BACHELORARBEIT

Bestimmung von Datenflüssen in Java-basierten Anwendungen für Sicherheitsüberprüfungen

---

Autor:
Philipp Schönemann

Gutachter:
Prof. Dr. Rainer Koschke
Dr. Karsten Sohr

Betreuer:
Bernhard Berger

AG Softwaretechnik

14. Juni 2021

Universität Bremen

Offizielle Erklärungen von

Nachname: Schönemann Vorname: Philipp

Matrikelnr.: 4435998

A) Eigenständigkeitserklärung

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

B) Erklärung zur Veröffentlichung von Bachelor- oder Masterarbeiten

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

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

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

☐ Ich bin damit einverstanden, dass meine Abschlussarbeit nach 30 Jahren (gem. §7 Abs. 2 BremArchivG) im Universitätsarchiv für wissenschaftliche Zwecke von Dritten eingesehen werden darf.

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

C) Einverständniserklärung über die Bereitstellung und Nutzung der Bachelorarbeit / Masterarbeit / Hausarbeit in elektronischer Form zur Überprüfung durch Plagiatssoftware

Eingereichte Arbeiten können mit der Software *Plagscan* auf einen hauseigenen Server auf Übereinstimmung mit externen Quellen und der institutionseigenen Datenbank untersucht werden. Zum Zweck des Abgleichs mit zukünftig zu überprüfenden Studien- und Prüfungsarbeiten kann die Arbeit dauerhaft in der institutionseigenen Datenbank der Universität Bremen gespeichert werden.

☐ Ich bin damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum Zweck der Überprüfung auf Plagiate auf den *Plagscan*-Server der Universität Bremen hochgeladen wird.

☐ Ich bin ebenfalls damit einverstanden, dass die von mir vorgelegte und verfasste Arbeit zum o.g. Zweck auf dem *Plagscan*-Server der Universität Bremen hochgeladen u. dauerhaft auf dem *Plagscan*-Server gespeichert wird.

☒ Ich bin nicht damit einverstanden, dass die von mir vorgelegte u. verfasste Arbeit zum o.g. Zweck auf dem *Plagscan*-Server der Universität Bremen hochgeladen u. dauerhaft gespeichert wird.

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

14.06.21, Bremen
Datum, Ort


Unterschrift

UNIVERSITÄT BREMEN

Abstract

AG Softwaretechnik

Bachelor of Science

Bestimmung von Datenflüssen in Java-basierten Anwendungen für Sicherheitsüberprüfungen

von Philipp Schönemann

Datensicherheit ist in der heutigen Zeit ein bedeutsames Thema. Neben der Aufbewahrung der Daten spielt auch deren Verarbeitung innerhalb von Softwaresystemen eine Rolle, da Schwachstellen von Angreifern ausgenutzt werden können. Die architekturelle Risikoanalyse spürt solche Schwachstellen auf, ist aber kostenintensiv und erfordert Expertenwissen. Das Ziel der vorliegenden Arbeit ist die Implementierung einer Datenflussanalyse, um Datenflüsse in Java-basierten Anwendungen zu bestimmen und in erweiterte Datenflussdiagramme zu überführen, sodass diese zur Erkennung von unsicheren Datenflüssen sowie anderen Schwachstellen verwendet werden können. Um das Ziel zu erreichen, wurde eine interprozedurale Datenflussanalyse auf Basis von Soot und VASCO implementiert, dessen Ergebnisse mit Hilfe von ArchSec in Datenflussdiagramme überführt werden. Die anschließende Evaluation anhand von drei Testanwendungen zeigt, dass die Analyse im Durchschnitt 86% der unsicheren Datenflüsse erfolgreich und im Vergleich zu LAPSE+, einem ähnlichen Analysewerkzeug, 110% mehr unsichere Datenflüsse ermittelt und rekonstruiert hat. Allerdings bereiten Datenflüsse, bei denen Arrays involviert sind, der Analyse Schwierigkeiten. Ungeachtet dessen eignet sich die Analyse für erste manuelle Sicherheitsreviews und kann bei einer architekturellen Risikoanalyse unterstützend eingesetzt werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ArchSec	**Arch**itectural **Sec**urity Tool Suite
DFD	**D**aten**f**luss**d**iagramm
EDFD	**E**rweitertes **D**aten**f**luss**d**iagramm
GUI	**G**raphical **U**ser **I**nterface
IDE	**I**ntegrated **D**evelopment **E**nvironment
IT	**I**nformations**t**echnik
JAR	**J**ava **Ar**chive
JSON	**J**ava**S**cript **O**bject **N**otation
KFG	**K**ontroll**f**luss**g**raph
LAPSE+	**L**ightweight **A**nalysis for **P**rogram **S**ecurity in **E**clipse +
SLOC	**S**ource **L**ines **o**f **C**ode
SQL	**S**tructured **Q**uery **L**anguage
UML	**U**nified **M**odeling **L**anguage
XML	**E**xtensible **M**arkup **L**anguage

Kapitel 1

Einleitung

Datensicherheit in der Informationstechnik (IT) ist ein zentrales Thema, welches die Menschen in der heutigen Zeit bewegt. Doch nicht nur die sichere Aufbewahrung von Daten spielt eine Rolle, sondern auch unsicherer für Attacken anfälliger Code sowie Datenflüsse innerhalb von Softwaresystemen, da diese zu unsicheren Zuständen führen können, sodass sensible Daten wie beispielsweise Passwörter oder Kreditkartendaten offengelegt werden und Angreifer ein leichtes Spiel haben, diese zu manipulieren oder zu stehlen. Daher ist es nicht verwunderlich, dass Code-Reviews neben der architekturellen Risikoanalyse bereits seit Jahren zur guten Praxis gehören [3].

1.1 Ziel der Arbeit

Ziel dieser Arbeit ist es, eine statische Programmanalyse für Java-basierte Anwendungen zu entwickeln, die ausgehend von einem frei wählbaren Einstiegspunkt den Datenfluss beliebiger Methodenparameter verfolgt und ermittelt, in welche Methoden diese Parameter fließen. Anschließend sollen die Ergebnisse automatisiert in erweiterten Datenflussdiagrammen (EDFDs) überführt werden, um mit Hilfe einer Sicherheitswissensdatenbank unsichere Datenflüsse von Interesse zu extrahieren oder andere Sicherheitslücken zu erkennen. Da die architekturelle Risikoanalyse nicht nur mit einem hohen Maß an Expertenwissen, sondern auch mit hohen Kosten verbunden ist, wird sie in kleineren wie auch in mittelständischen Unternehmen in der Regel vernachlässigt [1]. Die entwickelte Analyse ermöglicht und unterstützt insbesondere solche Unternehmen dabei, einen leichteren Zugang zu finden.

1.2 Beitrag der Arbeit

Beitrag der Arbeit ist die Entwicklung einer interprozeduralen Datenflussanalyse für Java-basierte Anwendungen, die alle Datenflüsse von sensitiven Daten, ausgehend von Methodenparametern, ermittelt und eines Konverters, der die Analyseergebnisse in ein EDFD überführt. Der generierte EDFD ist hierarchisch strukturiert und enthält mit Hilfe von Annotationen relevante Informationen wie sensitive Parameter, unsichere Senken und alle ermittelten Datenflüsse, sodass er sowohl für das manuelle Betrachten als auch für Sicherheitsüberprüfungen geeignet ist. Weiterhin wird eine in der Abfragesprache Cypher formulierte Regel für eine Sicherheitswissensdatenbank vorgestellt, die auf einen EDFD angewendet werden kann, um alle durch die Analyse identifizierten unsicheren Datenflüsse extrahieren und anzeigen zu lassen. Unsicher bedeutet, dass ein Datenfluss von einem sensitiven Parameter in eine unsichere Senke führt.

1.3 Aufbau der Arbeit

Zunächst werden in Kapitel 2 die Grundlagen erläutert, die für das Verständnis der Folgekapitel und dem Entwicklungsprozess notwendig sind. Kapitel 3 gibt anschließend eine grobe Übersicht über alle Frameworks sowie Bibliotheken und beschreibt die grundlegende Architektur der entwickelten Analyse, bevor am Ende des Kapitels detailliert auf die Implementierung und das Überführen der Ergebnisse in EDFDs eingegangen wird. Kapitel 4 gibt danach Auskunft darüber, wie die Grundfunktionalität der Analyse mit Hilfe von Unit- und Integrationstests getestet wird. Im Anschluss wird die Analyse anhand von drei konstruierten Testanwendungen evaluiert. Am Ende des Kapitels erfolgt ein Vergleich mit einem Analysewerkzeug namens Lightweight Analysis for Program Security in Eclipse + (LAPSE+), welches Ähnlichkeit zur entwickelten Analyse aufweist. Zum Abschluss werden in Kapitel 5 die wesentlichen Ergebnisse zusammengefasst, ein Fazit gezogen und ein Ausblick auf mögliche zukünftige Entwicklungen gegeben.

Kapitel 2

Grundlagen

In diesem Kapitel werden Themen und Begrifflichkeiten eingeführt, die die theoretische Grundlage für das tiefere Verständnis der Arbeit bilden. Insbesondere werden die beiden Kernthemen erläutert: die Datenflussanalyse und das Datenflussdiagramm.

2.1 Programmanalyse

Laut Wögerer [13] handelt es sich bei einer Programmanalyse um ein Prozess, bei dem das Verhalten oder andere Eigenschaften eines Programms automatisiert analysiert werden. Im Wesentlichen kann zwischen zwei verschiedenen Arten unterschieden werden: statische Programmanalyse und dynamische Programmanalyse. Der Unterschied liegt darin, dass bei einer statischen Analyse das Programm nicht ausgeführt wird und lediglich der Quelltext bzw. der kompilierte Byte- oder Maschinencode und andere Metadaten vorliegen. Eine dynamische Analyse hingegen besitzt zusätzlich Zugriff auf Laufzeitinformationen wie den momentanen Programmzustand.

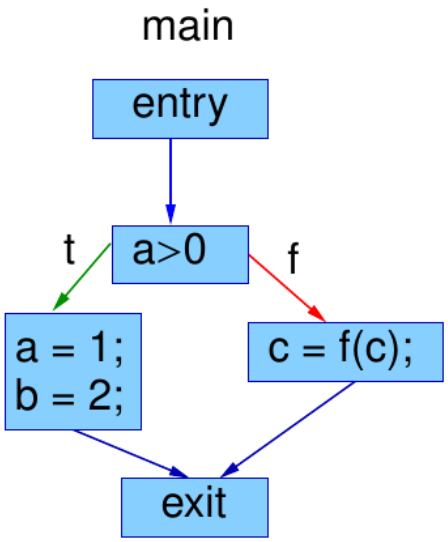
2.2 Datenflussanalyse

Wögerer [13] schreibt, dass die Datenflussanalyse ein Prozess ist, der darauf abzielt, Laufzeitinformationen, wie z. B. die mögliche Wertemenge von Variablen, über Daten zu sammeln, ohne das betrachtete Programm auszuführen. Somit handelt es sich um eine statische Programmanalyse. Nach Brabrand et al. [2] besteht eine Datenflussanalyse aus drei Komponenten: einem Kontrollflussgraphen (KFG), einem Verband und Übertragungsfunktionen. Im Folgenden werden alle Komponenten näher erläutert.

