt eines normalen Adapters.

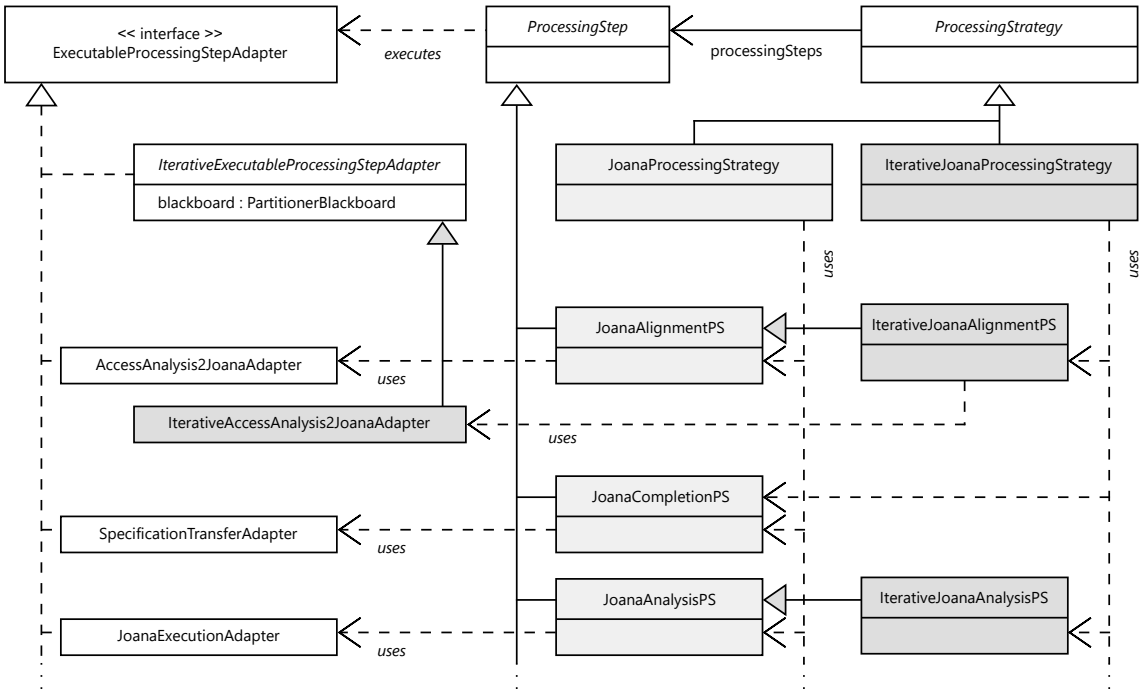


Abbildung 7.6: Anpassung der Adapter im iterativen Fall am Beispiel JOANA. Die nicht-iterativen Klassen sind in hellgrau, die iterativen in dunkelgrau dargestellt. Eine Referenz auf das Blackboard wird nur im iterativen ersten *ProcessingStep*, dem *IterativeJoanaAlignmentPS* benötigt.

7.3 Framework-Instanzen für konkrete Konfigurationen

Die vorgestellte Implementierung der Mechanismen und der Framework-Erweiterung wird dazu genutzt, um Instanzen des iterativen Ansatzes zu erstellen. Es wird in dieser Arbeit die Iterationsart implementiert, die in Abschnitt 5.4 ausgewählt wurde. Als Iterationskonstrukt wird für die Konfigurationen deshalb das Konstrukt *Sicherheitslevel-Quellenpartitionierung* gewählt. In jeder Iteration wird ein spezielles kombiniertes Sicherheitslevel untersucht. Die Annotationen im Quelltextmodell werden während ihrer Generierung so beschnitten, dass nur noch diejenigen Parameter als Quellen annotiert werden, die dieses Sicherheitslevel haben. Hierfür wird ein *Partitioner* genutzt, der mit den Repräsentationen der Level im Format [Airline;User;TravelAgency] initialisiert wird. Als Abbruchbedingung wird die vollständige Abdeckung des Iterationskonstrukts gewählt. Hierfür wird die Klasse *EveryIterationOnceTerminationCondition* genutzt und dem *IterativeRunner* als *TerminationCondition* übergeben.

Es werden vier Konfigurationen erstellt:

- **(JOANA-non-it):** Für den Vergleich zu Härings Ansatz [9] wird im Framework eine nicht-iterative Konfiguration angelegt, in der JOANA als Quelltextanalyse eingebunden wird. Für die Verbandsgenerierung wird der naive Ansatz genutzt, wie in Abschnitt 6.1 beschrieben. In der Ergebnisauflösungsphase werden bei gefundenen

illegalen Flüssen die Quellenlevel über den Quellendisjunktionsoperator verbunden und ergeben so das rückprojizierte neue Level für die Senke.

- **(JOANA-it):** Aufbauend auf der ersten Konfiguration wird eine iterative Variante angelegt, die alle Adapter übernimmt, aber den ersten *ProcessingStep*, der für die Generierung der Quelltextannotationen zuständig ist, erweitert. Die Nutzung des Partitioners wird in Unterabschnitt 7.3.1 beschrieben. Für die Verbandsgenerierung wird die Beschränkte Transitive Reduktion des *transitivereduction* Pakets genutzt, die in Abschnitt 7.1 beschrieben wurde. Weil immer nur ein Ursprungslevel in einer Iteration betrachtet wird, müssen in der Ergebnisauflösungsphase nie zwei verschiedene Level auf eine Senke zurückprojiziert werden. So wird bei einem gefundenen illegalen Fluss direkt das Senkenlevel durch das Quellenlevel ersetzt.
- **(CodeQL-non-it):** Als weitere Quelltextanalyse wird CodeQL ausgewählt. In der Konfiguration *CodeQL-non-it* wird im Framework eine nicht-iterative Ausführung implementiert. CodeQL führt dafür Abfragen auf einer Datenbank des Quelltexts aus [40]. Für die Verbandsgenerierung wird wie in Konfiguration *JOANA-non-it* der naive Ansatz genutzt. Als Verknüpfungsoperator wird die Quellendisjunktion genutzt.
- **(CodeQL-it):** Diese Konfiguration baut auf der dritten auf, aber passt die Generierung der CodeQL-Datenbankabfrage an. Dies wird zusammen mit der Nutzung des Partitioners, der die Selektion der Level für eine Iteration durchführt, in Unterabschnitt 7.3.2 beschrieben. Für die Verbandsgenerierung wird wie für Konfiguration *JOANA-it* die Beschränkte Transitive Reduktion des *transitivereduction* Pakets genutzt. Auch in der Rückprojektion wird analog zur Konfiguration *JOANA-it* nicht der Quellendisjunktionsoperator genutzt, sondern das Senkenlevel einfach durch das Quellenlevel ersetzt.

7.3.1 Partitionierung bei JOANA

Die Quelltextanalyse JOANA benötigt einen Einstieg in den Quelltext, von dem aus Informationsflüsse im Programmabhängigkeitsgraphen gesucht werden [37]. Für diesen Einstieg muss eine *Entrypoint*-Annotation im Quelltext eingefügt werden. Im nicht-iterativen Ansatz wird für jeden Parameter auf diese Art ein eigener *Entrypoint* an seine zugehörige Methode gesetzt. Zusätzlich wird für jeden Parameter eine *Source*- und eine *Sink*-Annotation hinzugefügt, damit JOANA Flüsse untersucht, die zu diesem Parameter fließen oder von diesem Parameter entspringen.

Entrypoints besitzen einen Identifizierungs-*Tag*, der für diese Arbeit als aufsteigende Nummer gewählt wird. Zu jedem *Entrypoint* wird genau eine *Source* über den gleichen *Tag* zugeordnet. Es werden in der Ausführung eines einzelnen *Entrypoints* somit genau die Flüsse untersucht, die von dem Parameter ausgehen, an dem der *Entrypoint* und die *Source* annotiert sind. Zusätzlich muss der *Tag* eines *Entrypoints* in die *Sink*-Annotationen jedes anderen Parameters hinzugefügt werden, damit alle Flusskombinationen zu diesen Stellen hin untersucht werden.

Im iterativen Ansatz wird JOANA nur für diejenigen *Entrypoints* ausgeführt, deren Nummern in einer externen Textdatei aufgelistet werden. So ist es möglich, über die Auswahl dieser Nummern nur einen Teil der *Entrypoints* und somit auch nur einen Teil der *Sources* zu analysieren. Listing 7.2 zeigt, wie die Liste an Nummern generiert wird. Ein *TrivialSingleElementPartitioner* wird im ersten Schritt mit allen ausgeprägten Leveln initialisiert, die über alle *Entrypoints* aus der Eingabeliste *entrypoints* gefunden werden. Als Nächstes wird eine Liste *currentIDs* über die aktuelle Partition des *Partitioners* bezogen. Im letzten Schritt werden diejenigen *Entrypoints* herausgefiltert, die eine *Source* mit einem der Level der *currentIDs* besitzen. Für diese *Entrypoints* wird der *Tag* in die Ergebnisliste mit aufgenommen.

```
private String selectSourcesWithCertainSecurityLevelForCurrentIteration(
      List<EntryPoint> entrypoints) {
   Partitioner levelPartitioner = this.blackboard.getPartitionerByID("p_lvl");

   if (!levelPartitioner.isInitialized()) {
      List<String> allIDs = entrypoints.stream()
            .map(e -> e.getLevel().stream().map(l -> l.getName()))
            .flatMap(s -> s).distinct().collect(Collectors.toList());

      levelPartitioner.init(allIDs);
   }

   List<String> currentIDs = levelPartitioner.currentPartition();

   String generatedEntryPointIDsAsString = "";
   for (EntryPoint e : entrypoints) {
      if (e.getAnnotation().stream()
            .filter(a -> a instanceof Source
                  && currentIDs.contains(a.getLevel().getName()))
            .count() > 0) {
         generatedEntryPointIDsAsString += e.getId() + System.lineSeparator();
      }
   }
   return generatedEntryPointIDsAsString;
}
```

Listing 7.2: Quelltextauszug aus der Klasse `IterativeAccessAnalysis2JOANASecurityGenerator` zur Auswahl der *Entrypoints* für eine Iteration.

7.3.2 Partitionierung bei CodeQL

Die Quelltextanalyse CodeQL führt Abfragen auf einer Datenbank des Quelltexts aus [40]. Es werden Flüsse zwischen denjenigen Elementen des Quelltexts gesucht, die in der Abfrage in der Query *getLabel()* aufgezählt sind.

In der iterativen Ausführung wird die Generierung der CodeQL-Datenbankabfrage angepasst. In dieser Abfrage werden die Parameter statt in die Query *getLabel()* in zwei

feingranularere Queries eingetragen: *getSourceLabel()* und *getSinkLabel()*. So kann für diejenigen Parameter, die nicht das Sicherheitslevel der aktuellen Iteration besitzen, auf das Hinzufügen in die Query *getSourceLabel()* verzichtet werden. In der Auswertung auf der Datenbank werden so nur die Flüsse untersucht, die von den verbleibenden Quellparametern ausgehen. Listing 7.3 zeigt, wie die Liste von Annotationen beschnitten wird. Ein *TrivialSingleElementPartitioner* wird im ersten Schritt mit allen ausgeprägten Leveln initialisiert, die über alle *Source*-Annotationen aus der Eingabeliste *annotations* gefunden werden. Als Nächstes wird eine Liste *currentIDs* über die aktuelle Partition des *Partitioners* bezogen. Zuletzt werden diejenigen Annotationen gelöscht, die vom Typ *Source* sind und kein Level der *currentIDs* besitzen.

```
private void selectSourcesWithCertainSecurityLevelForCurrentIteration(
      List<SecurityLevelAnnotation> annotations) {
   Partitioner levelPartitioner = this.blackboard.getPartitionerByID("p_lvl");

   if (!levelPartitioner.isInitialized()) {
      List<String> allIDs = annotations.stream()
            .filter(a -> a instanceof Source)
            .map(s -> s.getSecurityLevel().getName())
            .distinct().collect(Collectors.toList());

      levelPartitioner.init(allIDs);
   }

   List<String> currentIDs = levelPartitioner.currentPartition();

   // remove all source annotations that have not such a level
   annotations.removeIf(a -> ((a instanceof Source)
         && !currentIDs.contains(a.getSecurityLevel().getName())));
}
```

Listing 7.3: Quelltextauszug aus der Klasse `IterativeAccessAnalysis2CodeQLSecurityGenerator` zur Auswahl der Quellen für eine Iteration.

8 Evaluation

In diesem Kapitel werden der entworfene, iterative Analyseansatz und die Verbandsoptimierung evaluiert. Es wird eine fallstudienbasierte Evaluation durchgeführt, da diese Evaluationsart von vergleichbaren Arbeiten [9] [3] [8] ebenfalls verwendet wird. Hierfür wird die Fallstudie *TravelPlanner* von Katkalov et al. [8] übernommen. Das dort genutzte Reiseplanungssystem wurde als fortlaufendes Beispiel bereits in Kapitel 4 vorgestellt. Das System wird um eine prototypische Implementierung erweitert, durch die Informationsflüsse zwischen den Modellparametern entstehen. Durch das gezielte Einfügen von illegalen Informationsflüssen in die Implementierung wird die Vertraulichkeit des Systems beeinträchtigt. Es wird ein Goldstandard entworfen, der beschreibt, welche Verletzungen der Vertraulichkeit durch die eingefügten Flüsse im Architekturmodell entstehen sollen. Anhand dieses Goldstandards wird im Anschluss der iterative Analyseansatz mit dem nicht-iterativen Ansatz verglichen. Sowohl für den nicht-iterativen als auch den iterativen Ansatz werden als zugrundeliegende Quelltextanalyse jeweils JOANA [37] und CodeQL [40] eingebunden. In allen Fällen wird als Architekturanalyse in der Analysenkopplung die Access Analysis von Kramer [5] genutzt, um die Vertraulichkeit der rückprojizierten Modelle auf Architekturebene zu untersuchen.

Im Folgenden werden in Abschnitt 8.1 zuerst die Evaluationsziele und Metriken mittels des GQM-Ansatzes [63] präsentiert. Der Aufbau der Fallstudie wird in Abschnitt 8.2 und das Vorgehen der Messungen in Abschnitt 8.3 beschrieben. Die Ergebnisse werden in Abschnitt 8.4 aufgeführt und diskutiert. Auf Annahmen und Einschränkungen der Validität wird zuletzt in Abschnitt 8.6 eingegangen.

8.1 GQM

Für die Evaluation wird der GQM-Ansatz von Basili und Weiss [63] [64] genutzt. Hierfür werden im Folgenden die Ziele (**G**oals) der Evaluation beschrieben und in Fragen (**Q**uestions) und deren Metriken (**M**etrics) aufgegliedert.

Der Beitrag dieser Arbeit ist das Entwerfen und Umsetzen einer iterativen Quelltextanalyse für Informationsflusssicherheit zur Überprüfung von Vertraulichkeit auf Architekturebene. In der Konzeption des iterativen Ansatzes in Abschnitt 5.1 wurden als Verbesserungsziele, die mit einem solchen iterativen Ansatz angestrebt werden, die Genauigkeit, Performanz und Skalierbarkeit genannt. Es wurde daraufhin in Abschnitt 5.4 eine konkrete Konfiguration für eine iterative Analysenausführung entworfen und deren Implementierung in Kapitel 7

beschrieben. Diese iterative Ausführung soll bezüglich der Ziele mit der nicht-iterativen Ausführung verglichen und die Vor- und Nachteile im Anschluss diskutiert werden.

Als Hauptziele zur Bewertung des Ansatzes werden die Ziele (G1) bis (G3) formuliert. Sie werden im Anschluss begründet und ihre zugehörigen Fragen und Metriken beschrieben.

- **(G1)** Verbesserung der Genauigkeit der Vertraulichkeitsüberprüfung auf Architekturebene durch den iterativen Ansatz gegenüber dem nicht-iterativen.
- **(G2)** Verbesserung der Performanz des iterativen Quelltextanalyseansatzes gegenüber dem nicht-iterativen.
- **(G3)** Verbesserung der Skalierbarkeit durch die vorgestellte Verbandsgenerierung gegenüber dem naiven Ansatz.

Das erste genannte Ziel ist die Verbesserung der Genauigkeit (G1). In einem Artikel von Hammer und Snelting [65] wird für Programmanalysen abgeleitet, dass sich als Anforderungen die beiden Begriffe der Korrektheit und der Präzision eignen. Die Korrektheit besagt, dass sobald eine Verletzung im Quelltext vorliegt, eine Analyse diese finden muss. Die Präzision wiederum besagt, dass es keinen *Falschen Alarm* durch die Analyse geben darf. Ein *Falscher Alarm* bedeutet, dass eine Verletzung angezeigt wird, obwohl keine vorliegt. Wenn eine präzise Analyse ein Risiko feststellt, muss das Programm somit auch ein Sicherheitsleck enthalten. Hammer beschreibt jedoch, dass es aufgrund von Entscheidungsproblemen in der Informationsflusskontrolle keine totale Präzision geben kann, wenn die Korrektheit beibehalten werden soll [65]. Deshalb sei es für Ansätze wichtiger, eine *Konservative Approximation* durchzuführen und eher einen *Falschen Alarm* zu produzieren, als ein potentielles Risiko zu vernachlässigen.

Um (G1) zu evaluieren, werden somit folgende Fragen gestellt:

- **(Q1.1)** Verbessert der iterative Ansatz die Korrektheit gegenüber dem nicht-iterativen Ansatz?
- **(Q1.2)** Verbessert der iterative Ansatz die Präzision gegenüber dem nicht-iterativen Ansatz?

Als Metrik für die Korrektheit, wie sie im vorherigen Absatz definiert wurde, eignet sich für (Q1.1) die allgemein gebräuchliche, statistische Metrik der Sensitivität (Recall r) [66], da sie die Vollständigkeit an positiven Vorhersagen misst. Vollständigkeit bedeutet, dass alle Verletzungen abgedeckt werden, die ein Goldstandard vorschreibt. Sie wird über $r = \frac{TP}{TP+FN}$ mit der Anzahl an korrekt Positiven TP, falsch Positiven FP und falsch Negativen FN gefundenen Vertraulichkeitsverletzungen gegenüber dem Goldstandard berechnet. Hierbei ist ein TP eine Vertraulichkeitsverletzung durch einen bestimmten Angreifer, die im Goldstandard enthalten ist und auch durch die Analyse ausgelöst und erkannt wird. Ein FP ist ein *Falscher Alarm*, bei dem die Analyse eine Verletzung erkennt, aber der Goldstandard vorschreibt, dass eine derartige Verletzung im System nicht abgeleitet werden kann. Ein FN ist eine Verletzung, die im Goldstandard enthalten ist, aber von der Analyse nicht gefunden wird. Zur Beantwortung von (Q1.2) wird die statistische Metrik der Präzision (Precision p) [66]

genutzt, die über $p = \frac{TP}{TP+FP}$ aus den Analyseergebnissen und dem Goldstandard berechnet wird. Die Definition für TP und FP in der Präzision sind analog zu denen für die Sensitivität definiert.

Das zweite Ziel (G2) betrifft die Performanz. Da in Härings Ansatz zur Analysenkopplung besonders die Größe von erzeugten Eingabeartefakten für die Analyse als limitierender Faktor erkannt wurde [9], wird der Arbeitsspeicherverbrauch in der ersten Frage (Q2.1) herangezogen. Als Zweites wird die Dauer der Ausführung untersucht (Q2.2), da beim Aufteilen einer einzelnen Analyse auf mehrere Iterationen eine Veränderung der Laufzeit durch einen Zusatzaufwand von z. B. mehrfach ausgeführten Kompiliervorgängen erwartet wird. Auch die Kopplung mit der Architekturanalyse, die die Ergebnisse der iterativen Quelltextanalyse weiterverarbeitet, ist betroffen, da die Architekturanalyse auf allen separaten Ergebnismodellen der Iterationen einzeln ausgeführt werden muss. Wie hoch ein solcher Mehraufwand ist, soll die dritte Frage (Q2.3) beantworten. Als Viertes wird die Größe der zu speichernden Modelle untersucht, da durch den iterativen Ansatz durch getrennte Rückprojektionen mehr Modelldateien entstehen, als im nicht-iterativen Ansatz. Ob der Speicherplatzverbrauch sich stark verändert, wird in der vierten Frage (Q2.4) betrachtet.

Um (G2) zu evaluieren, werden somit folgende Fragen gestellt:

- **(Q2.1)** Verringert sich der Arbeitsspeicherverbrauch der Analyse durch den iterativen Ansatz gegenüber dem nicht-iterativen Ansatz?
- **(Q2.2)** Verringert sich die Laufzeit der Quelltextanalysendurchführung durch den iterativen Ansatz gegenüber dem nicht-iterativen Ansatz?
- **(Q2.3)** Verringert sich die Laufzeit der gesamten Kopplung von Quelltext- und Architekturanalyse durch den iterativen Ansatz gegenüber dem nicht-iterativen Ansatz?
- **(Q2.4)** Wie verändert sich der Speicherverbrauch des Ergebnismodells bei Nutzung des iterativen Ansatzes gegenüber dem nicht-iterativen Ansatz?

Zur Beantwortung der Frage (Q2.1) wird der maximale Arbeitsspeicherverbrauch des Analyseprozesses über die Dauer seiner Ausführung in der Einheit *MB* als technische Metrik genutzt. Der Verbrauch wird automatisch über ein Werkzeug [67] aufgezeichnet und durch mehrere Ausführungen auf Ausreißer (stark abweichende Werte) überprüft. Eine genauere Beschreibung des Vorgehens folgt in Unterabschnitt 8.3.2. Für Frage (Q2.2) wird als Metrik die Programmlaufzeit der Quelltextanalysen herangezogen. Sowohl im nicht-iterativen als auch im iterativen Ausführungsfall wird die Laufzeit als Zeit in Millisekunden vom Start des Frameworks bis hin zur Terminierung der Rückprojektion durch das Framework definiert. Insbesondere im iterativen Ausführungsfall sind somit alle Iterationen in der Frameworkausführung enthalten. Für Frage (Q2.3) wird zusätzlich zu der gemessenen Laufzeit aus (Q2.2) die Zeit addiert, die durch die Ausführung der Architekturanalyse hinzukommt. Zuletzt wird als Metrik für Frage (Q2.4) der gesamte Speicherverbrauch in *KB* aller Ergebnismodelldateien des *TravelPlanners* aufsummiert.

Das dritte Ziel behandelt die Skalierbarkeit. Hammer und Snelting begründen die Wichtigkeit dieses Ziels damit, dass ein Analyseansatz in der Praxis auch realistische Programme

verarbeiten muss, die z. B. 100 kLOC besitzen [65]. Da in Abschnitt 5.1 die Generierung der Eingabeartefakte für die Quelltextanalyse als Engstelle für die Skalierbarkeit erkannt wurde, soll die Verbandsgenerierung des iterativen Ansatzes das Untersuchungsobjekt für diese Frage sein.

Um (G3) zu evaluieren, wird somit folgende Frage gestellt:

- **(Q3.1)** Verbessert der Ansatz der Beschränkten Transitiven Reduktion bei großen Eingaben die Skalierbarkeit der Verbandsgenerierung?

Für die Frage (Q3.1) wird als Metrik die Programmlaufzeit der Verbandsgenerierung für größer werdende Eingaben sowohl mit dem neuen Ansatz als auch mit Härings Ansatz [9] gemessen und die Differenz als Metrik genutzt.

8.2 Aufbau der Fallstudie

Als Eingabemodell für die Analysen wird das Reiseplanungssystem *TravelPlanner* [8] genutzt, das in Kapitel 4 als fortlaufendes Beispiel eingeführt wurde und bereits in Ansätzen von Häring [9] und Seifermann [3] zu Evaluationszwecken herangezogen wurde. Es setzt sich aus den PCM-Modellinstanzen auf Architekturebene und einem Quelltextmodell auf Quelltextebene zusammen. Letzteres besteht aus einer prototypischen Implementierung in Java, die je nach eingebundener Quelltextanalyse weiterverarbeitet wird. Im Falle von JOANA werden die Klassen mit Annotationen für Flussquellen und Senken angereichert [37], im Falle von CodeQL wird eine Datenbank von Abhängigkeiten und eine Abfrage für diese Datenbank aufgebaut [40].

Abbildung 8.1 stellt in einem UML-Sequenzdiagramm das implementierte, erweiterte Nutzungsszenario des *TravelPlanner*-Systems dar. Es werden Interaktionen zwischen den Systemkomponenten beim Einholen von Flugangeboten und dem darauf folgenden Buchen eines Fluges gezeigt, die durch Eingaben an das UserInterface (UI) ausgelöst werden. Zuerst wird der Aufruf *requestOffers()* von dem UI an den TravelPlanner, von dort an die TravelAgency und von dort an die Airline delegiert. Dort werden die Parameter *discount* und *ratings* verarbeitet und deren Informationen durch den Aufruf des AirlineLoggers protokolliert. Hier befinden sich die ersten beiden riskanten Stellen bezüglich der Vertraulichkeit im Ablauf. Denn wenn die Daten nicht deklassifiziert werden (z. B. durch Verschlüsselung in der Implementierung) gelangen sie im Klartext in die Log-Einträge des AirlineLoggers. Wenn ein Angreifer auf den Ort des AirlineLoggers zugreifen kann, erhält er Zugriff zu vertraulichen Daten, für die er eventuell keine Zugriffsberechtigung besitzt. Der zweite Aufruf des Nutzers löst den Buchungsprozess eines ausgewählten Fluges aus. Zuerst wird hierbei das CreditCardCenter mit dem Aufruf *releaseCCD()* darum gebeten, die CreditCardDetails freizugeben. Hierfür muss der Nutzer über das UI die Freigabe erteilen. Im Anschluss sendet das CreditCardCenter die deklassifizierten CreditCardDetails an die Airline weiter. Da die CreditCardDetails vertraulich sind und ihnen nur das Sicherheitslevel User zugeordnet ist, befindet sich hier eine weitere riskante Stelle im System. Sollten durch eine fehlende Deklassifizierung private Nutzerdaten weitergegeben werden, ist die Vertraulichkeit

verletzt. Wie bereits beim ersten Aufruf der Airline, werden auch beim zweiten Aufruf Log-Einträge erstellt, was aus den gleichen Gründen riskant bezüglich der Vertraulichkeit ist. Im Anschluss zahlt die Airline durch den Aufruf *payCommission()* eine Vergütung an die Travelagency und bestätigt der TravelPlanner-App den Buchungsvorgang.

8.2.1 Äquivalenzklassen illegaler Informationsflüsse

Es werden nun in diese prototypische Implementierung an die genannten riskanten Stellen gezielt Informationsflüsse zwischen Parametern eingefügt. Diese Informationsflüsse sollen inkonsistent zu den spezifizierten Vertraulichkeitsleveln aus der Architektur sein, sodass jeweils ein Fluss von höher vertraulichen Quellen zu niedriger vertraulichen Senken entsteht. Im betrachteten, disjunktiven Verknüpfungskontext bedeutet dies, dass z. B. das Quellenlevel [User] an eine Senke mit Level [User;Airline] fließt (vgl. Unterabschnitt 2.1.3).

In Tabelle 8.1 werden sechs Akteure aufgelistet. Die ersten vier sind in der Modellierung bereits enthalten, die unteren beiden sind neu erstellt, um feingranularere Berechtigungen abzubilden. Dies wird im weiteren Verlauf des Abschnitts genauer ausgeführt. Für jeden Akteur ist ein Ort aufgelistet, auf dessen Klassen und Methoden er Zugriff hat. Die letzte Spalte beschreibt die Level, die dem Akteur zugewiesen sind. Er darf einen Parameter genau dann sehen, wenn mindestens eines seiner zugewiesenen Level in den Datasets des Parameters enthalten ist. Verletzungen der Vertraulichkeit werden von der Access Analysis in der Architektur genau dann gemeldet, wenn ein Akteur durch seinen Zugriffsort an eine Required oder Provided Schnittstelle des Systems gelangt, in dessen Einflussbereich ein Parameter liegt, den er nicht sehen darf [5]. Wenn z. B. ein Fluss vom Quellenlevel [User] an eine Senke des Ortes *Airline* mit dem Level [User;Airline] fließt, ist an dieser Senke der Akteur Airline berechtigt zuzugreifen, da er eines der enthaltenen Sicherheitslevel [Airline] besitzt. Durch den Fluss liegen nun zusätzlich Informationen mit dem Level [User] vor, zu dem der Akteur Airline jedoch keine Berechtigung besitzt und es kommt zu einer Vertraulichkeitsverletzung.

Akteur	abgekürzt	Zugriffsort	zugewiesene Level
User	Usr	*MobilePhone*	[A], [T], [U]
Airline	Air	*Airline*	[A]
TravelAgency	Trav	*TravelAgency*	[T]
SupportTechnician	Sup	*SupportCenter*	[S]
AirlineSupport (neu erstellt)	AirSup	*SupportCenter*	[A], [S]
UserSupport (neu erstellt)	UsrSup	*SupportCenter*	[S], [U]

Tabelle 8.1: Vollständige Liste der Akteure, ihrer Orte und ihrer Level, die ihnen zugewiesen sind. Neu erstellte Akteure sind entsprechend gekennzeichnet.

Für das Einfügen solcher Flüsse werden Äquivalenzklassen (ÄK) an Eigenschaften aufgestellt, in denen sich Flüsse unterscheiden und die Auswirkungen auf die Rückprojektion in die Architektur haben. Ein Repräsentant einer ÄK ist ein Fluss (oder eine Menge von Flüssen),