KFG Gemäß Brabrand et al. [2] wird der KFG als Abstraktion des Eingabeprogramms verwendet, auf dem die Datenflussanalyse durchgeführt wird. Dieser liegt als gerichteter Graph vor, bei dem die Knoten die Programmanweisungen und die gerichteten Kanten den Programmfluss zwischen ihnen darstellen. Streng genommen repräsentieren die Knoten sogenannte Grundblöcke, welche eine oder mehrere Programmanweisungen zusammenfassen, die als Einheit sequenziell ausgeführt werden. Das bedeutet, dass die erste Anweisung des Grundblocks der einzige Eingang und die letzte Anweisung der einzige Ausgang ist. Darüber hinaus werden KFGs häufig in zwei Gruppen eingeteilt: intra- und interprozedurale KFGs. Koschke [7] zufolge liegt der Unterschied in der Granularität des Graphen: Ein intraprozeduraler KFG (siehe Abbildung 2.1) beschreibt den Kontrollfluss innerhalb einer Prozedur, während ein interprozeduraler KFG (siehe Abbildung 2.2) ein Programm in seiner Gesamtheit mit allen Prozeduraufrufen abbildet.

```
1 int main() {
2   if (a > 0) {
3     a = 1;
4     b = 2;
5   }
6   else {
7     c = f(c);
8   }
9 }
```

(a) Programm



(b) Kontrollflussgraph

ABBILDUNG 2.1: Eine main-Methode und der dazugehörige intraprozedurale KFG nach Koschke [7].

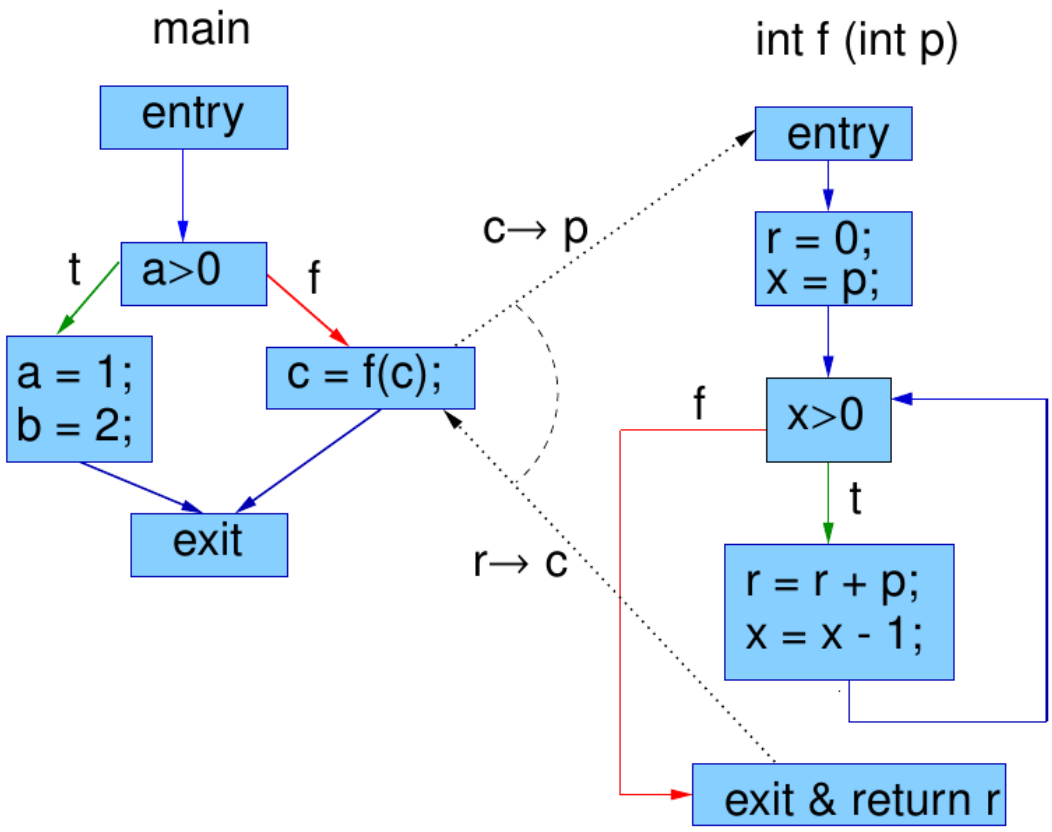


ABBILDUNG 2.2: Interprozeduraler KFG zweier Prozeduren nach Koschke [7].

Verband Nach Brabrand et al. [2] sind die Informationen, die von einer Datenflussanalyse berechnet werden, innerhalb eines Verbands $\mathcal{L} = (D, \sqsubseteq)$ organisiert, wobei D eine Menge von Elementen und $\sqsubseteq$ eine partielle Ordnung auf den Elementen ist. Es gibt zwei Elemente $\bot, \top \in D$, die eine besondere Bedeutung haben, um Sonderfälle abbilden zu können: $\bot$ heißt, dass der Verband zu einem bestimmten Betrachtungszeitpunkt noch nicht feststeht und $\top$ bedeutet, dass kein konkreter Wert ermittelt werden konnte. Eine wichtige Operation auf der Menge D ist der Least Upper Bound Operator $\sqcup$, welcher zwei Verbände und somit Informationen zu einem neuen Verband zusammenfasst. Bei intraprozeduralen Analysen ist dies immer dann notwendig, wenn eine Anweisung zwei oder mehr Vorgänger besitzt und die Ausgabeverbände vereinigt werden müssen.

Übertragungsfunktionen Brabrand et al. [2] schreiben, dass Übertragungsfunktionen die dritte wesentliche Komponente einer Datenflussanalyse bilden. Jeder Programmanweisung S wird eine Übertragungsfunktion $f_S = \mathcal{L} \to \mathcal{L}$

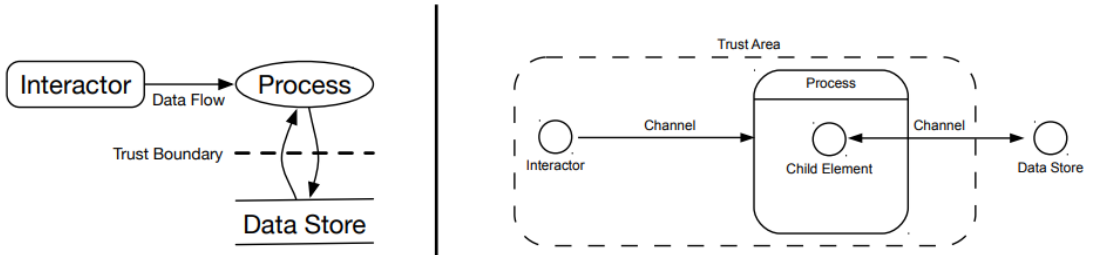
zugeordnet, die einem Eingabeverband einen Ausgabeverband zuordnet und die Ausführung der Anweisung S hinsichtlich des Verhaltens, was die Analyse untersucht, simuliert.

2.3 Datenflussdiagramm

Ein Datenflussdiagramm (DFD) veranschaulicht Datenflüsse in Systemen wie beispielsweise Software und kann als gerichteter Graph aufgefasst werden. Berger et al. [1] zufolge kann ein DFD aus fünf verschiedenen Elementen (siehe Abbildung 2.3a) zusammengesetzt sein: Prozesse, Datenspeicher, Interaktoren, Datenflüsse und Vertrauensgrenzen. Tabelle 2.1 beschreibt den Verwendungszweck aller Elemente oberflächlich.

Element	**Beschreibung**
Prozess	Stellt Komponenten dar, die Daten verarbeiten.
Datenspeicher	Stellt Komponenten dar, die Daten speichern.
Interaktor	Stellt externe Systeme oder Benutzer dar, die mit dem betrachteten System interagieren.
Datenfluss	Verbindet Prozesse, Datenspeicher und Interaktoren miteinander, um den Fluss bzw. Austausch von Daten abzubilden.
Vertrauensgrenze	Grenzt zwei verschiedene Vertrauensbereiche voneinander ab.

TABELLE 2.1: Übersicht aller DFD Elemente nach Berger et al. [1].



(a) Data Flow Diagram (b) Extended Data Flow Diagram

ABBILDUNG 2.3: DFD im Vergleich zu einem EDFD nach Berger et al. [1].

Erweiterte Datenflussdiagramme Da Berger et al. [1] mehrere Unzulänglichkeiten bei der Verwendung von DFDs für die Bedrohungsmodellierung festgestellt haben, benutzen sie EDFDs. Diese erweitern traditionelle DFDs und bestehen aus vier Konzepten (siehe Abbildung 2.3b): Elemente, Kanäle, Vertrauensbereiche und Daten. Das Besondere an den Konzepten ist, dass sie typisierbar sind und mit Hilfe von Annotationen beliebig viele Zusatzinformationen enthalten können. Einem EDFD liegt ein Schema zugrunde, welches eine Hierarchie von verfügbaren Typen sowie Annotationen spezifiziert und nach Bedarf erweitert werden kann. Tabelle 2.2 beschreibt den Verwendungszweck aller Konzepte und hebt die Gemeinsamkeiten bzw. Unterschiede zu einem traditionellen DFD hervor.

Konzept	**Beschreibung**
Element	Kombiniert Prozesse, Datenspeicher sowie Interaktoren aus traditionellen DFDs. Die Unterscheidung erfolgt über Typen. Ein Element kann untergeordnete Elemente besitzen, um Vater-Kind-Beziehungen und damit eine Hierarchie darzustellen.
Kanal	Stellt Datenflüsse zwischen Elementen dar, welche im Gegensatz zu einem traditionellen DFD auch bidirektional sein können.
Vertrauensbereich	Gruppiert mehrere Elemente zu einem Bereich.
Daten	Modellieren die Daten des betrachteten Systems, welche sowohl an Elemente als auch an Kanäle geheftet werden können.

TABELLE 2.2: Übersicht aller EDFD Konzepte nach Berger et al. [1].

Kapitel 3

Implementierung

Dieses Kapitel gibt Einblicke in die Implementierung. Zunächst werden die Ziele und das Verhalten der Analyse erläutert. Eine anschließende Vorstellung der verwendeten Technologien sowie Bibliotheken beantwortet unter anderem die Fragen, warum diese verwendet werden und welche Vorteile sie mit sich bringen. Bevor der Entwicklungsprozess im Detail beschrieben wird, werden die grundlegende Architektur der Software unter Verwendung der Unified Modeling Language (UML)[1] und wichtige Designentscheidungen aufgezeigt.