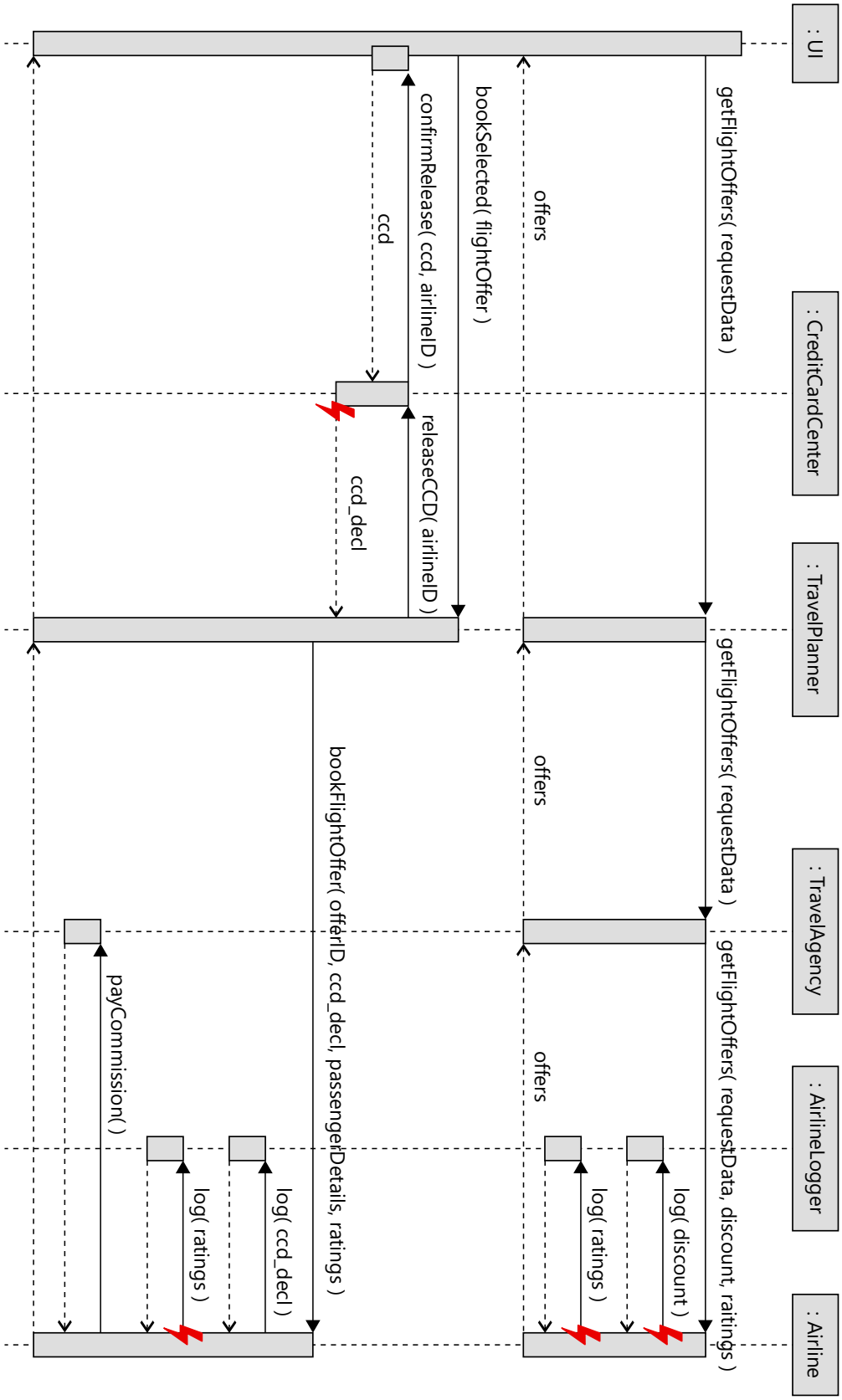


Abbildung 8.1: UML-Sequenzdiagramm der Komponenteninteraktionen des *TravelPlanners*

der den gleichen Einfluss auf die Analysenkopplung und das Ergebnis der Architekturanalyse haben soll, wie alle anderen Flüsse aus dieser ÄK. Es genügt somit, nur einen dieser Repräsentanten zu Evaluationszwecken einzufügen, um alle möglichen Effekte abzudecken, die für die Evaluation der Genauigkeit wichtig sind. Um einen solchen Repräsentanten in die Fallstudie einzubauen, werden Wertezuweisungen im Quelltext hinzugefügt. In manchen Fällen ist es zudem nötig, zusätzliche Elemente wie neue Akteure oder Parameter in das Architekturmodell einzufügen, um die Klassen realisieren zu können.

Anhand der Problemanalyse aus Unterabschnitt 6.3.2 sind folgende drei Dimensionen für mögliche ÄKs aufzustellen. Eine ÄK besitzt für jede dieser drei Dimensionen eine Zuweisung zu einem der dort genannten, alternativen Werte.

- **(D1) Legal oder Illegal:** Ein Informationsfluss kann legal oder illegal sein, je nachdem ob seine Flussrichtung konsistent zum Vertraulichkeitsbegriff ist oder nicht.
- **(D2) Einzeln oder Mehrfach:** Ein Informationsfluss zu einer Senke kann aus einer einzelnen Quelle oder aus mehreren Quellen stammen. Im zweiten Fall wird der Fluss zu einer Flussmenge.
- **(D3) Rückprojektion als Disjunktion darstellbar oder nicht:** Die Verknüpfung von Sicherheitsleveln für die Rückprojektion kann, wie in Absatz 5.2.2 beschrieben, manchmal als mehrstellige Disjunktion dargestellt werden, oder aber nur durch weitere Junktoren, die in der disjunktiven Darstellung nicht erlaubt sind.

Jeder eingefügte Fluss lässt sich über die Werte dieser drei Dimensionen eindeutig einer ÄK zuordnen. Er unterscheidet sich in seiner Zusammensetzung der Dimensionswerte von Flüssen, die aus anderen ÄKs stammen. Für jede ÄK wird im Folgenden beschrieben, welcher Repräsentant in der Fallstudie eingefügt wird. Eine Übersicht über alle eingebauten Flüsse bietet Abbildung 8.2.

(ÄK 1) Legal, Einzeln, Darstellbar: Ein solcher Fluss ist konsistent zum Vertraulichkeitsbegriff, besitzt nur eine Quelle und eine Senke und ist disjunktiv darstellbar, da sich das annotierte Sicherheitslevel der Flusssenke nicht verändert und bereits im Modell repräsentiert ist. Flüsse aus dieser Äquivalenzklasse haben keine Auswirkungen auf die Architektur und führen zu keinen Verletzungen.

Durch die Implementierung der grundlegenden Interaktionen, wie sie in Abbildung 8.1 dargestellt sind, existieren bereits legale Informationsflüsse. Bei der Propagierung des Parameters *requestData* der Methode *getflightOffers()* der Klasse *FlightQuery* an den Parameter *requestData* der Methode *getflightOffers()* der Klasse *FlightQueryWithDiscount* beim Abfragen von Angeboten haben alle Parameter das Level [Airline;TravelAgency;User]. Der Fluss ist somit legal (D1), einzeln (D2) und da es keine Aktualisierung gibt, auch darstellbar (D3). Diese ÄK muss deshalb nicht selbst injiziert werden.

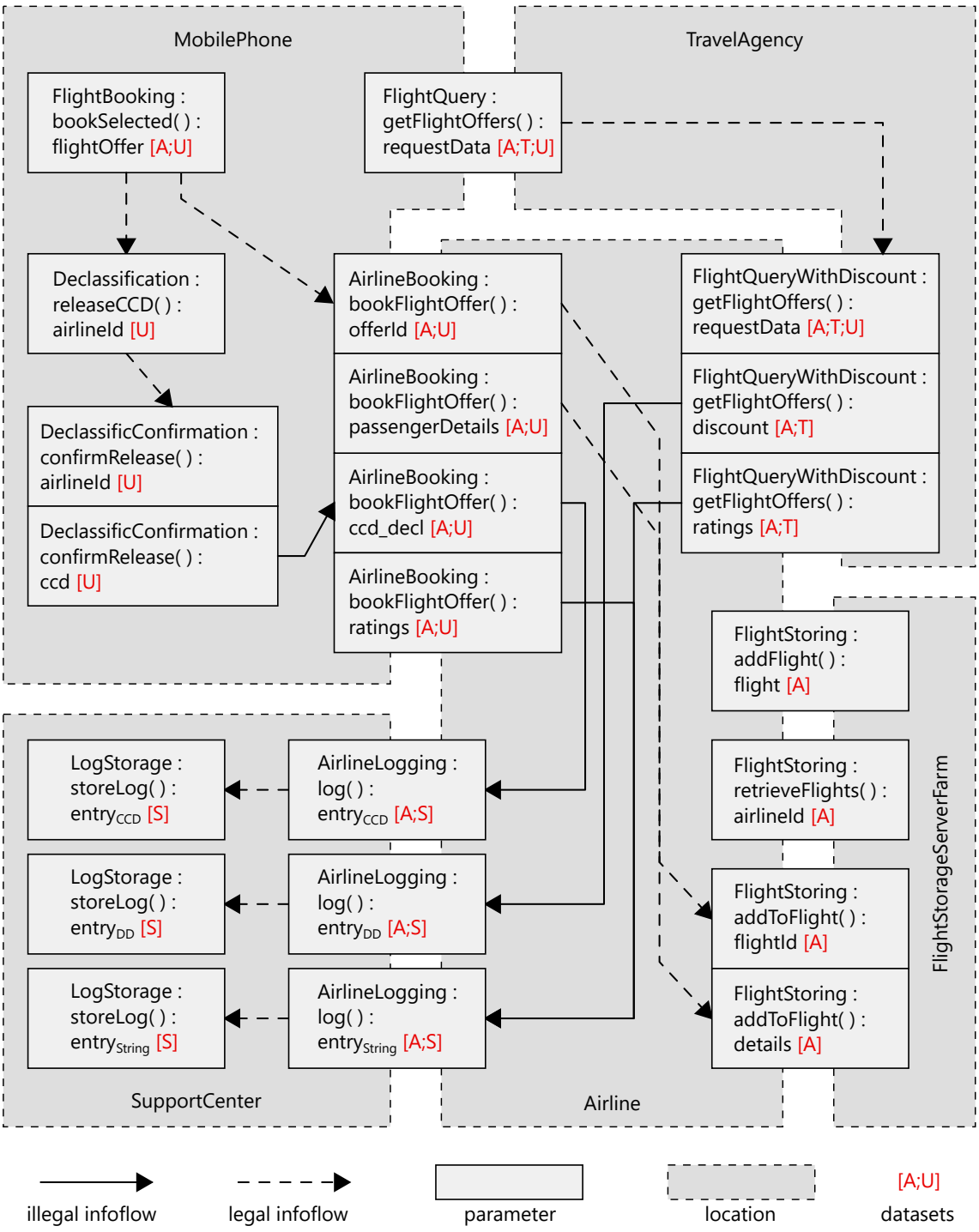


Abbildung 8.2: Informationsflüsse zwischen Parametern, die durch die Implementierung entstehen. Illegale Flüsse wurden explizit eingebaut, legale Datenflüsse entstehen durch die geforderte Funktionalität des Programms. Parameter die zwischen zwei *Locations* liegen, gehören zum Einflussbereich beider *Locations*.

(ÄK 2) Legal, Mehrfach, Darstellbar: Ein solcher Fluss ist konsistent zum Vertraulichkeitsbegriff, besitzt mehrere Quellen und eine Senke, und ist disjunktiv darstellbar, da die annotierten Sicherheitslevel der Quellen nicht auf die Flusssenke rückprojiziert und verknüpft werden müssen. Derartige Flüsse führen ebenfalls zu keinen Verletzungen.

In der Fallstudie liegt (ÄK 2) durch zwei Flüsse zwischen den verschiedenen Auftrittsorten des Parameters *requestData* bereits vor. Die erste Quelle befindet sich in der Methode *getflightOffers()* der Klasse *FlightQuery*, die von der Komponente TravelPlanner angeboten wird (vgl. Abbildung 8.1). Die zweite Quelle befindet sich in der gleichen Methodensignatur *getflightOffers()* der Klasse *FlightQuery*, die jedoch von der Komponente TravelAgency angeboten wird (vgl. Abbildung 8.1). Die Senke ist in beiden Fällen die Methode *getflightOffers()* der Klasse *FlightQueryWithDiscount*. An allen drei Orten hat der Parameter *requestData* das Level [Airline;TravelAgency;User]. Der Fluss ist somit legal (D1), mehrfach (D2) und da es keine Aktualisierung gibt, auch darstellbar (D3). Diese ÄK muss deshalb nicht selbst injiziert werden.

(ÄK 3) Illegal, Einzeln, Darstellbar: Ein solcher Fluss ist inkonsistent zum Vertraulichkeitsbegriff und besitzt nur eine Quelle und eine Senke. Die Inkonsistenz wird bei der Quelltextanalyse erkannt und das Senkenlevel muss in der Architektur aktualisiert werden. Sowohl mit dem Verknüpfungsoperator des Supremums als auch der Quellendisjunktion, vorgestellt in Absatz 5.2.2, wird das Senkenlevel durch das Quellenlevel ersetzt und ist somit disjunktiv darstellbar, da das Quellenlevel im Modell disjunktiv dargestellt vorliegt. Ein solcher Fluss führt zu einer Verletzung in der Architektur, sobald es am Ort der Senke einen Akteur gibt, der das zurückprojizierte Level nicht sehen darf.

Als Repräsentant dieser ÄK wird der illegale Fluss des Parameters *ccd* der Methode *confirmRelease()* aus der Klasse *DeclassificationConfirmation* zum Parameter *ccd_decl* der Methode *bookFlightOffer()* aus der Klasse *AirlineBooking* eingeführt. Bereits in der Evaluation von Katkalov et al. [8], Seifermann et al. [3] und Häring [9] wurde ein solcher Fluss benutzt, um eine fehlende Deklassifizierung der Kreditkarteninformationen zu repräsentieren. Der Fluss ist illegal, da er vom Sicherheitslevel [User] zum Sicherheitslevel [Airline;User] verläuft (D1). Außerdem ist er einzeln (D2) und disjunktiv darstellbar als [User](D3). Er führt am Ort der Senke durch den Akteur Airline zu einer Verletzung, da dieser Akteur keinen erlaubten Zugriff auf Level [User] hat, wie Tabelle 8.1 zeigt.

(ÄK 4) Illegal, Mehrfach, Darstellbar: Ein solcher Fluss ist inkonsistent zum Vertraulichkeitsbegriff und besitzt mehrere Quellen und eine Senke. Die Quellenlevel sind verschieden, aber beide höher vertraulich als das Senkenlevel. Eine der Sicherheitslevelmengen ist eine Teilmenge der anderen. Eine Darstellung ihrer Verknüpfung ist als Disjunktion möglich, indem nur die kleinere der beiden Levelmenge für die Rückprojektion genutzt wird, wie in Unterabschnitt 6.3.2 durch Äquivalenzumformungen gezeigt wurde. Eine solche, korrekte Rückprojektion findet jedoch nur der Supremumsoperator des iterativen Ansatzes, der Operator der Quellendisjunktion des nicht-iterativen Ansatzes hingegen nicht. Ein Fluss aus (ÄK 4) führt in der Architektur somit je nach Ansatz zu einer unterschiedlichen Anzahl an Verletzungen, da der Quellendisjunktionsoperator eine Unterabschätzung darstellt. Damit

dies im Analyseergebnis sichtbar wird, muss es in der Architektur Akteure geben, die im nicht-iterativen Ansatz fälschlicherweise keine Verletzung erzeugen, da sie durch die Unterabschätzung des rückprojizierten Levels weiterhin Zugriff auf das Element besitzen.

Es ist deshalb für Flüsse dieser ÄK wichtig, dass ein Akteur an dem Ort im Architekturmodell vorliegt, an dem die Klassen des Senkenparameter allokiert sind. Die zugewiesenen Level, die der Akteur sehen darf, müssen so verteilt sein, dass er zwar die Disjunktion der Levelmengen der Quelle sehen darf, aber nicht auf die kleinere der beiden Levelmengen der Quellen zugreifen darf. So wird für diese kleinere Quelle, die eigentlich eine Verletzung erzeugen sollte, keine solche Verletzung gefunden und somit die Unterabschätzung für die Evaluation sichtbar gemacht.

Es werden als Repräsentanten für diese ÄK folgende zwei Flüsse neu eingeführt: Als Erstes wird ein illegaler Fluss vom Parameter *ccd_decl* der Methode *bookFlightOffer()* aus der Klasse *AirlineBooking* zum Parameter $entry_{CCD}$ der Methode *log()* aus der Klasse *AirlineLogging* erstellt. Als Zweites führt der in (ÄK 3) injizierte Fluss des Parameters *ccd* der Methode *confirmRelease()* aus der Klasse *DeclassificationConfirmation* ebenfalls zu einem Fluss zum Parameter $entry_{CCD}$. Denn es besteht eine transitive Weitergabe der Parameter durch Zuweisungsoperationen im Quelltext, wodurch auch ein Informationsfluss entsteht. Der Fluss ist somit mehrfach (D2). Wie in Abbildung 8.2 annotiert, fließen dadurch einerseits das Level [User] nach [Airline;Support] und andererseits das Level [Airline;User] nach [Airline;Support], was beides illegal ist (D1). Der Supremumsoperator ergibt für die Rückprojektion des Senkenlevels das aktualisierte Level [User] $\wedge$ [Airline;User] = [User], was dadurch disjunktiv darstellbar ist (D3), während der Operator der Quellendisjunktion das aktualisierte Level [Airline;User] erzeugt. Um eine Unterabschätzung im nicht-iterativen Ansatz sichtbar zu machen, wird ein Akteur benötigt, der nach dem im vorherigen Absatz beschriebenen Vorgehen das Sicherheitslevel [Airline;User] sehen darf, aber das Level [User] nicht. Da der Parameter $entry_{CCD}$ in den Einflussbereich des Ortes *Airline* fällt und dort bereits der Akteur Airline existiert, der mit dem zugewiesenen Sicherheitslevel [Airline] genau diese Bedingung erfüllt (vgl. Tabelle 8.1), muss kein weiterer Akteur an diesen Ort hinzugefügt werden.