3.1 Ziele und Verhalten der Analyse

Bei der Analyse handelt es sich um eine statische Datenflussanalyse, die für Java-basierte Anwendungen Datenflüsse von geheimen bzw. sensitiven Methodenparametern ermittelt. Obwohl Android-Apps ebenfalls auf Java basieren, ist deren Unterstützung nicht vorgesehen. Die analysierte Anwendung und die ermittelten Datenflüsse sollen anschließend in ein EDFD übertragen werden, welches für weitere Untersuchungen wie z. B. die Extraktion von Datenflüssen genutzt werden kann. Die Akkuratesse einer statischen Programmanalyse wird zum einen durch das Fehlen von Laufzeitinformationen, wie z. B. die Anzahl von Schleifendurchläufen, und zum anderen durch Eigenschaften sowie das Verhalten der Analyse beeinflusst. Im Folgenden werden die wichtigsten Punkte erläutert.

Arrayinsensitivität Die Analyse ist nicht arraysensitiv, was bedeutet, dass sie ein Array als Ganzes betrachtet und nicht zwischen verschiedenen Indizes unterscheidet. Bei einer Zuweisung wird der Inhalt nicht überschrieben, sondern mit der rechten Seite vereinigt, selbst wenn es sich um einen zuvor

[1] `https://www.omg.org/spec/UML/`

zugewiesenen Index handelt. Für das folgende Beispiel würde die Analyse ermitteln, dass die Variable **result** die Werte A, B, und C enthalten könnte:

```
arr[0] = A;
arr[0] = B;
arr[1] = C;
result = arr[0];
```

LISTING 3.1: Code zur Demonstration von Arrayinsensitivität.

Objektinsensitivität Die Analyse ist nicht objektsensitiv. Findet ein Zugriff auf eine Instanzvariable statt, so spielt die Instanz keine Rolle und sie wird wie eine Klassenvariable behandelt. Auch hier wird bei einer Zuweisung vereinigt statt überschrieben. Für das folgende Beispiel würde die Analyse ermitteln, dass die Variable **result** die Werte A und B enthalten könnte:

```
instance1.Field = A;
instance2.Field = B;
result = instance3.Field;
```

LISTING 3.2: Code zur Demonstration von Objektinsensitivität.

Keine Non-Interference Focardi und Gorrieri [4] schreiben, dass die Hauptmotivation von Non-Interference darin besteht, den von Schadsoftware verursachten Schaden zu begrenzen oder zu vermeiden. Sie führen weiter aus, dass es die direkte Kontrolle über den gesamten Informationsfluss erfordert. In Bezug auf die Analyse bedeutet dies, dass sowohl direkte als auch indirekte Datenflüsse betrachtet und einbezogen werden müssten. Die Arbeit grenzt sich von diesem Ansatz ab und die Analyse betrachtet nur direkte Datenflüsse. Für das folgende Beispiel würde die Analyse also ermitteln, dass die Variable **result** nicht A und somit nichts enthält, obwohl A indirekt über die Länge zugewiesen wird, was für Passwörter sicherheitsrelevant sein kann:

```
result = new int[A.length];
```

LISTING 3.3: Beispiel eines indirekten Datenflusses.

3.2 Auswahl der Technologien

Zu Beginn des Entwicklungsprozesses stellt sich die Frage, welche Technologien oder Bibliotheken zur Umsetzung der Analyse verwendet werden sollen, da diese einen erheblichen Einfluss auf die Architektur ausüben. Dieser Abschnitt bietet eine Übersicht über alle verwendeten Technologien und erklärt relevante Details.

3.2.1 Soot

Soot[2] ist ein Framework, das statische Programmanalysen sowie Optimierungen von Java- und Android-Applikationen ermöglicht. Da der stack-basierte Bytecode von Java sich nicht für Optimierungen eignet, bietet Soot verschiedene Darstellungsformen an, die deutlich besser geeignet sind. Für Analysen bietet sich insbesondere Jimple an, da diese in 3-Adress-Code vorliegt und Berechnungen in einfache atomare Schritte zerlegt. Weil die Anweisungen sehr simpel aufgebaut sind und in der Regel lediglich einen Zieloperanden sowie zwei Quelloperanden besitzen, ist eine automatisierte Datenflussanalyse deutlich einfacher. [12]

Listing 3.4 zeigt eine einfache Methode in Java. Mit Hilfe von Soot wird diese in Jimple-Code (siehe Listing 3.5) überführt, der etwas umfangreicher ist und später Anweisung für Anweisung von der Analyse ausgewertet wird.

```
public int stepPoly(int x) {
    if(x < 0)        { System.out.println("foo"); return -1; }
    else if (x <= 5) return x * x;
    else             return x * 5 + 16;
}
```

LISTING 3.4: Die stepPoly-Methode in Java [12].

```
public int stepPoly(int) {
    Test r0;
    int i0, $i1, $i2, $i3;
    java.io.PrintStream $r1;

    r0 := @this;
    i0 := @parameter0;
```

[2] https://github.com/soot-oss/soot

```
    if i0 >= 0 goto label0;

    $r1 = java.lang.System.out;
    $r1.println("foo");
    return -1;

 label0:
    if i0 > 5 goto label1;

    $i1 = i0 * i0;
    return $i1;

 label1:
    $i2 = i0 * 5;
    $i3 = $i2 + 16;
    return $i3;
}
```

LISTING 3.5: Der Jimple-Code der stepPoly-Methode [12].

3.2.2 VASCO

VASCO[3] ist ein Framework, welches auf Soot basiert und interprozedurale Analysen unter Verwendung von wertesensitiven Kontexten ermöglicht. Im Gegensatz zur interprozeduralen Analyse von Soot, die mittels Graphenerreichbarkeit funktioniert, sind diese nicht auf bestimmte Probleme beschränkt. Abbildung 3.1 zeigt die wichtigsten Klassen von VASCO mit Hilfe eines UML-Klassendiagramms. [10]

3.2.3 ArchSec

Architectural Security Tool Suite (ArchSec)[4] ist ein von der Softwaretechnik AG der Universität Bremen entwickeltes Plugin, das die Eclipse Integrated Development Environment (IDE) um Werkzeuge erweitert, die bei einer architekturellen Risikoanalyse eingesetzt werden können. Für die Implementierung der Analyse spielt ArchSec keine Rolle, bietet aber die nötige Funktionalität an, um die Analyseergebnisse anschließend in einen EDFD überführen und diesen

[3] https://github.com/rohanpadhye/vasco
[4] https://archsec.informatik.uni-bremen.de/

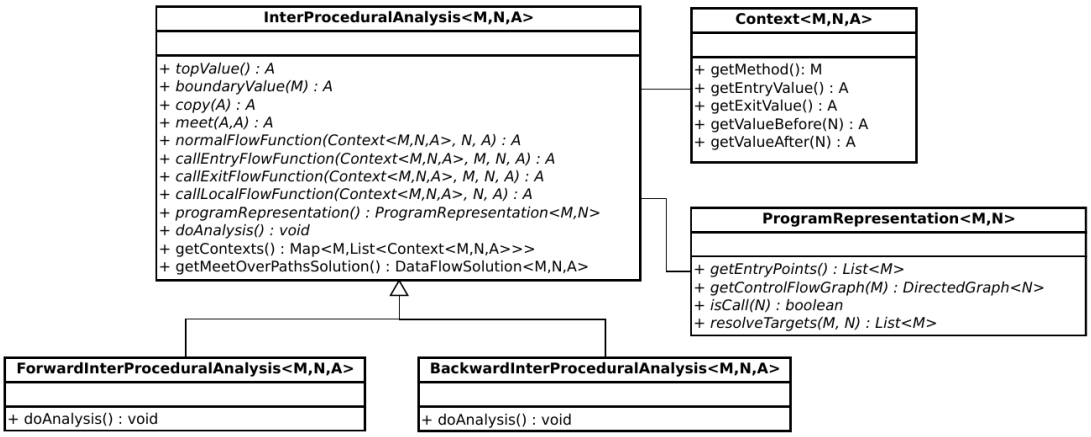


ABBILDUNG 3.1: Die Architektur von VASCO [10].

speichern zu können. Außerdem bietet es einen grafischen EDFD-Editor welcher auf den EcoreTools[5] von Eclipe basiert und hilfreich ist, um das Ergebnis zu überprüfen. Abbildung 3.2 zeigt einen Ausschnitt des Editors mit einem EDFD, welcher aus acht Elementen (Ein Feld, eine Klasse, drei Methoden und drei Parameter) und drei Kanälen besteht.

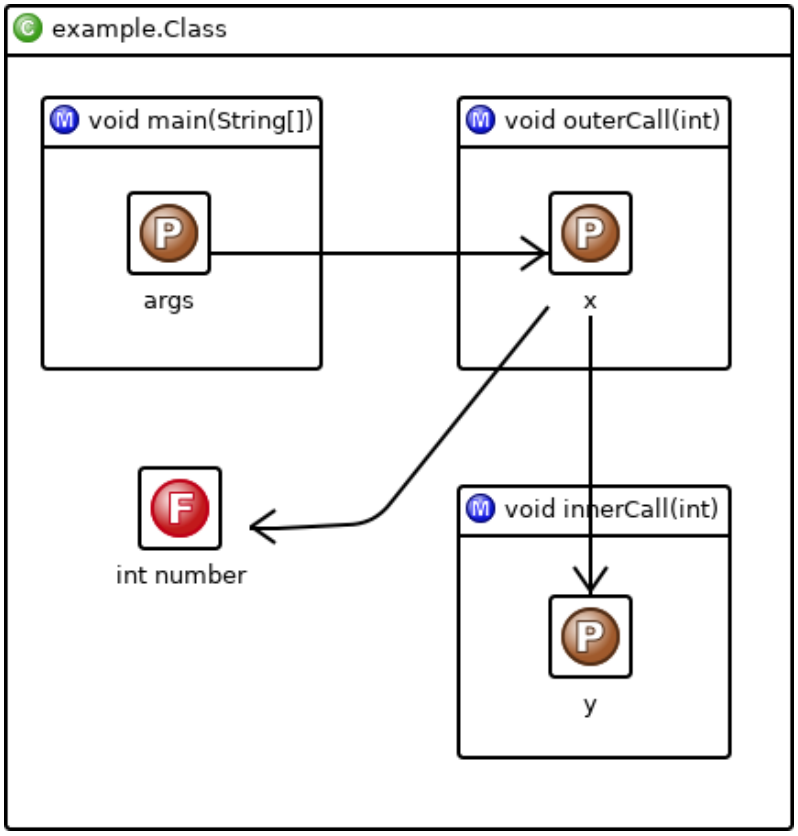


ABBILDUNG 3.2: Der EDFD-Editor von ArchSec.

[5] https://www.eclipse.org/ecoretools/

3.3 Architektur

Dieser Abschnitt stellt die Architektur der Datenflussanalyse vor. In Abbildung 3.3 ist ein UML-Klassendiagramm zu sehen, welches die Beziehungen zwischen bedeutenden Klassen veranschaulicht. Bei **AnalysisRunner** handelt es sich um die Hauptklasse, die den Einstiegspunkt enthält und bei dem Ausführen der Java Archive (JAR) Datei ausgeführt wird. Diese startet und steuert alle Vorgänge, welche mit Hilfe einer Konfiguration (siehe Abschnitt 3.4) anpassbar gestaltet sind, wie den Start des Analyseprozesses (siehe Abschnitt 3.5) oder der Generierung des EDFD (siehe Abschnitt 3.6). Ergebnisse, die nicht von VASCO berechnet werden, sind innerhalb der **AnalysisResults**-Klasse hinterlegt, nachdem alle Vorgänge beendet wurden.