Da der Parameter $entry_{CCD}$ jedoch durch die bereits vorhandene Implementierung auch an den gleichnamigen Parameter $entry_{CCD}$ der Methode *storeLog()* aus der Klasse *LogStorage* fließt und dies nicht im Einflussbereich des Akteurs Airline liegt, wird dem Ort *SupportCenter* der Akteur AirlineSupport neu hinzugefügt. So kann auch dort die Unterabschätzung gemessen werden, da ein Akteur, der eigentlich eine Verletzung erzeugen sollte, keine erzeugt. Damit dieser Akteur die Konsistenz des ursprünglichen Architekturmodells nicht verletzt, muss er zusätzlich zum Level [Airline] auch noch das Level [Support] erhalten. Denn es liegen in seinem Einflussbereich Parameter vor, die nur mit dem Level [Support] einsehbar sind. Somit ergibt sich für seine modellierten Berechtigungen [Airline;Support]. Eine Übersicht der vorhandenen und neu erstellten Akteure bietet Tabelle 8.1. Die dort aufgelisteten Abkürzungen werden im Folgenden zur vereinfachten Kennzeichnung in Grafiken und Tabellen genutzt.

(ÄK 5) Illegal, Mehrfach, Nicht Darstellbar: Ein solcher Fluss ist inkonsistent zum Vertraulichkeitsbegriff und besitzt mehrere Quellen und eine Senke. Die Quellenlevel sind verschieden, aber beide höher vertraulich als das Senkenlevel. Bei einem Fluss dieser ÄK ist es nicht möglich, die Sicherheitslevel der Quellen durch eine Disjunktion darzustellen. Der Operator der Quellendisjunktion führt wie bei Flüssen der (ÄK 4) ebenfalls eine Unterabschätzung des rückprojizierten verknüpften Levels durch. Der iterative Ansatz mit dem Supremumsoperator testet die Quellenlevel hingegen einzeln, wie in Abschnitt 5.4 beschrieben und kann deshalb eine feingranulare Überprüfung der Verletzungen gewährleisten. Ein Fluss der (ÄK 5) führt in der Architektur je nach Ansatz zu einer unterschiedlichen Anzahl an Verletzungen.

Wie für Flüsse der (ÄK 4) ist es auch hier wichtig, dass es einen Akteur gibt, der den Unterschied zwischen den korrekten und den unterabgeschätzten Verletzungen in der Access Analysis sichtbar macht. Wenn keiner existiert, soll er für die Evaluation hinzugefügt

Flusselement	Parameter	Parameterlevel
Quelle 1	DeclassificationConfirmation : confirmRelease() : ccd	U
Senke 1	AirlineBooking : bookFlightOffer() : ccd_decl	A;U
Quelle 2	DeclassificationConfirmation : confirmRelease() : ccd	U
Senke 2	AirlineLogging : log() : $\text{entry}_{\text{CCD}}$	A;S
Quelle 3	DeclassificationConfirmation : confirmRelease() : ccd	U
Senke 3	LogStorage : storeLog() : $\text{entry}_{\text{CCD}}$	S
Quelle 4	AirlineBooking : bookFlightOffer() : ccd_decl	A;U
Senke 4	AirlineLogging : log() : $\text{entry}_{\text{CCD}}$	A;S
Quelle 5	AirlineBooking : bookFlightOffer() : ccd_decl	A;U
Senke 5	LogStorage : storeLog() : $\text{entry}_{\text{CCD}}$	S
Quelle 6	FlightQueryWithDiscount : getFlightOffers() : discount	A;T
Senke 6	AirlineLogging : log() : entry_{DD}	A;S
Quelle 7	FlightQueryWithDiscount : getFlightOffers() : discount	A;T
Senke 7	LogStorage : storeLog() : entry_{DD}	S
Quelle 8	AirlineBooking : bookFlightOffer() : ratings	A;U
Senke 8	AirlineLogging : log() : $\text{entry}_{\text{String}}$	A;S
Quelle 9	AirlineBooking : bookFlightOffer() : ratings	A;U
Senke 9	LogStorage : storeLog() : $\text{entry}_{\text{String}}$	S
Quelle 10	FlightQueryWithDiscount : getFlightOffers() : ratings	A;T
Senke 10	AirlineLogging : log() : $\text{entry}_{\text{String}}$	A;S
Quelle 11	FlightQueryWithDiscount : getFlightOffers() : ratings	A;T
Senke 11	LogStorage : storeLog() : $\text{entry}_{\text{String}}$	S

Tabelle 8.2: Vollständige Liste aller eingefügten illegalen Informationsflüsse inklusive der transitiven illegalen Flüsse

werden. Die zugewiesenen Level, die der neue Akteur sehen darf, müssen so gewählt werden, dass er zwar die Disjunktion der Levelmengen der Quelle sehen darf, aber auf eine der beiden Quellenlevel nicht zugreifen darf. So wird für diese Quelle, die eigentlich eine Verletzung erzeugen sollte, keine solche Verletzung gefunden und somit die Unterabschätzung für die Evaluation sichtbar gemacht.

Als Repräsentanten für diese ÄK werden zwei illegale Flüsse zum Parameter $entry_{String}$ der Methode *log()* aus der Klasse *AirlineLogging* eingeführt, der das Level [Airline;Support] besitzt. Der erste Fluss beginnt im Parameter *ratings* der Methode *bookFlightOffer()* aus der Klasse *AirlineBooking* mit dem Level [Airline;User]. Der zweite Fluss beginnt im Parameter *ratings* der Methode *getFlightOffers()* aus der Klasse *FlightQueryWithDiscount* mit dem Level [Airline;TravelAgency]. Der Fluss ist somit mehrfach (D2) und inkonsistent zum Vertraulichkeitsbegriff (D1). Die Verknüpfung von [Airline;User] und [Airline;TravelAgency] ist nicht disjunktiv darstellbar(D3). Um eine Unterabschätzung für den nicht-iterativen Ansatz sichtbar zu machen, wird der Akteur UserSupport im *SupportCenter* eingeführt, der die Level [User] und [Support] sehen darf. Letzteres muss Bestandteil sein, um die Konsistenz zur ursprünglichen Vertraulichkeit des Modells zu erhalten. Im Falle des Supremumsoperators erzeugt der Akteur bezüglich des rückprojizierten Teillevels [Airline;TravelAgency] eine Verletzung, während für das rückprojizierte Level des Quellendisjunktionsoperators [Airline;TravelAgency;User] keine Verletzung entsteht, da der Akteur das dort enthaltene Level [User] sehen darf.

8.2.2 Goldstandard

Im Folgenden wird der Goldstandard beschrieben, der genau die Verletzungen der Vertraulichkeit enthält, die durch die eingefügten Flüsse und Modellmodifikationen aus dem vorherigen Abschnitt im Architekturmodell durch eine Rückprojektion der Quelltextanalyse entstehen sollen. Wie dort bereits beschrieben wurde, darf ein Akteur einen Parameter sehen, wenn mindestens eines seiner zugewiesenen Level in den Datasets des Parameters enthalten ist. Verletzungen der Vertraulichkeit werden von der Access Analysis genau dann gemeldet, wenn ein Akteur durch seinen Zugriffsort an eine Required oder Provided Schnittstelle des Systems gelangt, in dessen Einflussbereich ein Parameter liegt, den er nicht sehen darf.

In Tabelle 8.2 wurden die illegalen Informationsflüsse aufgelistet. Für jede der dort enthaltenen Flusssenken ist das resultierende verknüpfte Sicherheitslevel zu berechnen, indem bei einem einzelnen Fluss eine Ersetzung durch das Quellenlevel durchgeführt und bei einem mehrfachen Fluss alle Quellenlevel konjungiert (nicht disjungiert) werden, wie es der Supremumsoperator vorschreibt (vgl. Absatz 5.2.2). Die Ergebnisse der resultierenden Sicherheitslevel finden sich in Tabelle 8.3. Jede Senke, die in Tabelle 8.2 mehrfach vorkommt und die somit zu einem mehrfachen Fluss gehört, wird in Tabelle 8.3 jeweils nur mit der Nummer ihres ersten Auftretens in der Tabelle referenziert. Für die Required und Provided Schnittstellen, in denen die Senkenparameter liegen, wird die Zuordnung zu einem physikalischen Ort grafisch in Abbildung 8.2 dargestellt. Zu jeder Location, werden im nächsten Schritt die dort zugreifenden Akteure aus Tabelle 8.1 abgelesen und nach ihrem

Zugriffsort entweder unter A_{prov} oder A_{req} aufgelistet. Im Falle von z. B. Senke 3 in der Tabelle wird deutlich, dass manche Parameter nur innerhalb eines einzigen physikalischen Ortes weitergereicht werden und deshalb sowohl die Required als auch die Provided Schnittstelle an diesem Ort liegt. Aus diesem Grund erzeugt ein Akteur an diesem Ort an beiden dieser Schnittstellen eine Verletzung in der Architekturanalyse. Als Nächstes wird überprüft, ob ein gelisteter Akteur auf das resultierende Level zugreifen darf. Wenn dies nicht der Fall ist, liegt eine Verletzung vor. In Tabelle 8.3 sind alle Akteure unterstrichen, die eine Verletzung erzeugen. Für Senke 1 bedeutet z. B. Akteur Airline eine Verletzung, da er nur die Berechtigung für Level [A] besitzt und das resultierende Level [U] somit nicht sehen darf. Zuletzt wird für jede Senke die Anzahl der Akteure, die eine Verletzung verursachen, zu den erwarteten Verletzungen aufsummiert. Dies ist in der letzten Spalte von Tabelle 8.3 dargestellt. Die Gesamtsumme aller Verletzungen des Goldstandards beträgt 20.

Senke	Level	$Level_{result}$	A_{prov}	A_{req}	Verletzungen
1	A;U	U	Air	Usr	1
2	A;S	U	Sup, AirSup, UsrSup	Air	3
3	S	U	Sup, AirSup, UsrSup	Sup, AirSup, UsrSup	4
6	A;S	A;T	Sup, AirSup, UsrSup	Air	2
7	S	A;T	Sup, AirSup, UsrSup	Sup, AirSup, UsrSup	4
8	A;S	A;U ∧ A;T	Sup, AirSup, UsrSup	Air	2
9	S	A;U ∧ A;T	Sup, AirSup, UsrSup	Sup, AirSup, UsrSup	4

Tabelle 8.3: Goldstandard für die erwarteten Verletzungen. Die Nummern der Senken beziehen sich auf Tabelle 8.2. Abgekürzt mit A_{prov} sind die Akteure, die den Parameter über die Provided Schnittstelle sehen, A_{req} sind die Akteure, die den Parameter über die Required Schnittstelle sehen. Die Abkürzungen der Akteure wurden aus Tabelle 8.1 übernommen. Unterstrichene Akteure bedeuten eine Verletzung, da sie das resultierende Level nicht sehen dürfen. Die Summe aller unterstrichenen Akteure und somit Verletzungen im Goldstandard beträgt 20.

8.3 Erhebung der Metriken

In diesem Abschnitt wird für jedes der drei Hauptziele des GQM-Plans beschrieben, wie die Evaluation durchgeführt und die Metriken zur Beantwortung der Teilfragen gemessen werden.

8.3.1 Vorgehen zur Messung der Genauigkeitsmetriken

Für das erste GQM-Ziel (G1) werden die Metriken Sensitivität und Präzision gemessen. Im Folgenden wird das genaue Vorgehen beschrieben, wie die Werte aus der Analyse abgeleitet werden.

Es werden vier Konfigurationen des Quelltextanalyseansatzes anhand der Fallstudie aus Abschnitt 8.2 miteinander verglichen. Die erste Konfiguration nutzt als Quelltextanalyse JOANA und wird nicht-iterativ ausgeführt. Der nicht-iterative Ansatz nutzt den Operator der Quellendisjunktion zum Verknüpfen von mehreren Sicherheitsleveln in der Rückprojektion. Die zweite Konfiguration nutzt als Quelltextanalyse JOANA und wird iterativ ausgeführt. Als Iterationskonstrukt wird über gleiche Quellenlevel iteriert. Als Abbruchbedingung wird die vollständige Abdeckung des Iterationskonstrukts gewählt. Dies ist durch einmaliges Ausführen aller Iterationen gegeben. Die dritte Konfiguration nutzt als Quelltextanalyse CodeQL und wird nicht-iterativ ausgeführt. Ansonsten gleicht sie der ersten Konfiguration. Die vierte Konfiguration nutzt als Quelltextanalyse CodeQL und wird iterativ ausgeführt. Ansonsten gleicht sie der zweiten Konfiguration.

Alle Konfigurationen werden über das Architecture-And-StaticCode-Analyses-CouplingFramework ausgeführt, indem eine entsprechende Konfigurationsdatei im XML-Format erstellt wird, die die Pfade zu den Modellen der Fallstudie enthält.

Für jede der vier Konfigurationen ist das Ergebnis der Quelltextanalyse eine Rückprojektion neuer *DataSets* in die *ParameterAndDataPairs* der Confidentiality-Datei des Modells. Im Falle der iterativen Konfigurationen wird pro Iteration eine neue Ergebnisdatei mit indiziertem Dateinamen erstellt.

Im Anschluss folgt für alle vier Konfigurationen die Kopplung mit der Architekturanalyse. Hierfür werden alle Modelle einzeln mit der aktuellsten Prolog-Implementierung *haskalladio* [68] der Access Analysis aus Kramers Ansatz [5] analysiert. Als Ergebnis werden Verletzungen ausgegeben und durch einen Interpreter zu Textzeilen mit folgenden Informationen formatiert:

- Komponente, z. B. Airline
- Schnittstellensignatur, z. B. required Signature AirlineLogging::log(entry)
- Parameter, z. B. entry
- Akteur, z. B. Airline
- Verursachendes DataSet, z. B. [User;Support]

Für einen iterativen Ansatz werden alle Ergebniszeilen zusammen betrachtet, die aus einzeln ausgeführten Durchläufen der Access Analysis für jede Iteration entstehen. Wenn ein Akteur mehrere unterschiedliche Verletzungen erzeugt, werden diese einzeln mit dem Goldstandard aus Unterabschnitt 8.2.2 verglichen. Wenn zwei Ergebniszeilen der Access Analysis jedoch bis auf das verursachende DataSet identisch sind, werden die Zeilen nur einfach gewertet, da der dort spezifizierte Akteur eine Verletzung an derselben Schnittstelle erzeugt und dies nicht doppelt gezählt werden darf. Denn es geht darum, ob ein Akteur einen Parameter sehen darf oder nicht. Wenn zwei Verletzungen für den gleichen Parameter registriert werden, bieten diese keinen Mehrwert und können zusammengefasst werden.

Für jede Konfiguration liegt durch die beschriebene Prozedur eine Ergebnisliste an Zeilen vor, die jeweils eine Verletzung beschreiben, die sich über die enthaltenen Informationen

mit der Verletzungstabelle des Goldstandards aus Unterabschnitt 8.2.2 abgleichen lässt. Eine korrespondierende Verletzung liegt vor, wenn die Schnittstellensignatur Provided ist und im Goldstandard ein unterstrichener A_{prov} vorliegt, dessen Senke die gleiche Signatur hat. Analog liegt eine korrespondierende Verletzung vor, wenn die Schnittstellensignatur Required ist und im Goldstandard ein unterstrichener A_{req} vorliegt, dessen Senke die gleiche Signatur hat.

Jede Ergebniszeile, für die der Goldstandard keine korrespondierende Verletzung vorschreibt, wird als falsch Positiv *FP* markiert. Jede Ergebniszeile, für die auch der Goldstandard eine Verletzung beinhaltet, wird als korrekt Positiv *TP* markiert. Sowohl für die falsch Positiven, als auch die korrekt Positiven wird die Anzahl der markierten Zeilen abgelesen und als Wert für die Berechnung der Metriken von Präzision und Sensitivität genutzt. Die Anzahl an falsch Negativen *FN*, die für die Sensitivität notwendig ist, ergibt sich aus der Anzahl aller Verletzungen des Goldstandards, für die keine korrespondierende Ergebniszeile vorliegt. Zuletzt werden Sensitivität r und Präzision p über die Formeln $r = \frac{TP}{TP+FN}$ und $p = \frac{TP}{TP+FP}$ berechnet.

8.3.2 Vorgehen zur Messung der Performanzmetriken

Während der Ausführung der Analysen am Modell der Fallstudie werden technische Messungen des Analyseprozesses durchgeführt. In diesem Abschnitt wird beschrieben, wie die drei Metriken Laufzeit, Arbeitsspeicherverbrauch und Festplattenspeicherverbrauch für das zweite GQM-Ziel (G2) gemessen werden.

Die Laufzeit wird über einen Millisekundenzähler gemessen, der in der Runner-Klasse des Analyseframeworks eingebaut ist. Dieser berechnet die Laufzeit aus der Differenz der Prozessorzeiten vom Startzeitpunkt des Plugins bis zu seinem Endzeitpunkt und liefert das Ergebnis als Konsolenausgabe in der Einheit Millisekunden zurück.

Um den Arbeitsspeicherverbrauch eines Analysedurchlaufs zu messen, wurden zwei Alternativen in Betracht gezogen: das Messen über einen Java-Profiler oder das Messen über den Performance-Monitor von Windows. Java-Prozesse können über Profiler-Anwendungen wie z. B. JProfiler [69] oder JConsole [70] überwacht werden. Derartige Profiler-Anwendungen sind jedoch ungeeignet für die Messung des iterativen Analyseansatzes, da dort während jedes Analyseschritts und der Delegation an den zuständigen Adapter ein neuer Java-Prozess gestartet wird. Ein automatisiertes Verbinden zu solchen Kindprozessen ist in den genannten Anwendungen nicht möglich. Außerdem wird im Falle der Nutzung von CodeQL als Quelltextanalyse ein Prozess gestartet, der kein Java-Prozess ist und so nicht gemessen werden könnte.

Die zweite Alternative zur Messung des Arbeitsspeicherverbrauchs ist das Anlegen eines benutzerdefinierten Datensammlersatzes in der Leistungsüberwachung *perfmon.exe* von Windows [67]. Diese Alternative wird in der Evaluation genutzt, da im Gegensatz zu Java-Profilern der gesamte Verbrauch aller Analyseprozesse zusammen gemessen werden kann, unabhängig davon, ob es sich bei ihnen um Java-Prozesse handelt oder nicht. Hierfür wird ein DataCollector angelegt, der die beiden Leistungsindikatoren *Arbeitsspeicher*

- *Verfügbare MB* und *Arbeitsspeicher - Zugesicherte verwendete Bytes (%)* aufzeichnet. Als Abtastintervall wird 1 Sekunde eingestellt. Da hiermit der gesamte Arbeitsspeicherverbrauch des Computers gemessen wird, muss die Differenz zwischen dem Ausgangswert vor dem Analysestart und den Messwerten während der Analysedurchführung berechnet werden, um den prozessspezifischen Arbeitsspeicherverbrauch separat zu erhalten. Es werden keine anderen Nutzeranwendungen zur Analysezeit ausgeführt, um Verfälschungen durch deren Arbeitsspeicherverbrauch zu vermeiden. Um Hintergrundschwankungen der Arbeitsspeicherauslastung durch Windows-Systemprozesse zur Analysezeit auszugleichen, werden die Messungen mehrerer, gleich konfigurierter Analysedurchführungen zu einem Durchschnittswert zusammengefügt. Die Messwerte werden vom Performance-Monitor nach Beenden der Aufzeichnung in einem Datensatz als CSV-Datei gespeichert.

Als dritte Metrik wird der Speicherplatzverbrauch des Architekturmodells nach der Quelltextanalyse gemessen. Hierfür wird die Summe des Speicherplatzverbrauchs aller enthaltenen Dateien berechnet, die für die Kopplung an die Architekturanalyse weitergegeben werden müssen. Der Verbrauch wird jeweils aus dem Windows-Dateisystem abgelesen.

Für eine Reproduzierbarkeit der Evaluationsergebnisse oder zur Einordnung der Größen von erzeugten Messwerten in Abhängigkeit von der Computerleistung, werden die technischen Daten des für die Analysenausführung genutzten Computers in Tabelle 8.4 aufgelistet.

Prozessor	AMD Ryzen 9 5900X 12-Core Processor, 3.70 GHz
Grafikkarte	NVIDEA GeForce RTX 3080
Arbeitsspeicher	32.0 GB installierter DDR4-3600-RAM
Betriebssystem	Windows 10 Pro, 64-Bit

Tabelle 8.4: Systemspezifikationen des für die Evaluation genutzten Referenzsystems

8.3.3 Vorgehen zur Messung der Skalierbarkeitsmetriken der Verbandsoptimierung

Um das GQM-Ziel (G3) zu bewerten, wird als Metrik ebenfalls die Laufzeit herangezogen. Verglichen mit dem Vorgehen aus Unterabschnitt 8.3.2 wird hierbei jedoch nicht die gesamte Quelltextanalyse gemessen, sondern nur der Schritt der Verbandsgenerierung. Um die Skalierbarkeit zu testen, wird nacheinander eine immer größer werdende Eingabe, bestehend aus ausgeprägten Leveln, automatisch generiert. Hierfür werden im ersten Durchlauf 1 Einzellevel, im zweiten Durchlauf 2 Einzellevel, bis hin zum n-ten Durchlauf n Einzellevel zum Initialisieren der Grundmenge an ausgeprägten Leveln genutzt. Zusätzlich werden aus den Einzelleveln weitere kombinierte Level ausgeprägt, die eine zufällige Zusammensetzung der Einzellevel ihres Durchlaufs besitzen. Im ersten Durchlauf wird so nur ein ausgeprägtes Level an die beiden Generatoren übergeben. Für jeden weiteren Messlauf kommt ein neues Einzellevel zur Unterteilung hinzu. Es teilt die ausgeprägten Level des Vorgängermesslaufs in zwei Gruppen, in diejenigen Level, die das hinzukommende auch zugewiesen bekommen und in diejenigen Level, die das neue Level nicht zugewiesen bekommen. Aus diesem Grund

erhöht sich die Anzahl ausgeprägter Level um die Hälfte der ausgeprägten Level des Vorgängermesslaufs (entspricht dem Faktor 1, 5). Die Anzahl wird auf die nächst höhere ganze Zahl aufgerundet. Für die ersten 6 Durchläufe ergibt sich somit folgende Reihe für die Anzahl an insgesamt ausgeprägten Leveln: 1, 2, 3, 5, 8, 12. Die Durchläufe werden sowohl für den naiven Potenzmengen-Ansatz als auch für die Optimierung über die Beschränkte Transitive Reduktion automatisiert hintereinander gestartet, bis die Gesamtlaufzeit aller einzelnen Durchläufe eine Stunde überschreitet. In diesem Fall wird die Messreihe beendet.

8.4 Ergebnispräsentation und Diskussion der Fallstudien

Im Folgenden werden die Ergebnisse der Evaluation nach den drei GQM-Zielen getrennt beschrieben und diskutiert.

8.4.1 Evaluationsergebnisse und Diskussion der Genauigkeit

In Abbildung 8.3 werden die Ergebnisse der Rückprojektion dargestellt. Die erste Durchführung wurde mit der Konfiguration *JOANA-non-it* vollzogen, wie sie in Abschnitt 7.3 beschrieben wird. Es lässt sich an den rückprojizierten Leveln des Durchlaufs im Vergleich zu den Leveln des Goldstandards ablesen, dass JOANA viele Informationsflüsse findet, die nicht erwartet sind. Vor allem finden auch Rückprojektionen auf Parameter statt, die im Programmablauf nur als Eingabeparameter im System genutzt werden und deren zugehörige Methode den Rückgabetyp *void* hat, z. B. wie für den Parameter *requestData* der Methode *getFlightOffers()*. Auch dass das Level *Support* ein Teil einiger Rückprojektionen ist, widerspricht der Implementierung, da alle Aufrufe an Komponenten, die im *SupportCenter* allokiert sind, den Rückgabetyp *void* haben und somit keine ausgehenden Informationsflüsse erzeugen sollten. Es ist zu berücksichtigen, dass JOANA als Werkzeug zur Untersuchung der Informationsflusskontrolle eine Überabschätzung der gefundenen Informationsflüsse in Kauf nimmt, um im Sinne einer *Konservativen Approximation* die Korrektheit zu gewährleisten [65]. Auf der anderen Seite wird aber deutlich, dass der Operator der Quellendisjunktion in den erwarteten illegalen Flüssen, die durch den Goldstandard beschrieben werden, die rückprojizierten Levelmengen zu groß fasst und gegenüber Akteuren so mehr potentielle Zugriffsberechtigungen bietet, wie z. B. bei der Rückprojektion des Levels [A;T;U] für den Parameter $storeLog() : entry_{String}$. Dadurch wird eine Unterabschätzung erzeugt. In Tabelle 8.5 werden die Ergebnisse der anschließenden Architekturanalyse dargestellt. Die Sensitivität, die als Metrik für die Korrektheit aus GQM-Frage (Q1.1) steht, zeigt, dass der nicht-iterative Ansatz weniger als ein Drittel aller erwarteten Verletzungen findet. Die Präzision für GQM-Frage (Q1.2) ergibt zudem, dass ein Drittel der gefundenen Verletzungen einen *Falschen Alarm* darstellen.

Die zweite Durchführung wurde mit der Konfiguration *JOANA-it* vollzogen. In dieser iterativen Ausführung werden insgesamt sieben Iterationen durchgeführt und in jeder Iteration ein bestimmtes Level als Quelllevel festgelegt. Nur dieses Quelllevel tritt in den Rückprojektionen der jeweiligen Iteration auf. Wie in Abbildung 8.3 dargestellt, zeigt

Parameter	Level	$\text{Level}_{\text{Gold}}$	JOANA-non-it	JOANA-it	CodeQL-non-it	CodeQL-it
log() : $\text{entry}_{\text{CCD}}$	A;S	U	U;S	S - U - A;U - A;T	U	U - A;U
log() : entry_{DD}	A;S	A;T				A;T
log() : $\text{entry}_{\text{String}}$	A;S	A;U ∧ A;T	U;S	S - U - A;U - A;T		A;U - A;T
storeLog() : $\text{entry}_{\text{CCD}}$	S	U	A;T;U	U - A;U - A;T	A;U	U - A;U
storeLog() : entry_{DD}	S	A;T	A;T	A;T	A;T	A;T
storeLog() : $\text{entry}_{\text{String}}$	S	A;U ∧ A;T	A;T;U	U - A;U - A;T	A;T;U	A;U - A;T
getFlightOffers() : requestData	A;T;U		A;T	A;T		
releaseCCD() : airlineId	U		A;T	A;T		
confirmRelease() : ccd	U		A;T	A;T		
confirmRelease() : airlineId	U		A;T	A;T		
bookFlightOffer() : offerId	A;U			A;T		
bookFlightOffer() : ccd_decl	A;U	U	U;S	A;S - S - U - A;T	U	U
bookFlightOffer() : passengerDetails	A;U		U;S	A;S - S - U - A;T		
bookFlightOffer() : ratings	A;U		U;S	A;S - S - U - A;T		
FQWD : getFlightOffers() : requestData	A;T;U		A;T	A;T		
FQWD : getFlightOffers() : discount	A;T					
FQWD : getFlightOffers() : ratings	A;T					
bookSelected() : flightOffer	A;U			A;T		
addToFlight() : flightId	A					
addToFlight() : details	A					
addFlight() : flight	A					
retrieveFlights() : airlineId	A					

Abbildung 8.3: Ergebnisse der Rückprojektionen im Vergleich zu den Ursprungsleveln und dem Goldstandard. Die Parameter werden nur über ihre Methoden referenziert. Einzig die Parameter der Schnittstelle FlightQueryWithDiscount : getFlightOffers() werden zusätzlich mit FQWD annotiert, um eine eindeutige Referenzierung zu ermöglichen. Rückprojizierte Ergebnisse der iterativen Konfigurationen werden durch Bindestriche separiert.

Konfiguration	TP	FP	FN	Sensitivität	Präzision
JOANA-non-it	6	3	14	0,3	0,66667
JOANA-it	19	13	1	0,95	0,59375
CodeQL-non-it	12	0	8	0,6	1,0
CodeQL-it	20	0	0	1,0	1,0

Tabelle 8.5: Gemessene Werte für die Metriken Sensitivität und Präzision, sowie die Anzahl an korrekt Positiven *TP*, falsch Positiven *FP* und falsch Negativen *FN* nach Ausführung der Analysenkopplung.

sich, dass nur in fünf der sieben Iterationen überhaupt eine Rückprojektion stattgefunden hat. Verglichen mit dem Goldstandard wird außerdem deutlich, dass (mit Ausnahme der zweiten Zeile) jedes erwartete Level des Goldstandards in einer der Iterationen für den entsprechenden Parameter rückprojiziert wird. Es wird somit eine sehr hohe Abdeckung der erwarteten Verletzungen erreicht, was sich in Tabelle 8.5 in der Metrik der Sensitivität für diese Konfiguration mit einem Wert von 0,95 wiederspiegelt. Insbesondere zeigt diese Metrik, dass die iterative Ausführung der Analyse die Korrektheit gegenüber dem nicht-iterativen Ausführung aus der ersten Konfiguration deutlich verbessert. Ein Nachteil der Iterationen ist, dass es sehr viele feingranularere Rückprojektionen gibt, die mehr Raum für *Falsche Alarme* bieten. Die Präzision verringert sich im Vergleich zur nicht-iterativen Ausführung aber nur leicht, wie Tabelle 8.5 belegt.

Als Drittes wurde die Konfiguration *CodeQL-non-it* ausgeführt. Wichtig hierbei ist, dass CodeQL im Vergleich zu JOANA nur Datenflüsse und keine Informationsflüsse untersucht. Da die im Goldstandard betrachteten Informationsflüsse jedoch durch die Implementierung abgeleitet wurden, ist zu bestätigen, dass alle Flüsse ebenfalls als echte Datenflüsse zwischen den Parametern umgesetzt wurden. Im Vergleich zu den Analysen mit JOANA fällt auf, dass Rückprojektionen nur an Parametern auftreten, deren Level auch durch den Goldstandard aktualisiert wird (vgl. Abbildung 8.3). Da das rückprojizierte Level in dieser Durchführung außerdem durch den Quellendisjunktionsoperator unterabgeschätzt wird, werden keine falsch Positiven Verletzungen erzeugt, was zu einer Präzision von 1,0 führt (vgl. Tabelle 8.5). Die Sensitivität von 0,6 ist doppelt so hoch wie bei der Nutzung der nicht-iterativen Konfiguration von JOANA. Für die Abdeckung bedeutet dies, dass fast zwei Drittel aller erwarteten Verletzungen gefunden werden.

Die vierte Durchführung wurde mit der Konfiguration *CodeQL-it* vollzogen. Wie in der iterativen Ausführung von JOANA wurden sieben Iterationen durchgeführt. Wie Abbildung 8.3 zeigt, spiegelt die Rückprojektion dieser Ausführung den Goldstandard wieder. Auch die beiden Rückprojektionen zu den Parametern $entry_{CCD}$ unterscheiden sich vom Goldstandard nicht, denn dieser zeigt zwar nur das Level [User] an, wurde jedoch durch Äquivalenzumformungen vereinfacht. Bei der Kopplung durch die Architekturanalyse werden somit genau die Verletzungen erkannt, die durch den Goldstandard erwartet werden und es ergibt sich eine Präzision von 1,0 und eine Sensitivität von 1,0 (vgl. Tabelle 8.5). Die Metriken zeigen, dass sich die Korrektheit durch die iterative Ausführung der Analyse mit CodeQL verbessert und die Präzision weiterhin vollständig ist.