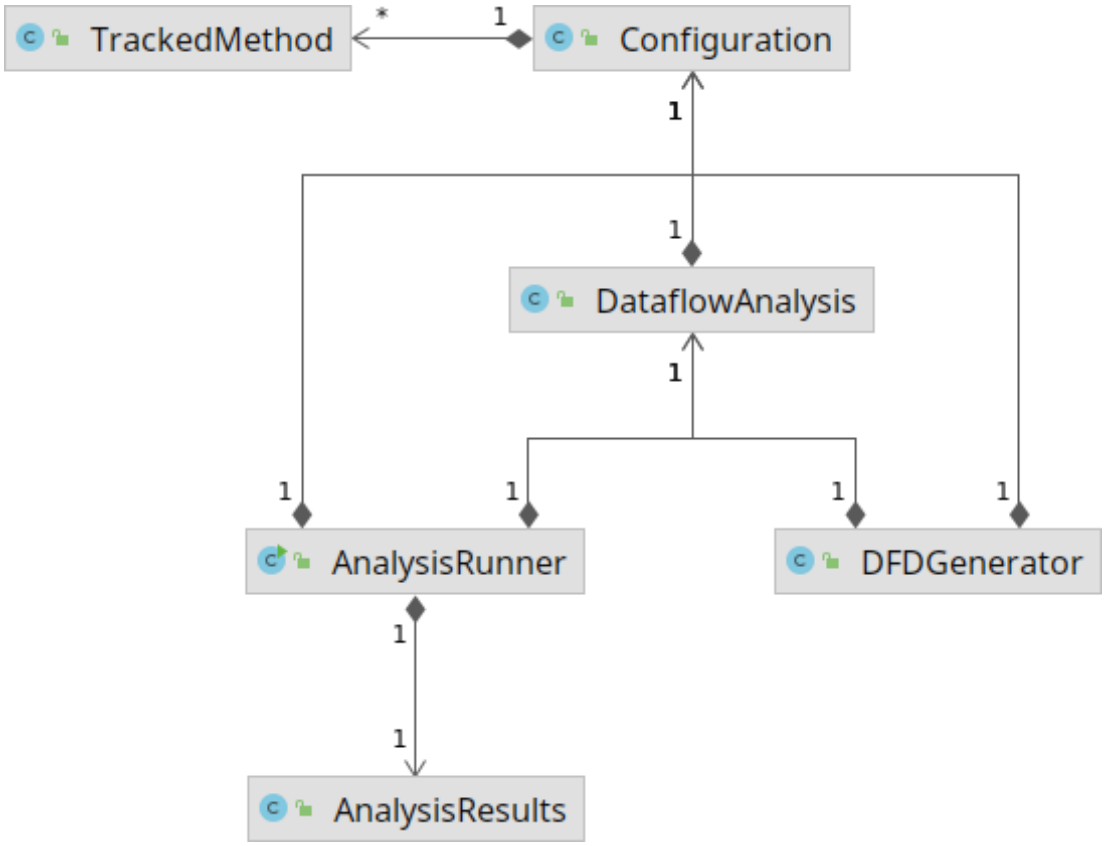


ABBILDUNG 3.3: Die Architektur der Analyse.

3.4 Konfiguration

Eine fundamentale Designentscheidung der Architektur ist, dass das Verhalten der Programmanalyse, die Überführung in ein EDFD und andere Einstellungen über Konfigurationsdateien beliebig angepasst werden können. An diesem Punkt stellt sich die Frage, welches Dateiformat für die Speicherung der Konfiguration sinnvoll ist. Die mit Abstand bekanntesten Formate sind JavaScript Object Notation (JSON) und Extensible Markup Language (XML). Weil JSON schneller verarbeitet werden kann und einen geringeren Speicherverbrauch als XML aufweist, liegt die Konfiguration in JSON vor [9][14].

Die **Configuration**-Klasse ist für das Einlesen, Verarbeiten und Speichern der Konfiguration verantwortlich. Eine Klasse zur Kapselung ist hilfreich, da das zugrunde liegende Dateiformat ausgetauscht werden kann, ohne dass sich die Schnittstelle zu den anderen Klassen ändert. Listing 3.6 zeigt eine mögliche Konfiguration mit den gebräuchlichsten Einstellungen. Lediglich das Setzen der Hauptklasse und den Klassenpfaden ist zwingend notwendig, damit die Analyse einen definierten Startpunkt besitzt und Klassen findet, die nicht in den Kernbibliotheken von Java enthalten sind. Um allerdings sinnvolle Ergebnisse zu erhalten, müssen sowohl Einstiegspunkte und unsichere Senken definiert werden als auch die EDFD-Generierung aktiviert werden.

```
{
  "classPaths": [ "./classes" ],
  "mainClass": "package.MainClass",
  "generateDFD": true,
  "entryPoints": [ {
      "method": "<C: void someEntry(int,int)>",
      "confidentialParameters": [ 0, 1 ]
    } ],
  "insecureSinks": [ {
      "method": "<C: void someSink(int)>",
      "confidentialParameters": [ 0 ]
    } ]
}
```

LISTING 3.6: JSON einer möglichen Konfiguration.

Tabelle 3.1 gibt einen Überblick über alle verfügbaren Einstellungen und ihre Auswirkungen auf alle Vorgänge der Software. Bei Einstiegspunkten und unsicheren Senken müssen die Methoden als Soot-Signaturen angegeben werden, dessen Syntax wie folgt aussieht:

```
<Klasse: Rückgabetyp Methodenname(Parametertyp 1,...)>
```

Die sensitiven Parameter werden anhand ihrer Position, beginnend bei 0, in der Methodensignatur angegeben. Positionen, die nicht vorhanden sind, werden ignoriert und führen zu keinem Fehler.

Name	**Typ**	**Beschreibung**
classPaths	String[]	Gibt alle Klassenpfade an, in denen Soot nach kompilierten Klassendateien sucht.
entryPoints	TrackedMethod[]	Gibt alle geheimen bzw. sensitiven Parameter an, deren Datenfluss verfolgt und bestimmt wird.
generateDFD	Boolean	Gibt an, ob der EDFD generiert wird.
insecureSinks	TrackedMethod[]	Gibt alle Parameter an, die als unsichere Senke gelten.
mainClass	String	Gibt die Hauptklasse an, die den Einstiegspunkt enthält. Dort startet die Analyse mit der Berechnung.
measureRuntime	Boolean	Gibt an, ob die Laufzeit der Initialisierung, der Analyse und der EDFD Generierung gemessen wird.
outputDFDName	String	Gibt den Namen des generierten EDFD an.
outputJimple	Boolean	Gibt an, ob der generierte Jimple-Code gespeichert werden soll.
outputPath	String	Gibt das Verzeichnis an, in welchem der EDFD gespeichert wird.

testMode	Boolean	Gibt an, ob der Testmodus aktiviert ist. In diesem werden alle Parameter aller Methoden als Einstiegspunkt betrachtet.
verbose	Boolean	Gibt an, ob Ein- und Ausgabeverbände aller Anweisungen auf der Konsole ausgegeben werden.

TABELLE 3.1: Übersicht aller Konfigurationseinstellungen.

3.5 Analyse

Die Datenflussanalyse ist innerhalb der **DataflowAnalysis**-Klasse implementiert. Diese erbt von der **ForwardInterProceduralAnalysis<M, N, A>**-Klasse, die sich im VASCO-Framework befindet und die nötige Funktionalität für eine interprozedurale Datenflussanalyse bietet. Außerdem hält sie einige abstrakte Methoden bereit, die überschrieben werden müssen, um die Analyse an den gewünschten Verwendungszweck anzupassen. Zu beachten ist, dass es sich um eine vorwärtsgerichtete Datenflussanalyse handelt, bei der die Analyse mit der ersten Anweisung des Einstiegspunktes beginnt. Padhye und Khedker [10] schreiben, dass die Analyse für Soot initiiert werden kann, indem die Typparameter **M** und **N** mit **SootMethod** und **Unit** ersetzt werden.

3.5.1 Verband

Der Typparameter **A** legt den Datentyp des Verbands fest, der von der Datenflussanalyse verwendet wird. In der entwickelten Analyse fungiert die **Lattice**-Klasse als Verband und verhält sich nach außen ähnlich wie eine Zuordnungstabelle von **ValueWrapper**-Instanzen auf **Result**-Instanzen. Theoretisch handelt es sich bei den Schlüsseln lediglich um **Value**-Instanzen aus Soot, die unter anderem lokale Variablen, Instanz- oder Klassenfelder darstellen. Da sie allerdings über keinen geeigneten Hashwert für die Verwendung in einer Zuordnungstabelle verfügen, ist **ValueWrapper** nötig, um einen Hashwert bereitzustellen. Die **Result**-Klasse stellt eine Menge von geheimen bzw. sensitiven Parametern dar. Eine Besonderheit der **Lattice**-Klasse gegenüber einer

gewöhnlichen Zuordnungstabelle ist, dass sie Aliase erlaubt. Das bedeutet, dass ein Schlüssel auf einen anderen Schlüssel verweisen kann, sodass mehrere Schlüssel die gleiche **Result**-Instanz zugeordnet bekommen. Abbildung 3.4 veranschaulicht den Aufbau des Verbandes in einem UML-Klassendiagramm.

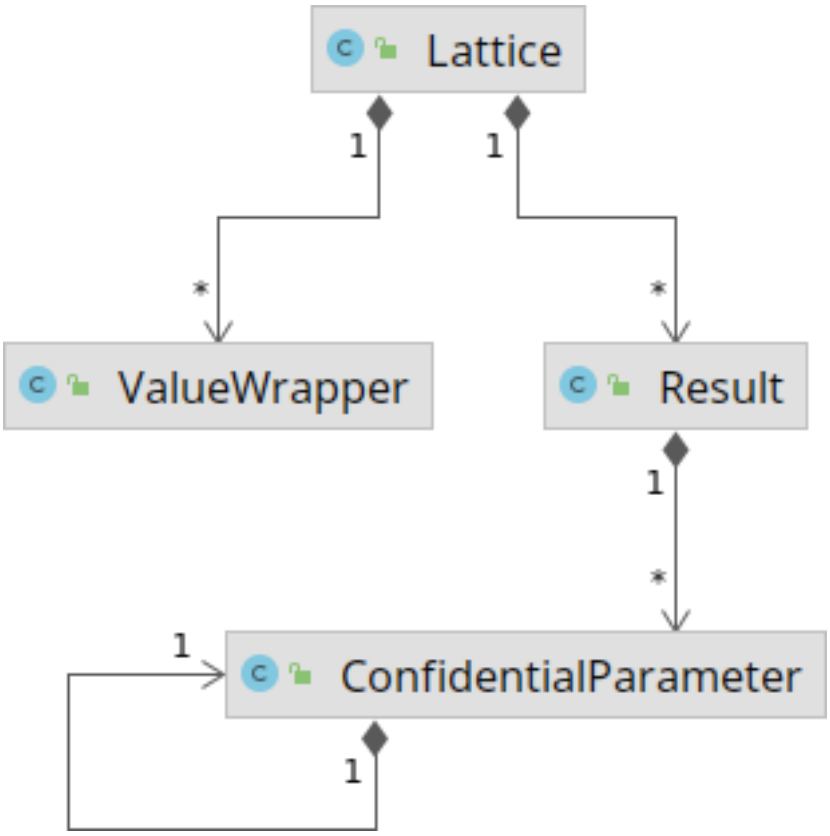


ABBILDUNG 3.4: Der Aufbau des Verbands.

Die **ConfidentialParameter**-Klasse enthält einen Verweis auf sich selbst, um den Parameter zu speichern, über den der Parameter als Letztes übertragen wurde. Dies ist notwendig, um den Datenfluss nach der Analyse präzise rekonstruieren zu können.

3.5.2 Intraprozeduraler Datenfluss

Das VASCO-Framework bietet die abstrakten Methoden **normalFlowFunction** und **callLocalFlowFunction** an, die von einer Analyse überschrieben werden, um den intraprozeduralen Datenfluss zu verarbeiten. Damit stellen sie die Übertragungsfunktionen aller Anweisungen innerhalb einer Prozedur dar.

Beide erhalten als Parameter den aktuellen Werte-Kontext, die zu analysierende Anweisung sowie den Eingabeverband (IN-Set) und geben den resultierenden Verband (OUT-Set) zurück.

Findet innerhalb einer Anweisung kein Methodenaufruf statt, so wird von VASCO die **normalFlowFunction**-Methode (siehe Pseudocode 1) aufgerufen. Zu Beginn wird der Eingabeverband kopiert und überprüft, was für eine Art von Anweisung momentan analysiert wird. Für diese Datenflussanalyse spielen drei Arten eine Rolle: Identitäts-, Zuweisungs- und Rückgabeanweisungen.

Pseudocode 1 : normalFlow Übertragungsfunktion

```
   Input : context
   Input : unit
   Input : inLattice
1  outLattice ← COPY(inLattice)
2  if unit is IdentityStatement then
3      if configuration.isSensitive(unit.parameter) then
4          outLattice.add(unit.parameter)
5  else if unit is AssignStatement then
6      assignValue(outLattice, unit.leftHand, unit.rightHand)
7  else if unit is ReturnStatement then
8      unionValue(outLattice, RETURN_VALUES, unit.returnValue)
9  return outLattice
```

Bei einer Identitätsanweisung wird ein Parameter der Methode der entsprechenden lokalen Variable zugeordnet. Da an dieser Stelle alle relevanten Informationen wie Methodenname, Parametername sowie Parameterindex zur Verfügung stehen, wird mit Hilfe der Konfiguration geprüft, ob dieser Parameter als geheim bzw. sensitiv markiert ist und entsprechend in den Ausgabeverband aufgenommen. Auch Rückgabeanweisungen sind wesentlich, da sämtliche geheimen bzw. sensitiven Parameter erfasst werden müssen, die von einer Methode zurückgegeben werden können. **RETURN_VALUES** dient als Behälter für alle Rückgabewerte einer Methode und hilft bei dem interprozeduralen Ablauf. Weil eine Methode mehr als eine Rückgabeanweisung besitzen kann, werden die Rückgabewerte vereinigt. Die wohl wichtigste Anweisungsart ist die Zuweisungsanweisung, da diese im intraprozeduralen Bereich in der Regel zu einer Änderung des Verbands führt. Weil eine Zuweisung in

Jimple in vielen verschiedenen Facetten vorkommen kann, wird diese separat in der **assignValue**-Methode (siehe Pseudocode 2) verarbeitet. So kann es beispielsweise die Zuweisung eines einzelnen Wertes sein, die Zuweisung des Ergebnisses einer binären Operation, die Instanziierung einer Klasse oder das Erstellen eines Arrays.

Pseudocode 2 : assignValue

```
   Input : lattice
   Input : key
   Input : value
 1 if value is NewExpr OR NewArrayExpr OR LengthExpr then
 2     return
 3 if value is BinaryOperatorExpr then
 4     foreach operand ∈ value.operands do
 5         unionValue(lattice, key, operand)
 6     return
 7 if key is ArrayReference then
 8     unionValue(lattice, key.base, value)
 9 else if key is Local OR InstanceField OR StaticField then
10     if value is ArrayReference then
11         if value.base in lattice then
12             lattice.add(key, value.base)
13     else if value is InstanceField OR StaticField then
14         if value.type is Array then
15             lattice.addAlias(key, value)
16         else if value in lattice then
17             lattice.add(key, lattice.get(value))
18     else if value in lattice then
19         lattice.add(key, lattice.get(value))
```

Zu aller erst sei erwähnt, dass die Instanziierung von Klassen ignoriert wird, da sie im Verfolgen von geheimen bzw. sensitiven Parametern nichts beitragen. Dasselbe gilt für das Erstellen von Arrays oder das Abfragen der Länge von Arrays. Theoretisch könnte es sicherheitskritisch sein, wenn ein sensitiver Parameter als Länge eines Arrays verwendet wird. Solche indirekten Annahmen verfolgt man im Non-Interference Ansatz. Dieser ist allerdings nicht Gegenstand dieser Arbeit.

Wenn die rechte Seite der Zuweisung eine binäre Operation ist, so werden die geheimen bzw. sensitiven Parameter beider Operanden vereinigt und dem Verband hinzugefügt. Das Verhalten ist bei arithmetischen, bitweisen und logischen Operationen identisch. Handelt es sich um eine Zuweisung an ein Array, so wird der Index aufgrund der Arrayinsensitivität ignoriert und die sensitiven Parameter mit dem Array vereinigt. In dem Fall, dass die linke Seite auf eine anderweitige lokale, statische oder Instanzvariable verweist, so hängt das weitere Vorgehen von der rechten Seite ab. Die linke Seite der Zuweisung wird im Verband grundsätzlich überschrieben, außer wenn die rechte Seite ein Array und Feld zugleich ist. In diesem Fall wird die linke Seite als Alias für die rechte Seite verwendet, sodass das referenzierte Array aktualisiert wird, sobald ein Schreibzugriff auf den Alias stattfindet.

3.5.3 Interprozeduraler Datenfluss

Das VASCO-Framework bietet die abstrakten Methoden **callEntryFlowFunction** und **callExitFlowFunction** an, die von einer Analyse überschrieben werden, um den interprozeduralen Datenfluss zu verarbeiten. Sie repräsentieren also die Übertragungsfunktionen für Methodenaufrufe. Beide erhalten als Parameter den aktuellen Werte-Kontext, die aufgerufene Methode, die zu analysierende Anweisung sowie einen Eingabeverband (IN-Set) und geben den resultierenden Verband (OUT-Set) zurück.

Aufruf Findet ein Methodenaufruf statt, so wird der Eingabeverband der ersten Anweisung von der aufgerufenen Methode in der **callEntryFlowFunction**-Methode berechnet. Pseudocode 3 zeigt den Ablauf der Methode.

Zu Beginn wird ein leerer Ausgabeverband erzeugt. Der nächste Schritt ist die Übergabe der Argumente an die aufzurufende Methode. Dazu wird über alle Argumente iteriert und geprüft, ob das Argument im Verband vorhanden ist und geheime bzw. sensitive Parameter enthält. Falls dies zutrifft, so werden alle sensitiven Parameter zum Ausgabeverband kopiert. Allerdings wird dabei **over** auf den Parameter gesetzt, damit später der exakte Datenfluss rekonstruiert werden kann. Zum Schluss werden alle Instanz- und Klassenfelder in den Ausgabeverband kopiert. Das ist erlaubt, da sich die Instanzfelder aufgrund der Objektinsensitivität exakt wie Klassenfelder verhalten.

Pseudocode 3 : callEntryFlow Übertragungsfunktion

```
   Input : context
   Input : callee
   Input : unit
   Input : inSet
 1 outSet ← EMPTY()
 2 invoke ← unit.invokeExpression
 3 foreach argument ∈ invoke.arguments do
 4     if argument in inSet then
 5         result ← inSet.get(argument)
 6         result.over ← callee.localParameter
 7         outSet.add(callee.localParameter, result)
 8 foreach key ∈ inSet.keys do
 9     if key is StaticField OR key is InstanceField then
10         outSet.add(key, inSet.get(key))
11 return outSet
```

Rücksprung Nach der Analyse der aufgerufenen Methode wird der Ausgabeverband des Aufrufs in der **callExitFlowFunction**-Methode berechnet und von Daten, die für den Aufrufer keine Relevanz besitzen, bereinigt. Pseudocode 4 zeigt den Ablauf der Methode.

Pseudocode 4 : callExitFlow Übertragungsfunktion

```
   Input : context
   Input : callee
   Input : unit
   Input : exitSet
 1 outSet ← EMPTY()
 2 invoke ← unit.invokeExpression
 3 if unit is Assignement AND RETURN_VALUES in exitSet then
 4     outSet.add(unit.getLeft, exitSet.get(RETURN_VALUES))
 5 foreach key ∈ exitSet.keys do
 6     if key is StaticField OR key is InstanceField then
 7         outSet.add(key, exitSet.get(key))
 8 return outSet
```

Zu Beginn wird ein leerer Ausgabeverband erzeugt. Handelt es sich bei der Anweisung um eine Zuweisung, so wird dem Verband die linke Seite hinzugefügt mit dem Ergebnis des künstlichen **RETURN_VALUES** Behälters. Danach werden alle Instanz- und Klassenfelder heraus kopiert. Direkt im Anschluss wird die **callLocalFlowFunction**-Methode (siehe Pseudocode 5) aufgerufen, um den Verband des intraprozeduralen Ablaufs des Aufrufers auf die Rückgabe vorzubereiten. Diese löscht lediglich alle Instanz- und Klassenfelder sowie im Falle einer Zuweisung die linke Seite, da diese von der aufgerufenen Methode überschrieben wird.

Pseudocode 5 : callLocalFlow Übertragungsfunktion

```
   Input : context
   Input : unit
   Input : inLattice
1  outLattice ← COPY(inLattice)
2  if unit is Assignement then
3      outLattice.remove(unit.getLeft)
4  foreach key ∈ inLattice.keys do
5      if key is StaticField OR InstanceField then
6          outLattice.remove(key)
7  return outLattice
```

Zum Abschluss werden die berechneten Verbände der beiden Methoden vereinigt und bilden den Ausgabeverband der intraprozeduralen Aufrufanweisung. Mit der Vereinigung ist der Rücksprung abgeschlossen.

3.6 EDFD Generierung

Für die Überführung der Analyseergebnisse in ein EDFD ist die **DFDGenerator**-Klasse zuständig. Diese lädt als ersten Schritt einen leeren EDFD, welcher bereits ein vordefiniertes Schema mit sinnvollen Typen für Klassen, Methoden, Parametern, Feldern sowie Datenflüssen vorgibt.