Im direkten Vergleich der beiden genutzten Analysen wird deutlich, dass CodeQL eine höhere Genauigkeit in den Metriken Sensitivität und Präzision erreicht als JOANA.

Zusammengefasst ergibt die Evaluation der Genauigkeit, dass der iterative Ansatz für beide eingebundenen Quelltextanalysen korrektere Ergebnisse erzielt, indem die Abdeckung für vorliegende Vertraulichkeitsverletzungen erhöht wird (Q1.1). Dies ist insbesondere für eine feingranulare Spezifikation von Zugriffsberechtigungen sinnvoll, wie sie in die Fallstudie eingebaut wurden. Bezüglich der Präzision wurde kein deutlicher Unterschied zum nicht-iterativen Ansatz erzielt. Die Präzision bei JOANA verschlechterte sich um ca. 0,07. Für GQM-Frage (Q1.2) bedeutet dies, dass eine Verbesserung der Präzision nicht erreicht wurde. Insgesamt folgt für (G1), dass eine Genauigkeitsverbesserung der Vertraulichkeitsüberprüfung auf Architekturebene durch den iterativen Ansatz gegenüber dem nicht-iterativen Ansatz erreicht wurde.

8.4.2 Evaluationsergebnisse und Diskussion der Performanz

Das zweite GQM-Ziel (G2) der Evaluation betrifft die Performanz. Hierfür stellt Tabelle 8.6 die Messwerte dar, die während der Ausführung der vier Analysekonfigurationen für die in Abschnitt 8.1 genannten Metriken aufgezeichnet wurden.

Konfiguration	RAM-Verbrauch	$t_{Quelltextanalyse}$	t_{gesamt}	Speicherplatz
JOANA-non-it	1030 MB	101,867 s	103,357 s	103 KB
JOANA-it	1019 MB	103,985 s	109,945 s	132 KB
CodeQL-non-it	2011 MB	49,863 s	51,353 s	103 KB
CodeQL-it	2397 MB	345,409 s	349,879 s	122,7 KB

Tabelle 8.6: Gemessene Werte für die Metriken zu (G2): maximaler Arbeitsspeicherverbrauch, Programmlaufzeit der Quelltextanalyse, Programmlaufzeit der Analysekopplung, Speicherverbrauch der Ergebnismodelldateien.

Sowohl für die Ansätze, die JOANA nutzen, als auch die Ansätze, die CodeQL nutzen, ändert sich der Arbeitsspeicherverbrauch durch die iterative Ausführung nicht signifikant. Da die Analysen auch in den gewählten iterativen Konfigurationen das vollständige Modell laden, wird keine Verringerung des Arbeitsspeicherverbrauchs erzielt.

Die Gesamtlaufzeit der Analysenkopplung wird durch die Ausführung der Access Analysis jedes Mal konstant erhöht. Nur die Anzahl der zu untersuchenden Rückprojektionen unterscheidet die Ansätze. Während es in den nicht-iterativen Konfigurationen jeweils nur eine Rückprojektion gibt, müssen für die Konfiguration *JOANA-it* vier und für die Konfiguration *CodeQL-it* drei Rückprojektionen analysiert werden (vgl. Anzahl verschiedener Rückprojektionslevel in Abbildung 8.3). Die alleinige Laufzeit der Quelltextanalyse unterscheidet sich für die beiden JOANA-Ansätze kaum. Durch die iterative Ausführung wird nur ein minimaler Mehraufwand durch das mehrfache Ausführen der ProcessingSteps im Analyseframework erzeugt. Der zeitliche Aufwand für die interne Analyse in JOANA

bleibt gleich, da die *Entrypoints*, die zuvor alle hintereinander ausgeführt wurden, weiterhin hintereinander ausgeführt werden. Der einzige Unterschied ist, dass ihre Ausführung auf die Iterationen verteilt wird und sich die Reihenfolge ihrer Ausführung eventuell verändert. Demgegenüber wird beim Vergleich der CodeQL-Ansätze ein deutlicher Anstieg der Programmlaufzeit aufgezeichnet. Jede der sieben Iterationen benötigt ähnlich lange wie die gesamte nicht-iterative Ausführung alleine. Dies lässt sich dadurch begründen, dass der zeitliche Hauptanteil bei der Analyse durch CodeQL der Aufbau der internen Datenbank ist, die im weiteren Analyseverlauf mit Abfragen untersucht wird. Dieser Datenbank-Aufbau muss in jeder Iteration erneut ausgeführt werden, da dies in der Ausführung der Skripte mit dem Analysevorgang gekoppelt ist.

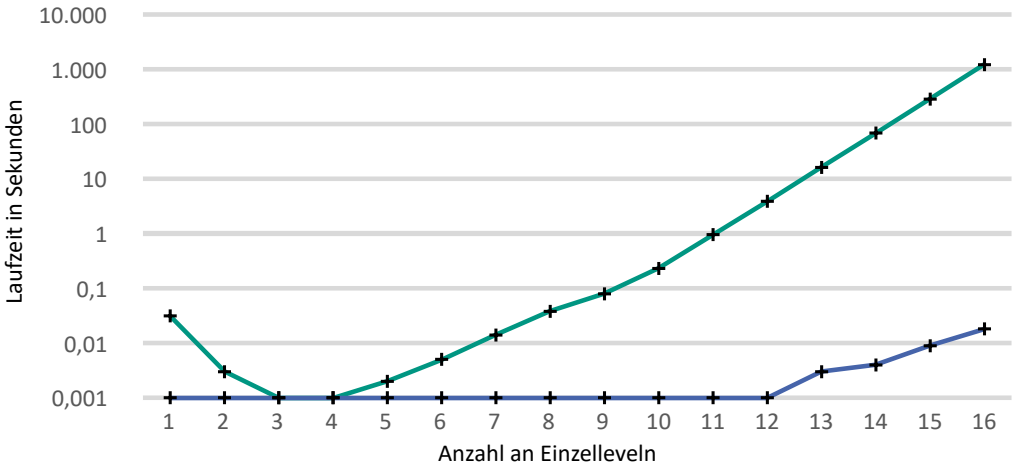
Zuletzt wird der Speicherplatz des rückprojizierten Modells verglichen. Unterschiede zwischen den Ansätzen ergeben sich einzig durch die Anzahl der Confidentiality-Dateien des Modells. Bei den nicht-iterativen Konfigurationen beträgt diese Anzahl 1 und bei den iterativen Konfigurationen entspricht sie der Anzahl an Iterationen. Im Rahmen der Fallstudie ist so kein erheblicher Speicherverbrauch erzeugt worden. Bei vielen verschiedenen ausgeprägten Sicherheitsleveln und somit vielen Iterationen kann diese Anzahl aber stark steigen.

Zusammengefasst ergibt die Evaluation der Performanz, dass es für JOANA keine großen Veränderungen in den Speicher und Laufzeitmetriken gibt, wenn der iterative Ansatz verwendet wird. Für CodeQL bedeutet der iterative Ansatz sogar einen deutlichen Anstieg der Ausführungsdauer. Die GQM-Fragen (Q2.1) bis (Q2.4) müssen alle verneint werden, da mit der ausgewählten Konfiguration des iterativen Ansatzes keine Verbesserung erzielt wurde. Das Ziel der Performanzverbesserung durch den iterativen Quelltextanalyseansatzes wurde somit nicht erreicht.

8.4.3 Evaluationsergebnisse und Diskussion der Skalierbarkeit

Für die Skalierbarkeit wurden immer größer werdende Eingaben für die beiden zu vergleichenden Ansätze zur Verbandsgenerierung automatisch und zufällig generiert und die Laufzeit in Millisekunden gemessen. Die Ergebnisse der Messreihe sind in Abbildung 8.4 einerseits als Datensatz dargestellt und zur Veranschaulichung in ein Kurvendiagramm in logarithmischer Darstellung übertragen worden. Die Messreihe wurde nach dem 16. Durchlauf beendet, da dieser für den naiven Ansatz eine Laufzeit von über 20 Minuten beanspruchte. Für den 17. Durchlauf wurde die festgelegte Obergrenze (vgl. Unterabschnitt 8.3.3) von 1 Stunde überschritten. Im Vergleich dazu beträgt die Laufzeit des optimierten Ansatzes für den 16. Durchlauf nur 18 Millisekunden. Es wird deutlich, dass der optimierte Ansatz zur Verbandsgenerierung über die Beschränkte Transitive Reduktion die Skalierbarkeit stark verbessert und in allen Durchläufen eine Laufzeit von wenigen Millisekunden erzielt.

Die Laufzeit des naiven Ansatzes beträgt für den 1. Durchlauf 31 Millisekunden und sinkt im 3. und 4. Durchlauf auf 1 Millisekunde herab. Da zur Generierung der Potenzmenge im naiven Ansatz eine externe Bibliothek (vgl. Guava [71]) verwendet wird, kann dieses Verhalten



Anzahl an Einzelleveln	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Anz. an ausgeprägten Leveln	1	2	3	5	8	12	18	27	41	62	93	140	210	315	473	710
Laufzeit in [s] B. T. R.	0,001	0,001	0,001	0,001	0,001	0,001	0,001	0,001	0,001	0,001	0,001	0,001	0,003	0,004	0,009	0,018
Laufzeit in [s] Naiv	0,031	0,003	0,001	0,001	0,002	0,005	0,014	0,038	0,079	0,23	0,951	3,866	16	68	287	1.210

Abbildung 8.4: Gemessene Laufzeiten der Verbandsgenerierung in logarithmischer Darstellung für den Ansatz der Beschränkten Transitiven Reduktion (B. T. R.) und für den naiven Ansatz Härings.

nur auf interne Optimierungen und Fallunterscheidungen dieser Bibliothek zurückgeführt werden.

Für GQM-Ziel (G3) ergibt die Evaluation das Erreichen einer höheren Skalierbarkeit durch den eigenen, in dieser Arbeit vorgestellten Ansatz.

8.5 Annahmen und Einschränkungen

In diesem Abschnitt wird diskutiert, welchen Einschränkungen der Ansatz und die Evaluation unterliegen und welche Annahmen über den Ansatz zu machen sind.

Der Ansatz erfordert es, dass der nutzende Software-Architekt Kenntnisse und Erfahrung im Modellieren besitzt, um ein System im PCM nachzubilden. Außerdem muss die Spezifikation von Vertraulichkeits-relevanten Modellelementen bekannt sein, um die Analyse zu nutzen.

Zudem ist das Ergebnis und dessen Genauigkeit von den zugrunde liegenden Analysen abhängig, die auf Quelltextebene und auf Architekturebene eingebunden wurden. Einerseits können interne Fehler dieser Analysen nicht behoben werden und andererseits kann die korrekte Funktionsweise dieser Analysen nicht belegt werden. Der iterative Ansatz ist hierbei jedoch konzeptionell so entworfen, dass eine Vielzahl von Anpassungsmöglichkeiten bereitstehen, um weitere Analysen einzubinden oder die Aufrufe an die Analysen anzupassen.

Eine weitere Limitierung des Ansatzes ist, dass die Verbesserung der Genauigkeit nur unter dem Sicherheitsbegriff der disjunktiven Level erzielt wurde und dies ein kleiner Bereich im Kontext der Vertraulichkeit ist. Zudem beschränkt sich die Analysierbarkeit auf diejenigen Elemente, die im Architekturmodell vorliegen. Um die Granularität anzupassen, muss das genutzte Architekturmodell erweitert werden. Dies ist jedoch in der verwendeten Analyse begründet und liegt nicht am eigenen Ansatz.

Für die Wahl des Iterationskonstruktes wurden in dieser Arbeit nur Elemente aus der Spezifikation betrachtet. Da der Ansatz aber Anpassungen am Iterationskonstrukt durch das Zusammenfügen mehrerer *Partitioner* in der Implementierung ermöglicht, können beispielsweise auch Systemelemente für Iterationen zerteilt werden. In der Domäne des Program-Slicing könnten so leichtgewichtigere Analysen pro Iteration ausgeführt werden. Dies zu untersuchen ist Aufgabe zukünftiger Arbeiten.

Das manuelle Übertragen der Ergebnisse der Quelltextanalyse in die Eingabe der Architekturanalyse erfordert derzeit mehrere Schritte der Dateiverschiebung und Generierung von passenden Eingabeformaten. Eine Automatisierung könnte in zukünftigen Arbeiten die Nutzbarkeit der Kopplung erhöhen.

Bezüglich der Evaluation entstehen zwei Einschränkungen. Als Erstes wurde die Nutzbarkeit des Ansatzes nicht evaluiert. Da es in dieser Arbeit jedoch vorrangig um die Konzepte und nicht um die Implementierung des iterativen Ansatzes geht, wurde dieses Evaluationsziel vernachlässigt. Als Zweites wurde nur eine konkrete iterative Konfiguration in ihrer Ausführung evaluiert. Da das Iterationskonstrukt der Quellenlevel aber als am besten geeignet hergeleitet wurde, um die Genauigkeit zu verbessern, ist diese Einschränkung angemessen (vgl. Abschnitt 5.4).

8.6 Gefährdungen der Validität

Im Folgenden wird nun nach den Richtlinien von Runeson und Höst [72] der Umgang mit Einschränkungen der Validität in der Evaluation beschrieben und wie diese minimiert wurden.

Interne Validität Die Interne Validität steht dafür, dass keine rivalisierenden Hypothesen durch andere Einflussfaktoren entstehen, die die Ergebnisse der Evaluation erklären. Ein Einflussfaktor der Ergebnisse ist z. B. die Messung durch den Experimentator selbst, was als Instrumentierungseffekt bezeichnet wird [72]. Die rivalisierende Hypothese ist in diesem Fall, dass Messergebnisse durch Erwartungen und die Sorgfalt des Experimentators verfälscht werden. Hier ist wichtig, dass die Objektivität bei der Datenerhebung trotz der Erwartungen an das Ergebnis gewahrt wird. Durch eine automatisierte Aufzeichnung der Messwerte für die Performanzmetriken und eine vollständige Übernahme aller Verletzungswarnungen für die Genauigkeitsmetriken wurde in dieser Arbeit eine objektive Erhebung der Daten genutzt, wodurch der Instrumentierungseffekt minimiert wird. Ein weiterer Einflussfaktor, der die Genauigkeitsverbesserung des Ansatzes erklären könnte, ist die Wahl der Fallstudie. Da das

Modell der Fallstudie bereits in verwandten Arbeiten [3] [9] [8] zur Evaluation genutzt wurde, wurde es nicht nur für den eigenen Ansatz erstellt. Auch die Erweiterungen, die im Rahmen der Evaluation am *TravelPlanner* durchgeführt wurden, wurden über Äquivalenzklassen von Flüssen hergeleitet. So ist die Genauigkeitsverbesserung des iterativen Ansatzes nur von den hergeleiteten Flusseigenschaften abhängig. Rivalisierende Hypothesen, die für die Genauigkeitsverbesserung andere Gründe bieten, können so eingeschränkt werden. Eine weitere Gefahr der internen Validität ist eine bloße Re-Implementierung eines vorhandenen Ansatzes. Dieses Risiko wurde minimiert, indem der entworfene Ansatz mit dem originalen Analyseansatz Härings [9] verglichen wurde und eine Verbesserung in der Genauigkeit erzielt.

Externe Validität Die Externe Validität bezieht sich auf die Generalisierbarkeit der Ergebnisse. Hierfür muss die Realitätsnähe der gewählten Fallstudie betrachtet werden. Wie bereits verwandte Arbeiten, die die Fallstudie nutzen, werden in dieser Arbeit Risiken in einem Reiseplanungssystem betrachtet, die sich an realen Gefahren für die Vertraulichkeit privater Daten orientieren. Zwar ist die Systemgröße in der Modellierung durch Abstraktion beschränkt und nicht realistisch. Doch laut Runeson und Höst [72] ist es in Fallstudien-orientierter Forschung wichtiger, ein tiefes Verständnis für ein Phänomen zu entwickeln, als repräsentativ zu sein. Sie sehen einen Vorteil in der Nutzung von bereits vorhandenen Fallstudien mit ähnlichen Charakteristiken, um eine Vergleichbarkeit und neue Erkenntnisse zu erzeugen.

Konstruktvalidität Die Konstruktvalidität soll belegen, dass die gemessenen Metriken zur Beantwortung der Forschungsfrage angemessen sind. Durch das Nutzen eines GQM-Plans [64] wird in dieser Masterarbeit die Nachvollziehbarkeit der Evaluationsdurchführung maximiert. Es wird hergeleitet, dass Metriken wie die Sensitivität und Präzision für die Erreichung der Ziele aussagekräftig sind [65]. Außerdem orientiert sich der Evaluationsplan an ähnlichen Arbeiten [3] [73], um ein angemessenes Vorgehen bei der Evaluation zu nutzen.

Reliabilität Der Begriff der Reliabilität bedeutet, dass die erzielten Ergebnisse nicht vom Forschenden abhängig sind, der diese in seiner Arbeit beschreibt. Hierfür muss die Evaluation *wiederholbar*, *reproduzierbar* und *replizierbar* sein. Für die Wiederholbarkeit, muss es demselben Forschenden möglich sein, die gleichen Ergebnisse erneut zu erzeugen. Da die Analyseausführung in den genutzten statischen Analysen deterministisch ist und die Eingabemodelle unverändert vorliegen, ist die Wiederholbarkeit gegeben. Für die Reproduzierbarkeit müssen andere Forschende den Ansatz nutzen können und die gleichen Ergebnisse erzielen. Dies ist durch die Veröffentlichung des Quelltexts des Ansatzes und der Modelle für die Fallstudie auf GitLab gegeben [74]. Die Replizierbarkeit beschreibt, dass das Vorgehen ausreichend beschrieben werden muss, um es anderen Forschenden zu ermöglichen, den Ansatz selbstständig nachzubilden und damit die gleichen Ergebnisse zu erzielen. Durch die theoretische Herleitung des Konzepts in Kapitel 5 und die Beschreibung der Implementierung in Kapitel 7 werden alle Informationen bereitgestellt, den vorgestellten iterativen Ansatz selbstständig umzusetzen.

9 Zusammenfassung und Ausblick

Diese Masterarbeit präsentiert einen neuen iterativen Ansatz und eine damit einhergehende Umsetzung, um in Softwaresystemen Verletzungen der Vertraulichkeit auf Architekturebene zu untersuchen. Hierbei wird ein Quelltextmodell mit den Spezifikationen aus der Architektur angereichert und eine iterative Informationsflussanalyse auf Quelltextebene ausgeführt. Die Ergebnisse werden in die Architektur zurückprojiziert und legen dort neue Verletzungen offen, die durch eine alleinige Untersuchung der Architekturebene verborgen geblieben wären.

Der Fokus dieser Arbeit liegt hinsichtlich möglicher Verbesserungen auf Skalierbarkeits-, Genauigkeits- und Performanzproblemen existierender Kopplungsansätze, die eine Quelltextanalyse nur einmalig durchführen und alle Informationen auf einmal in die Architekturanalyse transformieren. Durch partitionierte Eingaben wird die Quelltextanalyse im iterativen Ansatz nicht nur einmal ganzheitlich, sondern mehrmals ausgeführt.

Es wurde eine Bewertung mehrerer möglicher Iterationsarten durchgeführt und die Bedeutung der Mechanismen Iterationskonstrukt, Iterator und Abbruchbedingung herausgearbeitet. Da das Ziel der Genauigkeitsverbesserung priorisiert wurde, wurde für die umgesetzte Konfiguration des Ansatzes eine parallele Iterationsart mit getrennten Ergebnismodellen gewählt. Es wurde aussagenlogisch abgeleitet, dass durch diese Wahl eine Unterabschätzung von erwarteten Verletzungen, wie sie in verwandten Ansätzen auftritt, vermieden werden kann.

Durch die Evaluation konnte in einer Fallstudie gezeigt werden, dass die Genauigkeit der gefundenen Verletzungen durch den iterativen gegenüber dem nicht-iterativen Ansatz stark erhöht wurde. Die Sensitivität (Recall) von erwarteten Verletzungen wurde bei der Nutzung der Quelltextanalyse *JOANA* von 0,3 auf 0,95 erhöht und bei der Nutzung der Quelltextanalyse *CodeQL* von 0,6 auf 1,0.

Um die Skalierbarkeit des Ansatzes zu gewährleisten, wurde die Generierung von Sicherheitsverbänden überarbeitet. In der vorgestellten Optimierung der *Beschränkten Transitiven Reduktion* werden die Verbände nur noch aus den im Modell ausgeprägten Sicherheitsleveln erzeugt und nicht mehr vollständig über die Potenzmenge an möglichen Leveln dargestellt. Die Evaluation zeigt, dass im direkten Vergleich zum naiven Ansatz, der für einen gewählten Analyselauf ca. 20 Minuten brauchte, Laufzeiten im Millisekundenbereich garantiert werden konnten.

Die Implementierung des iterativen Ansatzes steht als Erweiterung eines Analyse-Frameworks auf GitLab [74] zur Verfügung.

In zukünftigen Arbeiten können drei Themen weiter untersucht werden: Die Umsetzung weiterer Iterationsarten, die Einbindung weiterer Analysen sowohl auf Quelltextebene als auch auf Architekturebene und eine Fortführung der Evaluation. Im zeitlichen Rahmen dieser Masterarbeit lag der Fokus auf einer spezifischen Iterationsart (Parallel - Getrennte Ergebnisdateien). Die Umsetzung weiterer Varianten vorgeschlagener Iterationsarten als konkrete Konfigurationen blieben deshalb zunächst unberücksichtigt.

Aufgrund zeitlicher Limitierungen dieser Masterarbeit konnten neben der Iterationsart (Parallel - Getrennte Ergebnisdateien) keine weiteren vorgeschlagenen Varianten als konkrete Konfigurationen umgesetzt werden.

Es stehen jedoch alle Mechanismen bereit, um diese in zukünftigen Arbeiten durchzuführen. Besonders die sequentielle Iterationsart bietet das Potential, große Systemelemente partitionierbar zu machen und so Skalierbarkeit und Performanz zu verbessern. In diesem Bereich würde das Assume-Guarantee-Reasoning eine größere Rolle spielen als in der ausgewählten Iterationsart für diese Masterarbeit.

Für zukünftige Arbeiten könnte der iterative Ansatz auf weitere Sicherheitsdomänen ausgeweitet werden und durch Einbindung von passenden Analysen auf Quelltext- und Architekturebene auch dort einen Nutzen für die Sicherheitsaussagen über ein System erzeugen. Für eine bessere Nutzbarkeit sind im Analyse-Framework außerdem weitere Anpassungen denkbar, um eine vollständig automatisierte Kopplung der Ebenen bereitzustellen.

In dieser Arbeit wurden nur die drei Verbesserungsziele Genauigkeit, Skalierbarkeit und Performanz in einer einzigen Fallstudie untersucht. Die Evaluation im Bereich der Nutzbarkeit und Erweiterbarkeit steht aus und kann in zukünftigen Arbeiten fortgeführt werden. Für weitere Iterationsarten und Sicherheitsdomänen sollte die Evaluation jedoch ebenfalls an die drei Ziele aus dieser Arbeit angelehnt werden.

Literatur

[1] *Verordnung (EU) 2016/679 des Europäischen Parlaments und des Rates vom 27. April 2016 zum Schutz natürlicher Personen bei der Verarbeitung personenbezogener Daten, zum freien Datenverkehr und zur Aufhebung der Richtlinie 95/46/EG (Datenschutz-Grundverordnung) (Text von Bedeutung für den EWR)*. Legislative Body: EP, CONSIL. 27. Apr. 2016. URL: http://data.europa.eu/eli/reg/2016/679/oj/deu (besucht am 30. 05. 2024).

[2] Yulkeidi Martínez, Cristina Cachero und Santiago Meliá. „Evaluating the Impact of a Model-Driven Web Engineering Approach on the Productivity and the Satisfaction of Software Development Teams“. In: *Web Engineering*. Hrsg. von Marco Brambilla, Takehiro Tokuda und Robert Tolksdorf. Bearb. von David Hutchison u. a. Bd. 7387. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 223–237. ISBN: 978-3-642-31752-1 978-3-642-31753-8. DOI: 10.1007/978-3-642-31753-8_17. URL: http://link.springer.com/10.1007/978-3-642-31753-8_17 (besucht am 11. 09. 2023).

[3] Stephan Seifermann u. a. „Detecting violations of access control and information flow policies in data flow diagrams“. In: *Journal of Systems and Software* 184 (Feb. 2022), S. 111138. ISSN: 01641212. DOI: 10.1016/j.jss.2021.111138. URL: https://linkinghub.elsevier.com/retrieve/pii/S0164121221002351 (besucht am 18. 09. 2023).

[4] Jan Jürjens. „UMLsec: Extending UML for Secure Systems Development“. In: *UML 2002 — The Unified Modeling Language*. Hrsg. von Jean-Marc Jézéquel, Heinrich Hussmann und Stephen Cook. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2002, S. 412–425. ISBN: 978-3-540-45800-5. DOI: 10.1007/3-540-45800-X_32.

[5] Max E. Kramer u. a. *Model-Driven Specification and Analysis of Confidentiality in Component-Based Systems*. 12. ISSN: 2190-4782 Num Pages: 32 Series: Karlsruhe Reports in Informatics Volume: 2017. Karlsruher Institut für Technologie (KIT), 2017. DOI: 10.5445/IR/1000076957.

[6] Paul Muntean u. a. „Automated Detection of Information Flow Vulnerabilities in UML State Charts and C Code“. In: *Reliability and Security - Companion 2015 IEEE International Conference on Software Quality*. Reliability and Security - Companion 2015 IEEE International Conference on Software Quality. Aug. 2015, S. 128–137. DOI: 10.1109/QRS-C.2015.30.

[7] Sven Matthias Peldszus. *Security Compliance in Model-driven Development of Software Systems in Presence of Long-Term Evolution and Variants*. Wiesbaden: Springer Fachmedien, 2022. ISBN: 978-3-658-37664-2 978-3-658-37665-9. DOI: 10.1007/978-3-658-37665-9. URL: https://link.springer.com/10.1007/978-3-658-37665-9 (besucht am 11. 09. 2023).

[8] Kuzman Katkalov u. a. „Model-driven development of information flow-secure systems with IFlow“. In: (2013). URL: https://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/docId/53957 (besucht am 22. 09. 2023).

[9] Johannes Häring. „Enabling the Information Transfer between Architecture and Source Code for Security Analysis“. In: (2021). Publisher: Karlsruher Institut für Technologie (KIT). DOI: 10.5445/IR/1000142571. URL: https://publikationen.bibliothek.kit.edu/1000142571 (besucht am 11. 09. 2023).

[10] Kuzman Katkalov. „Ein modellgetriebener Ansatz zur Entwicklung informationsflusssicherer Systeme“. Diss. Universität Augsburg, 2017. 285 S. URL: https://web.archive.org/web/20210926154149/https://opus.bibliothek.uni-augsburg.de/opus4/frontdoor/index/index/docId/4339 (besucht am 10. 05. 2024).

[11] Thomas Stahl und Jorn Bettin, Hrsg. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. 2., aktualisierte und erw. Aufl. Heidelberg: dpunkt-Verl, 2007. 441 S. ISBN: 978-3-89864-448-8.

[12] C. Atkinson und T. Kuhne. „Model-driven development: a metamodeling foundation“. In: *IEEE Software* 20.5 (Sep. 2003), S. 36–41. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2003.1231149. URL: https://ieeexplore.ieee.org/document/1231149/ (besucht am 11. 09. 2023).

[13] Ralf H. Reussner u. a., Hrsg. *Modeling and Simulating Software Architectures – The Palladio Approach*. MIT Press, 2016. 408 S. ISBN: 978-0-262-03476-0.

[14] Herbert Stachowiak. *Allgemeine Modelltheorie*. Wien New York: Springer, 1973. ISBN: 978-3-211-81106-1 978-0-387-81106-2.

[15] Holger Giese. *Models in Software Engineering: Workshops and Symposia at MoDELS 2007, Nashville, TN, USA, September 30 - October 5, 2007, Reports and Revised Selected Papers*. Unter Mitarb. von MODELS 2007. Computer Science (Springer-11645; ZDB-2-SCS) 5002. Berlin, Heidelberg: Springer Berlin Heidelberg Springer e-books, 2008. ISBN: 978-3-540-69073-3.

[16] Kevin Lano und Shekoufeh Kolahdouz-Rahimi. „Model-Driven Development of Model Transformations“. In: *Theory and Practice of Model Transformations*. Hrsg. von Jordi Cabot und Eelco Visser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2011, S. 47–61. ISBN: 978-3-642-21732-6. DOI: 10.1007/978-3-642-21732-6_4.

[17] Greg Brunet u. a. „A manifesto for model merging“. In: *Proceedings of the 2006 international workshop on Global integrated model management*. ICSE06: International Conference on Software Engineering. Shanghai China: ACM, 22. Mai 2006, S. 5–12. ISBN: 978-1-59593-410-9. DOI: 10.1145/1138304.1138307. URL: https://dl.acm.org/doi/10.1145/1138304.1138307 (besucht am 22. 09. 2023).

[18] Th Reiter u. a. „Model integration through mega operations“. In: *Workshop on Model-driven Web Engineering*. Bd. 20. Citeseer, 2005.

[19] E. Kindler und Robert Wagner. „Triple Graph Grammars : Concepts , Extensions , Implementations , and Application Scenarios“. In: 2007. URL: https://www.semanticscholar.org/paper/Triple-Graph-Grammars-%3A-Concepts-%2C-Extensions-%2C-%2C-Kindler-Wagner/54cfdbba4488b6a025a5c8cc6270272bcf554679 (besucht am 22. 09. 2023).

[20] Hartmut Ehrig u. a. „Information Preserving Bidirectional Model Transformations“. In: Bd. 4422. 24. März 2007, S. 72–86. ISBN: 978-3-540-71288-6. DOI: 10.1007/978-3-540-71289-3_7.

[21] Hartmut Ehrig. *Graph transformations: 4th international conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008 proceedings*. Unter Mitarb. von ICGT 2008. Lecture notes in computer science 5214. Berlin: Springer, 2008. ISBN: 978-3-540-87405-8.

[22] A. Avizienis u. a. „Basic concepts and taxonomy of dependable and secure computing“. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), S. 11–33. ISSN: 1545-5971. DOI: 10.1109/TDSC.2004.2. URL: http://ieeexplore.ieee.org/document/1335465/ (besucht am 14. 09. 2023).

[23] Katja Tuma, Riccardo Scandariato und Musard Balliu. „Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis“. In: *2019 IEEE International Conference on Software Architecture (ICSA)*. 2019 IEEE International Conference on Software Architecture (ICSA). Hamburg, Germany: IEEE, März 2019, S. 191–200. ISBN: 978-1-72810-528-4. DOI: 10.1109/ICSA.2019.00028. URL: https://ieeexplore.ieee.org/document/8703905/ (besucht am 18. 09. 2023).

[24] Dorothy E. Denning. „A lattice model of secure information flow“. In: *Communications of the ACM* 19.5 (1. Mai 1976), S. 236–243. ISSN: 0001-0782. DOI: 10.1145/360051.360056. URL: https://dl.acm.org/doi/10.1145/360051.360056 (besucht am 14. 09. 2023).