3.6.1 Schema-Erweiterung

Um die Analyseergebnisse adäquat innerhalb des Diagramms widerzuspiegeln sind allerdings weitere Typen, Attribute und Datentypen (siehe Tabelle 3.2) notwendig, die vor der Generierung dem Schema hinzugefügt werden

müssen. Zu beachten ist, dass die tatsächlichen Namen mit dem Präfix ≪DataFlowAnalysis.≫ beginnen. Für das Erstellen von neuen Schema-Elementen stellt ArchSec die Fabrikklasse **DFDFactory**-Klasse zur Verfügung. Diese neuen Instanzen müssen danach mit den angebotenen Setter-Methoden konfiguriert und letzlich dem Schema hinzugefügt werden.

Name	**Typ**	**Beschreibung**
IsEntryPoint	ElementAttribute	Markiert Methoden, die mindestens einen geheimen bzw. sensitiven Parameter besitzen.
IsConfidential	ElementAttribute	Markiert Parameter, die geheim bzw. sensitiv sind.
IsSink	ElementAttribute	Markiert sowohl Methoden, die mindestens eine unsichere Senke enthalten, als auch Parameter, die eine unsichere Senke sind.
DataType	DataType	Datentyp, um geheime bzw. sensitive Parameter zu speichern.
IsConfidential	DataAttribute	Attribut, das für ein Datum angibt, ob es geheim bzw. sensitiv ist. Da nur solche gespeichert werden, ist der Wert immer wahr und dient bei späteren Anfragen lediglich als Hilfsmittel.

TABELLE 3.2: Übersicht aller Schema-Erweiterungen.

3.6.2 Algorithmus

Die Generierung des EDFDs beginnt mit der Main-Methode der Hauptklasse, die in der Konfiguration angegeben wurde. Ausgehend von dieser werden alle anderen Methoden des Programms direkt oder indirekt verarbeitet. Pseudocode 6 veranschaulicht die Schritte, die bei der Verarbeitung einer Methode vorgenommen werden.

Um die Terminierung des Algorithmus bei rekursiven oder zyklischen Methodenaufrufen zu gewährleisten wird zu Beginn überprüft, ob die Methode bereits zuvor verarbeitet wurde. Ist dies nicht der Fall so wird die Methode zusammen mit ihren Parametern zu dem EDFD hinzugefügt. VASCO legt für

Pseudocode 6 : processMethod

```
   Input : diagram
   Input : method
 1 if alreadyProcessed then
 2     return
 3 addMethod(diagram, method)
 4 foreach context ∈ methodContexts do
 5     foreach unit ∈ methodUnits do
 6         if unit is Assignment then
 7             processAssignment(diagram, context, unit)
 8         else if unit is Invocation then
 9             processInvocation(diagram, context, unit)
```

jeden Aufruf derselben Methode, bei dem sich der Eingabeverband unterscheidet, einen neuen Werte-Kontext an. Außerdem liegt für jede Anweisung Ein- und Ausgabeverband vor. Aus diesen Gründen reicht es für die vollständige Generierung des EDFDs aus, wenn alle Anweisungen der Methode für alle Werte-Kontexte durchlaufen werden und Zuweisungs- (siehe Pseudocode 8) sowie Aufrufanweisungen (siehe Pseudocode 7) verarbeitet werden.

Pseudocode 7 : processInvocation

```
   Input : diagram
   Input : context
   Input : unit
 1 callee ← unit.getCallee
 2 processMethod(diagram, callee)
 3 inLattice ← context.getInLattice(unit)
 4 for i ← 0 to callee.argumentCount do
 5     argument ← unit.getInvokeArgument(i)
 6     if inLattice.contains(argument) then
 7         foreach value ∈ inLattice.get(argument) do
 8             src ← addParameter(diagram, value.over.method,
                 value.over.parameterName)
 9             dst ← addParameter(diagram, callee,
                 value.over.parameterName)
10             addDataFlow(diagram, src, dst, value)
```

Wenn eine Aufrufanweisung angetroffen wird, so wird diese innerhalb der **processInvocation**-Methode verarbeitet. Zunächst verarbeitet diese mit Hilfe von **processMethod** die aufgerufene Methode und verbindet anschließend übergebene Argumente mit den Parametern der aufgerufenen Methode. Dazu wird anhand des Eingabeverbands der Aufrufanweisung überprüft, ob das Argument geheime bzw. sensitive Parameter enthält. Ist dies der Fall, so wird für jeden sensitiven Parameter die Herkunft (d.h. ein Parameter oder Feld) mit dem Parameter verbunden.

Pseudocode 8 : processAssignment

```
   Input : diagram
   Input : context
   Input : unit
 1 if righthand side is Invocation then
 2     processInvocation(diagram, context, unit)
 3 if lefthand side is StaticField OR InstanceField then
 4     outLattice ← context.getOutLattice(unit)
 5     if outLattice.contains(lefthand) then
 6         foreach value ∈ outLattice.get(lefthand) do
 7             src ← addParameter(value.over.method,
                 value.over.parameterName)
 8             dst ← addStaticOrInstanceField(diagram, lefthand)
 9             addDataFlow(diagram, src, dst, value)
```

Wenn eine Zuweisungsanweisung angetroffen wird, so wird diese mit der **processAssignment**-Methode verarbeitet. Diese überprüft zunächst, ob die rechte Seite der Zuweisung ein Methodenaufruf ist und verarbeitet diesen gegebenfalls. Weil nur Zuweisungen an Klassen- und Instanzfelder eine Rolle spielen, wird geprüft, ob die linke Seite der Zuweisung solche sind. Handelt es sich um ein Feld, so wird im Ausgabeverband überprüft, ob dieses geheime bzw. sensitive Parameter enthält. Die Herkunft jedes Parameters wird mit dem Feld verbunden. Nachdem der EDFD vollständig konstruiert ist, werden alle Elemente dahingehend überprüft, ob sie Relevanz besitzen und werden andernfalls entfernt. Dabei gelten Elemente, die weder Daten tragen noch Kinder besitzen als nicht relevant.

Kapitel 4

Evaluation

Dieses Kapitel befasst sich mit der Auswertung der Analyse. Zunächst wird die Vorgehensweise beim automatisierten Testen erläutert. Dann wird die Analyse anhand von drei konstruierten Testanwendungen ausgewertet, bevor abschließend ein Vergleich mit LAPSE+, einem ähnlichen Analysewerkzeug, vorgenommen wird.

4.1 Automatisierte Tests

Bevor die Analyse mit Hilfe größerer Applikationen evaluiert wird, bietet es sich an Unit- und Integrationstests zu schreiben, die Eigenschaften der Analyse überprüfen. Laut Khorikov [6] handelt es sich bei einem Unittest um einen automatisierten Vorgang, der eine kleine Einheit schnell und isoliert testet. Nach ihm gilt ein Test, der mindestens eine dieser Eigenschaften nicht erfüllt, als Integrationstest. Da die automatisierten Tests die Analyse ohne Mocking testen sollen und Soot bereits mehrere Sekunden für die Initialisierung der Analyse benötigt, sind schnelle Tests ausgeschlossen. Aus diesen Gründen handelt es sich um Integrationstests, die aus einer Ansammlung von ähnlichen Unittests aufgebaut sind.

Aufbau der Tests Die Tests basieren auf dem JUnit-Framework und folgen dem Arrange-Act-Assert Muster. Um zu gewährleisten, dass diese isoliert sind, wird vor jedem Test eine neue **AnalysisRunner**-Instanz erzeugt und so konfiguriert, dass der Testmodus aktiviert ist. Listing 4.1 verdeutlicht die Struktur anhand eines Beispieltests. Nachdem zu Beginn eines Tests die Hauptklasse spezifiziert wurde, wird die Analyse gestartet. Anschließend werden in der Assert-Sektion die Analyseergebnisse hinsichtlich des gewünschten Verhaltens überprüft. Dazu werden Ein- und Ausgabeverbände von Methoden oder

einzelnen Anweisungen über einen VASCO Werte-Kontext mit Hilfe von **getEntryValue()**, **getExitValue()**, **getValueBefore(unit)** oder **getValueAfter(unit)** abgefragt.

```
@Test
public void exampleTest() {
  // Arrange
  runner
    .getConfiguration()
    .setMainClass("package.MainClass");

  // Act
  runner.Run();

  // Assert
  DataflowAnalysis analysis = runner.getAnalysis();

  for (SootMethod method : analysis.getMethods()) {
    if (method.getName().equals("main")) {
      Context context = analysis.getContexts(method).get(0);

      // Assert results here ...
    }
  }
}
```

LISTING 4.1: Die Struktur eines Tests.

Die automatisierten Tests dienen in erster Linie dazu, die Analyseergebnisse bei typischen Java-Konstrukten wie Verzweigungen, Schleifen, Arrays sowie Rekursionen zu testen und zu prüfen, ob der Datenfluss bei der Verwendung von Instanz- und Klassenfeldern korrekt abgebildet wird. Hierbei wird besonders darauf geachtet, dass die festgelegten Analyseeigenschaften, wie Array- und Objektinsensitivität, eingehalten werden.

4.2 Testfälle

In diesem Abschnitt wird die Datenflussanalyse anhand von drei konstruierten Anwendungen evaluiert. Als Testsystem wird ein Notebook mit Manjaro Linux (64 Bit) verwendet, welches mit einem Intel Core i7-6500U (2,50 GHz) sowie 8 GB Arbeitsspeicher ausgestattet ist. Wegen des hohen Bedarfs

an Arbeitsspeicher der Analyse, wurde ergänzend eine Auslagerungsdatei der Größe 60 GB angelegt, damit größere Anwendungen mit vielen und umfangreichen Klassen analysiert werden können.

4.2.1 Vorgehen

Für eine Testanwendung existiert jeweils eine Konfigurationsdatei, die den Klassenpfad auf die Anwendungsklassen, die Hauptklasse der Anwendung und Einstiegspunkte sowie unsichere Senken definiert. Abgesehen davon aktiviert sie die EDFD-Generierung und die Messung der Laufzeit. Unter Verwendung der JAR wird die Analyse mittels der folgenden Befehle auf der Kommandozeile ausgeführt:

```
java -Xmx60g -jar analysis.jar --configuration <file>
jconsole <analysis_process_id>
```

Das Argument **-Xmx60g** erlaubt der Analyse einen Arbeitsspeicherverbrauch von bis zu 60 GB und ist nötig, damit sie nicht frühzeitig vom Betriebssystem terminiert wird, falls der Arbeitsspeicherbedarf zunimmt. JConsole ist ein Überwachungswerkzeug, das zur Ermittlung des höchsten Arbeitsspeicherverbrauchs der Analyse verwendet wird, da es genauere Informationen als der Task-Manager des Betriebssystems liefert. Nach erfolgreichem Abschluss der Analyse werden die Laufzeiten der einzelnen Phasen (Initialisierung, Analyse, EDFD Generierung) sowie der höchste Arbeitsspeicherverbrauch notiert. Damit die Werte der verschiedenen Anwendungen in Relation gesetzt werden können, werden die Source Lines of Code (SLOC) des Java- sowie Jimple-Codes herangezogen. Diese zählen weder Leer- noch Kommentarzeilen mit. Anschließend wird der generierte EDFD in die Eclipse IDE exportiert und im EDFD-Editor von ArchSec betrachtet, um manuell nach Fehlern zu suchen. Danach werden mit ArchSec und einer Sicherheitswissensdatenbank, die eine Regel für die Extraktion von unsicheren Datenflüssen enthält, Anfragen auf dem generierten EDFD ausgeführt.