[25] D.E. Bell und Leonard J. LaPadula. *Secure Computer Systems: Mathematical Foundations*. Section: Technical Reports. 1973. URL: https://apps.dtic.mil/sti/citations/AD0770768 (besucht am 17. 09. 2023).

[26] K. J. Biba. *Integrity Considerations for Secure Computer Systems*. Section: Technical Reports. URL: https://apps.dtic.mil/sti/citations/ADA039324 (besucht am 17. 09. 2023).

[27] J. A. Goguen und J. Meseguer. „Security Policies and Security Models“. In: *1982 IEEE Symposium on Security and Privacy*. 1982 IEEE Symposium on Security and Privacy. ISSN: 1540-7993. Apr. 1982, S. 11–11. DOI: 10.1109/SP.1982.10014.

[28] Gavin Keighren. „Restricting information flow in security APIs via typing“. In: (27. Juni 2014). Accepted: 2014-06-20T15:17:38Z Publisher: The University of Edinburgh. URL: https://era.ed.ac.uk/handle/1842/8963 (besucht am 18. 09. 2023).

[29] D.E. Bell. „Looking back at the Bell-La Padula model“. In: *21st Annual Computer Security Applications Conference (ACSAC'05)*. 21st Annual Computer Security Applications Conference (ACSAC'05). ISSN: 1063-9527. Dez. 2005, 15 pp.–351. DOI: 10.1109/CSAC.2005.37.

[30] Garrett Birkhoff. *Lattice Theory*. Google-Books-ID: ePqVAwAAQBAJ. American Mathematical Soc., 31. Dez. 1940. 434 S. ISBN: 978-0-8218-1025-5.

[31] Dimitra Giannakopoulou, Kedar S. Namjoshi und Corina S. Păsăreanu. „Compositional Reasoning“. In: *Handbook of Model Checking*. Hrsg. von Edmund M. Clarke u. a. Cham: Springer International Publishing, 2018, S. 345–383. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_12. URL: https://doi.org/10.1007/978-3-319-10575-8_12 (besucht am 22. 09. 2023).

[32] D. Giannakopoulou, C.S. Pasareanu und J.M. Cobleigh. „Assume-guarantee verification of source code with design-level assumptions“. In: *Proceedings. 26th International Conference on Software Engineering*. Proceedings. 26th International Conference on Software Engineering. ISSN: 0270-5257. Mai 2004, S. 211–220. DOI: 10.1109/ICSE.2004.1317443.

[33] Chris Chilton, Bengt Jonsson und Marta Kwiatkowska. „Assume-Guarantee Reasoning for Safe Component Behaviours“. In: *Formal Aspects of Component Software*. Hrsg. von Corina S. Păsăreanu und Gwen Salaün. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, S. 92–109. ISBN: 978-3-642-35861-6. DOI: 10.1007/978-3-642-35861-6_6.

[34] Cong Liu u. a. „Assume-Guarantee Reasoning with Scheduled Components“. In: *NASA Formal Methods: 14th International Symposium, NFM 2022, Pasadena, CA, USA, May 24–27, 2022, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 24. Mai 2022, S. 355–372. ISBN: 978-3-031-06772-3. DOI: 10.1007/978-3-031-06773-0_19. URL: https://doi.org/10.1007/978-3-031-06773-0_19 (besucht am 19. 09. 2023).

[35] Sagar Chaki u. a. „Automated Assume-Guarantee Reasoning for Simulation Conformance“. In: *Computer Aided Verification*. Hrsg. von Kousha Etessami und Sriram K. Rajamani. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, S. 534–547. ISBN: 978-3-540-31686-2. DOI: 10.1007/11513988_51.

[36] Darren Cofer u. a. „Compositional Verification of Architectural Models“. In: *NASA Formal Methods*. Hrsg. von Alwyn E. Goodloe und Suzette Person. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2012, S. 126–140. ISBN: 978-3-642-28891-3. DOI: 10.1007/978-3-642-28891-3_13.

[37] *JOANA (Java Object-sensitive ANAlysis) - Information Flow Control Framework for Java*. Archive Location: KIT. URL: https://pp.ipd.kit.edu/projects/joana/ (besucht am 11. 09. 2023).

[38] Elisavet Kozyri u. a. „JRIF: Reactive Information Flow Control for Java“. In: *Foundations of Security, Protocols, and Equational Reasoning*. Hrsg. von Joshua D. Guttman u. a. Bd. 11565. Series Title: Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, S. 70–88. ISBN: 978-3-030-19051-4 978-3-030-19052-1.

DOI: 10.1007/978-3-030-19052-1_7. URL: http://link.springer.com/10.1007/978-3-030-19052-1_7 (besucht am 22. 09. 2023).

[39] Wolfgang Ahrendt u. a., Hrsg. *Deductive Software Verification – The KeY Book*. Bd. 10001. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2016. ISBN: 978-3-319-49811-9 978-3-319-49812-6. DOI: 10.1007/978-3-319-49812-6. URL: http://link.springer.com/10.1007/978-3-319-49812-6 (besucht am 11. 09. 2023).

[40] *About CodeQL — CodeQL*. URL: https://codeql.github.com/docs/codeql-overview/about-codeql/ (besucht am 11. 09. 2023).

[41] *Confidentiality4CBSE*. original-date: 2017-02-22T15:48:21Z. 13. Juli 2021. URL: https://github.com/KASTEL-SCBS/Confidentiality4CBSE (besucht am 22. 09. 2023).

[42] Vittorio Cortellessa, Romina Eramo und Michele Tucci. „From software architecture to analysis models and back: Model-driven refactoring aimed at availability improvement“. In: *Information and Software Technology* 127 (1. Nov. 2020), S. 106362. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2020.106362. URL: https://www.sciencedirect.com/science/article/pii/S0950584920301300 (besucht am 06. 09. 2023).

[43] *The GRaViTY-Tool*. GRaViTY Tool. URL: https://gravity-tool.org/ (besucht am 22. 09. 2023).

[44] Robert Heinrich u. a. „An Architectural Model-Based Approach to Quality-Aware DevOps in Cloud Applicationsc“. In: *Software Architecture for Big Data and the Cloud. Ed.: I. Mistrik* (2017). ISBN: 9780128054673, S. 69. URL: https://publikationen.bibliothek.kit.edu/1000098502 (besucht am 22. 09. 2023).

[45] Kuzman Katkalov u. a. „Evaluation of Jif and Joana as Information Flow Analyzers in a Model-Driven Approach“. In: *Data Privacy Management and Autonomous Spontaneous Security*. Hrsg. von Roberto Di Pietro u. a. Bearb. von David Hutchison u. a. Bd. 7731. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, S. 174–186. ISBN: 978-3-642-35889-0 978-3-642-35890-6. DOI: 10.1007/978-3-642-35890-6_13. URL: https://link.springer.com/10.1007/978-3-642-35890-6_13 (besucht am 22. 09. 2023).

[46] Adam Shostack. *Threat modeling: designing for security*. Indianapolis, Ind: Wiley, 2014. 590 S. ISBN: 978-1-118-80999-0.

[47] Christopher Gerking, David Schubert und Eric Bodden. „Model Checking the Information Flow Security of Real-Time Systems“. In: *Engineering Secure Software and Systems*. Hrsg. von Mathias Payer, Awais Rashid und Jose M. Such. Cham: Springer International Publishing, 2018, S. 27–43. ISBN: 978-3-319-94496-8. DOI: 10.1007/978-3-319-94496-8_3.

[48] Gregor Snelting u. a. „Checking probabilistic noninterference using JOANA“. In: *it - Information Technology* 56.6 (28. Dez. 2014), S. 280–287. ISSN: 1611-2776, 2196-7032. DOI: 10.1515/itit-2014-1051. URL: https://www.degruyter.com/document/doi/10.1515/itit-2014-1051/html (besucht am 30. 05. 2024).

[49] Tobias Runge u. a. „Lattice-Based Information Flow Control-by-Construction for Security-by-Design“. In: *Proceedings of the 8th International Conference on Formal Methods in Software Engineering*. FormaliSE '20. New York, NY, USA: Association for Computing Machinery, 7. Okt. 2020, S. 44–54. ISBN: 978-1-4503-7071-4. DOI: 10.1145/3372020.3391565. URL: https://doi.org/10.1145/3372020.3391565 (besucht am 30. 05. 2024).

[50] Benjamin Livshits, John Whaley und Monica S. Lam. „Reflection Analysis for Java“. In: *Programming Languages and Systems*. Hrsg. von Kwangkeun Yi. Berlin, Heidelberg: Springer, 2005, S. 139–160. ISBN: 978-3-540-32247-4. DOI: 10.1007/11575467_11.

[51] Eric Bodden u. a. „Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders“. In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. New York, NY, USA: Association for Computing Machinery, 21. Mai 2011, S. 241–250. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985827. URL: https://dl.acm.org/doi/10.1145/1985793.1985827 (besucht am 30. 05. 2024).

[52] Kateryna Yurchenko u. a. „Architecture-driven reduction of specification overhead for verifying confidentiality in component-based software systems“. In: *ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS 2017), Austin, TX, September 17-22, 2017. Ed.: L. Burgueño* (2017), S. 321. ISSN: 1613-0073. URL: https://publikationen.bibliothek.kit.edu/1000080494 (besucht am 30. 05. 2024).

[53] Siegfried Rasthofer, Steven Arzt und Eric Bodden. *A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks*. NDSS Symposium. 22. Feb. 2014. URL: https://www.ndss-symposium.org/ndss2014/ndss-2014-programme/machine-learning-approach-classifying-and-categorizing-android-sources-and-sinks/ (besucht am 30. 05. 2024).

[54] Johannes Lerch u. a. „FlowTwist: efficient context-sensitive inside-out taint analysis for large codebases“. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. New York, NY, USA: Association for Computing Machinery, 11. Nov. 2014, S. 98–108. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635878. URL: https://doi.org/10.1145/2635868.2635878 (besucht am 30. 05. 2024).

[55] Steven Arzt u. a. „FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps“. In: *ACM SIGPLAN Notices* 49.6 (9. Juni 2014), S. 259–269. ISSN: 0362-1340. DOI: 10.1145/2666356.2594299. URL: https://doi.org/10.1145/2666356.2594299 (besucht am 30. 05. 2024).

[56] Cristian-Alexandru Staicu u. a. „An Empirical Study of Information Flows in Real-World JavaScript“. In: *Proceedings of the 14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security*. PLAS'19. New York, NY, USA: Association for Computing Machinery, 15. Nov. 2019, S. 45–59. ISBN: 978-1-4503-6836-0. DOI: 10.1145/3338504.3357339. URL: https://doi.org/10.1145/3338504.3357339 (besucht am 30. 05. 2024).

[57] Dorothy Elizabeth Robling Denning. *Cryptography and data security*. USA: Addison-Wesley Longman Publishing Co., Inc., 1982. 414 S. ISBN: 978-0-201-10150-8.

[58] R.S. Sandhu. „Lattice-based access control models“. In: *Computer* 26.11 (Nov. 1993). Conference Name: Computer, S. 9–19. ISSN: 1558-0814. DOI: 10.1109/2.241422.

[59] John McLean. „A comment on the 'basic security theorem' of Bell and LaPadula“. In: *Information Processing Letters* 20.2 (Feb. 1985), S. 67–70. ISSN: 00200190. DOI: 10.1016/0020-0190(85)90065-1. URL: https://linkinghub.elsevier.com/retrieve/pii/0020019085900651 (besucht am 30. 05. 2024).

[60] A. V. Aho, M. R. Garey und J. D. Ullman. „The Transitive Reduction of a Directed Graph“. In: *SIAM Journal on Computing* 1.2 (Juni 1972), S. 131–137. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/0201008. URL: http://epubs.siam.org/doi/10.1137/0201008 (besucht am 28. 05. 2024).

[61] Oliver Deiser. *Einführung in die Mengenlehre: die Mengenlehre Georg Cantors und ihre Axiomatisierung durch Ernst Zermelo*. 2., verb. und erw. Aufl. Springer-Lehrbuch. Berlin Heidelberg: Springer, 2004. 551 S. ISBN: 978-3-540-20401-5.

[62] Thomas H. Cormen u. a. *Introduction to algorithms*. 3rd ed. OCLC: ocn311310321. Cambridge, Mass: MIT Press, 2009. 1292 S. ISBN: 978-0-262-03384-8 978-0-262-53305-8.

[63] Victor R. Basili und David M. Weiss. „A Methodology for Collecting Valid Software Engineering Data“. In: *IEEE Transactions on Software Engineering* SE-10.6 (Nov. 1984). Conference Name: IEEE Transactions on Software Engineering, S. 728–738. ISSN: 1939-3520. DOI: 10.1109/TSE.1984.5010301.

[64] Victor R Basili, Gianluigi Caldiera und H Dieter Rombach. „THE GOAL QUESTION METRIC APPROACH“. In: *Encyclopedia of Software Engineering, John Wiley & Sons* 2 (1994), S. 528–532. URL: https://scholar.google.com/scholar_lookup?title=The%20goal%20question%20metric%20approach&publication_year=1994&author=V.R.%20Basili&author=G.%20Caldiera&author=H.D.%20Rombach (besucht am 07. 05. 2024).

[65] Christian Hammer und Gregor Snelting. „Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs“. In: *International Journal of Information Security* 8.6 (Dez. 2009), S. 399–422. ISSN: 1615-5262, 1615-5270. DOI: 10.1007/s10207-009-0086-1. URL: http://link.springer.com/10.1007/s10207-009-0086-1 (besucht am 24. 05. 2024).

[66] David M. W. Powers. *Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation*. 10. Okt. 2020. DOI: 10.48550/arXiv.2010.16061. arXiv: 2010.16061[cs,stat]. URL: http://arxiv.org/abs/2010.16061 (besucht am 23. 09. 2023).

[67] *Erfassen von Daten mit Windows-Leistungsüberwachung*. URL: https://help.tableau.com/current/server/de-de/perf_collect_perfmon.htm (besucht am 10. 05. 2024).

[68] *KASTEL-SCBS/haskalladio: Haskell and Prolog Protoype Implementations of the Palladio Model Access Analysis*. URL: https://github.com/KASTEL-SCBS/haskalladio (besucht am 25. 05. 2024).

[69] *JProfiler - Overview*. URL: https://www.ej-technologies.com/products/jprofiler/overview.html (besucht am 10. 05. 2024).

[70] *JConsole verwenden - IBM Dokumentation*. URL: https://www.ibm.com/docs/de/semeru-runtime-ce-z/11?topic=reference-using-jconsole (besucht am 10. 05. 2024).

[71] *Guava: Google Core Libraries for Java*. original-date: 2014-05-29T16:23:17Z. 31. Mai 2024. URL: https://github.com/google/guava (besucht am 31. 05. 2024).

[72] Per Runeson und Martin Höst. „Guidelines for conducting and reporting case study research in software engineering“. In: *Empirical Software Engineering* 14.2 (1. Apr. 2009), S. 131–164. ISSN: 1573-7616. DOI: 10.1007/s10664-008-9102-8. URL: https://doi.org/10.1007/s10664-008-9102-8 (besucht am 10. 05. 2024).

[73] Stephan Seifermann, Robert Heinrich und Ralf Reussner. „Data-Driven Software Architecture for Analyzing Confidentiality“. In: *2019 IEEE International Conference on Software Architecture (ICSA)*. 2019 IEEE International Conference on Software Architecture (ICSA). Hamburg, Germany: IEEE, März 2019, S. 1–10. ISBN: 978-1-72810-528-4. DOI: 10.1109/ICSA.2019.00009. URL: https://ieeexplore.ieee.org/document/8703910/ (besucht am 26. 05. 2024).

[74] Jonas Lehmann. *GitLab Repository MA thesis Jonas Lehmann - KIT / KASTEL / Software Design and Quality (SDQ)*. URL: https://gitlab.kit.edu/kit/kastel/sdq/stud/abschlussarbeiten/masterarbeiten/jonaslehmann (besucht am 30. 05. 2024).