Listing 4.2 zeigt den gesamten Inhalt der Sicherheitswissensdatenbank. Der relevante und wichtigste Teil beginnt mit **matchPatterns**, welcher definiert wie die unsicheren Datenflüsse extrahiert werden sollen. Die Definition wird mit einem Musterabgleich in der deklarativen Abfragesprache Cypher angegeben,

welche auf Graphen arbeitet und an die Structured Query Language (SQL) angelehnt ist [5]. Die Ausgabe ist, ähnlich wie bei SQL, eine Tabelle, in der eine Reihe genau einem unsicheren Datenfluss entspricht.

Die Anfrage basiert auf einer **MATCH**-Klausel, die ein Muster erwartet, mit dessen Hilfe der EDFD nach unsicheren Datenflüssen durchsucht werden soll. Zunächst muss eine Quelle (input) ermittelt werden, welche ein Element vom Typ eines Java-Parameters ist und das Attribut **IsConfidential** erfüllen muss. Dieses wird in einem Eingangselement (entry) gesucht, das eine Java-Methode ist und das Attribut **IsEntryPoint** erfüllt. Die Senke wird genau auf die gleiche Weise ermittelt, nur dass die Methode (exit) und der darin enthaltene Parameter (target) das Attribut **IsSink** erfüllen müssen. Nachdem die beiden Parameterelemente bestimmt sind, wird ein unsicherer Datenfluss zwischen diesen mit `-[: "Channel"* 1.. {type : "Data Flow"}]->` gesucht. Dieser Teil definiert, dass ein solcher Datenfluss eine Folge von Kanälen des Typs **Data Flow** ist und aus mindestens einem Kanal bestehen muss. Die anschließend folgende **RETURN**-Klausel gibt das Ergebnis zurück und spezifiziert dabei die Namen der Tabellenspalten.

```
Repository DataFlow {
  CypherRule InsecureDataFlow {
    id = "InsecureDataFlow"
    category (InformationDisclosure)
    severity = High
    exploitability = Likely
    description = "..."

    matchPatterns (
      MATCH p = (entry : "Element" {type : "Software.Source Code.
          Java.Method", "DataFlowAnalysis.IsEntryPoint" : true})
          -[:contains *1]->
             (input : "Element" {type : "Software.Source Code.Java
                 .Parameter", "DataFlowAnalysis.IsConfidential" :
                 true}) -[: "Channel"  * 1.. {type : "Data Flow"
                 }]->
             (target : "Element" {type : "Software.Source Code.
                 Java.Parameter", "DataFlowAnalysis.IsSink" : true
                 }) <-[:contains *1]-
             (exit : "Element" {type : "Software.Source Code.Java.
                 Method", "DataFlowAnalysis.IsSink" : true})
```

```
      RETURN entry AS host, entry AS entry, exit AS exit, input
         AS input, target AS target, p AS path
    )

  }
  RulePack All {
    uid = 1;
    InsecureDataFlow
  }
}
```

LISTING 4.2: Sicherheitswissensdatenbank mit einer Regel zur Extraktion von unsicheren Datenflüssen.

Die ermittelten unsicheren Datenflüsse werden mit den erwarteten Ergebnissen verglichen und in einer Konfusionsmatrix festgehalten. Im Folgenden bedeutet positiv, dass ein Datenfluss als unsicher angesehen wird. Auf die Angabe von wahr-negativen Datenflüssen wird in der Matrix verzichtet und als 0 angenommen, da nicht existente Datenflüsse unpraktikabel sind und bereits kleine Anwendungen eine sehr hohe Anzahl aufweisen. Auf Basis der Matrix werden Kennwerte wie Accuracy, Precision und Recall berechnet, um Aussagen über die Treffsicherheit der Analyse zu machen. Am Ende des Kapitels findet eine abschließende Auswertung statt, in der alle Ergebnisse aller Testanwendungen zusammen betrachtet werden.

Zu beachten ist, dass es sich bei den in den Testanwendungen ausgewählten Einstiegspunkten sowie Senken nicht unbedingt um in der Praxis unsichere Parameter bzw. Methoden handelt. Dies verfälscht jedoch keineswegs die hier vorgestellten Ergebnisse, da es sich letztendlich nur um eine Markierung in der Konfiguration handelt und die Semantik der Methode keine Bedeutung für die Analyse besitzt.

4.2.2 Testanwendung 1

Die erste Testanwendung besteht aus drei Klassen und verwendet nur wenige Methoden aus den Kernbibliotheken von Java, in denen intern wenig bis keine weiteren Aufrufe stattfinden. Darüber hinaus besitzt sie weder zyklische Datenflüsse noch Datenflüsse, die über Instanzfelder, Klassenfelder oder Arrays laufen. Tabelle 4.1 zeigt Kennwerte über die Ausmaße der Anwendung.

Name	Wert
Klassenanzahl	3
Methodenanzahl	11
SLOC (Java)	70
SLOC (Jimple)	157

TABELLE 4.1: Komplexität der Anwendung.

Konfiguration Tabelle 4.2 zeigt die eingestellten Einstiegspunkte und unsicheren Senken. Basierend auf dieser Konfiguration entstehen insgesamt fünf unsichere Datenflüsse in der Anwendung.

	Methode	Parameter
Einstiegspunkte	Program.main	0
	Program.sendInformation	0
Unsichere Senken	java.io.PrintStream.println	0
	UnsafeWorker.send	0

TABELLE 4.2: Konfiguration der Anwendung.

Ergebnisse Zunächst zeigt Tabelle 4.3 Laufzeitdaten und den Arbeitsspeicherverbrauch. Dass die Initialisierung von Soot den größten Teil der Laufzeit in Anspruch nimmt ist bemerkenswert, aber nicht verwunderlich, da in dieser Phase viele Vorgänge wie beispielsweise die Konvertierung des Java-Bytecodes in Jimple-Code stattfinden.

Name	Wert
Laufzeit (Initialisierung)	60,402s
Laufzeit (Analyse)	0,049s
Laufzeit (EDFD Generierung)	0,358s
Arbeitsspeicherverbrauch	1,561 GB

TABELLE 4.3: Statistiken der Anwendung.

In Abbildung 4.1 ist der generierte EDFD abgebildet, welcher bereits bei einer einfachen Anwendung mit geringen SLOC und wenigen externen Methoden umfangreich ausfällt. Positiv ist, dass der Datenfluss innerhalb der Java-Kernbibliothek bzw. des Fremdcodes korrekt bestimmt wurde. Dieser Aspekt

ist wichtig, wenn aus dem Fremdcode heraus Methoden der eigentlichen Anwendung aufgerufen werden, was z. B. häufig mit der **hashCode**-Methode bei Zuordnungstabellen passiert.

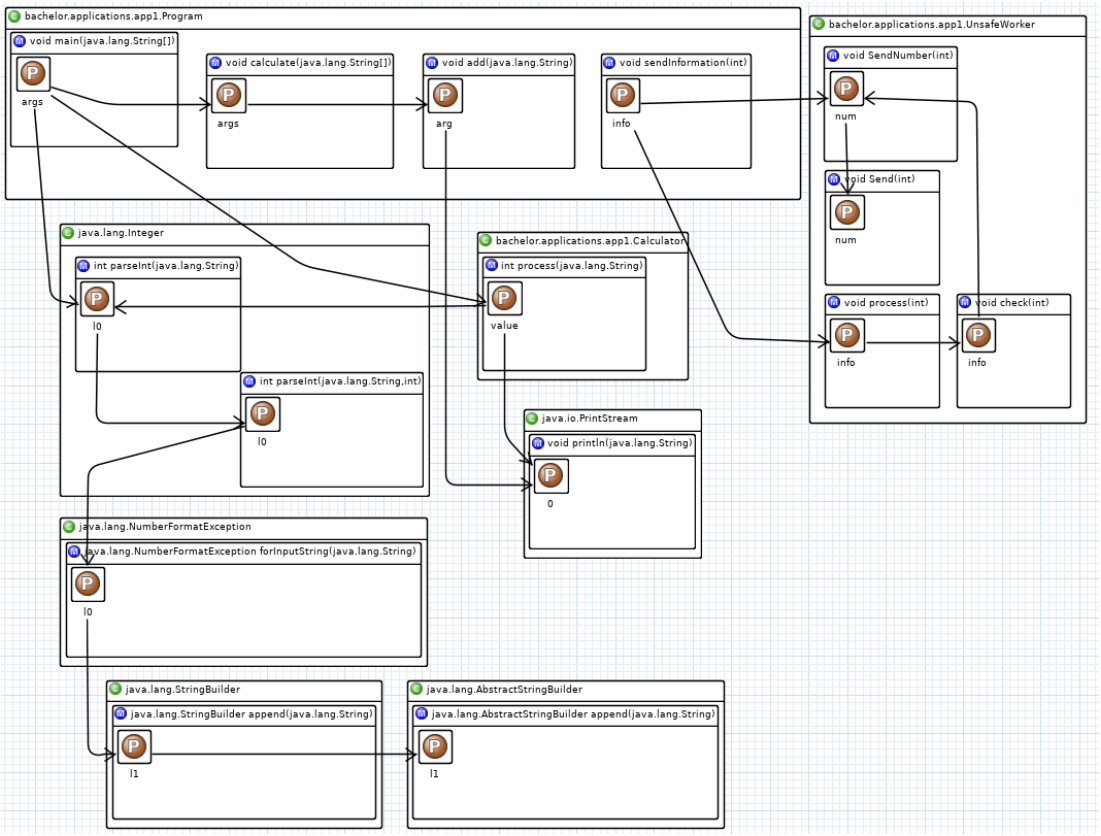


ABBILDUNG 4.1: Der generierte EDFD der Anwendung.

Nun werden die Datenflüsse betrachtet, die von den Einstiegspunkten zu unsicheren Senken führen. Die Analyse hat vier von fünf Datenflüssen erfolgreich abgebildet, sodass die Anfrage mit der Sicherheitswissensdatenbank vier Gefahren entdeckt hat. Der Datenfluss, der nach **println(int)** führt, wurde von der Analyse nicht vollständig erkannt und endet in einer Methode, dessen Rückgabewert sensitiv ist. Unter den Ergebnissen ist auch kein einziger falschpositiver Datenfluss, was Sinn ergibt, da die Anwendung weder Arrays noch Felder enthält und somit weder die Array- noch Objektinsensitivität zum Tragen kommt. Abbildung 4.2 stellt die Ergebnisse in einer Konfusionsmatrix dar.

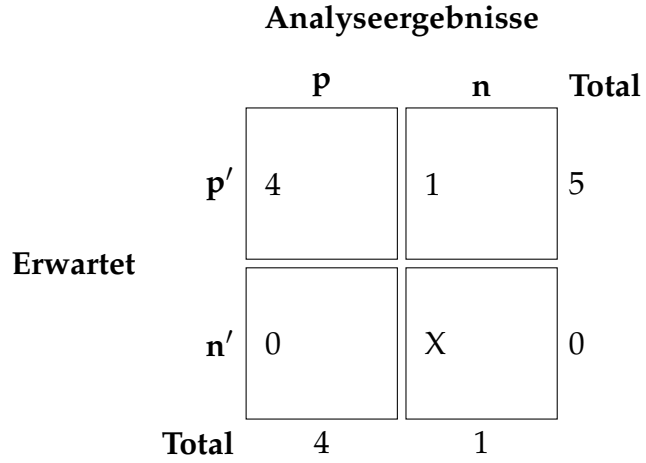


ABBILDUNG 4.2: Die Ergebnisse in einer Konfusionsmatrix.

Somit ergeben sich folgende Kennwerte:

$$Accuracy = \frac{CorrectPredictions}{TotalPredictions} = \frac{4}{5} = 0,8$$

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} = \frac{4}{4+0} = \frac{4}{4} = 1$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} = \frac{4}{4+1} = \frac{4}{5} = 0,8$$

Weil alle Kennwerte mindestens 0,8 betragen, macht die Analyse einen guten Eindruck. Allerdings ist im Sicherheitssektor Recall deutlich wichtiger als Precision, da jeder falsch-negative Datenfluss ein schwerwiegender Angriffspunkt sein könnte, der unentdeckt bleibt.

4.2.3 Testanwendung 2

Die zweite Testanwendung besteht aus drei Klassen und verwendet, wie die erste Testanwendung, nur wenige Methoden aus den Kernbibliotheken von Java, in denen intern nicht viele weitere Aufrufe stattfinden. Im Gegensatz zur ersten Anwendung kommen nun sowohl rekursive als auch zyklische Datenflüsse vor. Zusätzlich enthalten die Klassen Felder, die aber bei der gewählten Konfiguration keine Bedeutung haben. Tabelle 4.4 zeigt Kennwerte über die Ausmaße der Anwendung.

Name	Wert
Klassenanzahl	3
Methodenanzahl	6
SLOC (Java)	41
SLOC (Jimple)	93

TABELLE 4.4: Komplexität der Anwendung.

Konfiguration Tabelle 4.5 zeigt die eingestellten Einstiegspunkte und unsicheren Senken. Basierend auf dieser Konfiguration entstehen insgesamt zwölf unsichere Datenflüsse in der Anwendung.

	Methode	Parameter
Einstiegspunkt	Main.entry	1
Unsichere Senken	java.io.PrintStream.println	0
	java.lang.String.valueOf	0

TABELLE 4.5: Konfiguration der Anwendung.

Ergebnisse Zunächst zeigt Tabelle 4.6 Laufzeitdaten und Arbeitsspeicherverbrauch. Auch bei dieser Anwendung nimmt die Initialisierung die größte Zeit in Anspruch. Allerdings benötigt sie weitaus weniger Zeit als in der ersten Testanwendung, was daran liegt, dass die Methoden aus den Kernbibliotheken weniger Aufrufe ausführen.

Name	Wert
Laufzeit (Initialisierung)	5,148s
Laufzeit (Analyse)	0,035s
Laufzeit (EDFD Generierung)	0,722s
Arbeitsspeicherverbrauch	0,352 GB

TABELLE 4.6: Statistiken der Anwendung.

In Abbildung 4.3 ist der generierte EDFD abgebildet. Es fällt auf, dass der zyklische Datenfluss erkennbar ist. Allerdings weist ArchSecs grafischer Diagrammeditor eine Schwäche bei rekursiven Aufrufen auf, weil eine Kante von einem Element auf sich selbst nicht angezeigt wird. Dies ist aber kein gravierender Fehler, sondern eher ein Schönheitsfehler, da die Kante intern existiert und die Anfragen korrekt funktionieren.

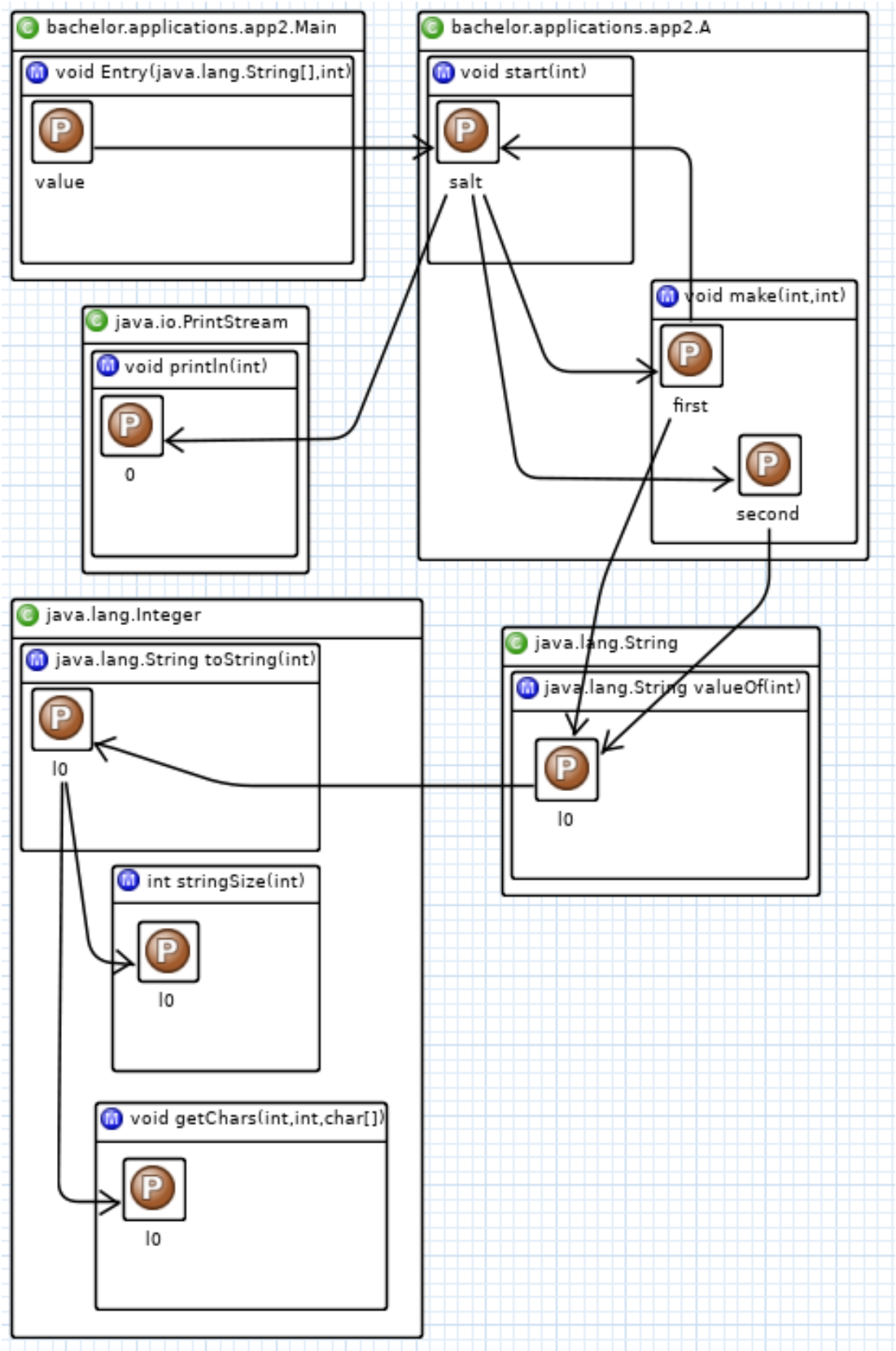


ABBILDUNG 4.3: Der generierte EDFD der Anwendung.

Nun werden die Datenflüsse betrachtet, die von den Einstiegspunkten zu unsicheren Senken führen. Die Analyse hat alle zwölf Datenflüsse erfolgreich abgebildet, sodass die Anfrage zwölf Gefahren entdeckt hat. Unter den Ergebnissen ist auch kein einziger falsch-positiver Datenfluss, was erneut Sinn ergibt, da die Anwendung keine Arrays besitzt und die übrigen Felder keine Bedeutung spielen. In Abbildung 4.4 sind die Ergebnisse in einer Konfusionsmatrix dargestellt.

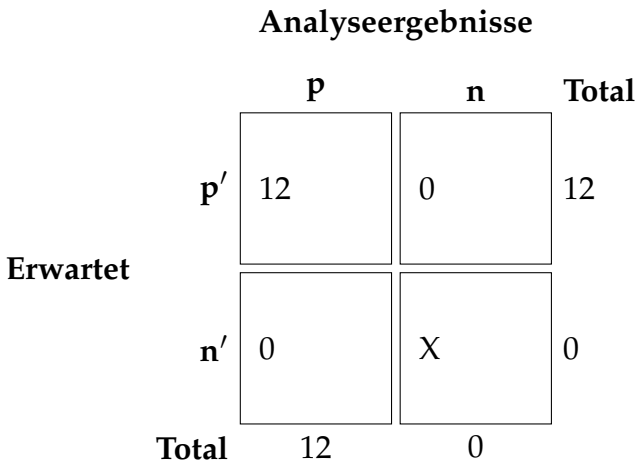


ABBILDUNG 4.4: Die Ergebnisse in einer Konfusionsmatrix.

Somit ergeben sich folgende Kennwerte:

$$Accuracy = \frac{CorrectPredictions}{TotalPredictions} = \frac{12}{12} = 1$$

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} = \frac{12}{12+0} = \frac{12}{12} = 1$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} = \frac{12}{12+0} = \frac{12}{12} = 1$$

Alle Werte liegen bei 100% und die Analyse hat perfekte Resultate geliefert. Somit läuft sie bei rekursiven und anderen zyklischen Datenflüssen stabil. Zu beachten ist jedoch, dass es sich in dieser Anwendung nur um einfache Datenflüsse gehandelt hat, die weder Rückgabewerte enthalten haben noch über Arrays oder Felder verliefen.

4.2.4 Testanwendung 3

Die dritte Testanwendung ist eine größere Anwendung mit einer Graphical User Interface (GUI). Die GUI ist mit Hilfe des Swing-Frameworks umgesetzt und ein Bibliotheksaufruf führt zu einer hohen Anzahl von internen Aufrufen. Neben zyklischen Datenflüssen, besitzt sie nun auch Datenflüsse, die über Arrays und Felder führen, sodass die wichtigsten Konstrukte vorhanden sind. Tabelle 4.7 zeigt Kennwerte über die Ausmaße der Anwendung.

Name	**Wert**
Klassenanzahl	4
Methodenanzahl	12
SLOC (Java)	171
SLOC (Jimple)	365

TABELLE 4.7: Komplexität der Anwendung.

Konfiguration Tabelle 4.8 zeigt die eingestellten Einstiegspunkte und unsicheren Senken. Basierend auf dieser Konfiguration entstehen insgesamt 20 unsichere Datenflüsse in der Anwendung.

	Methode	**Parameter**
Einstiegspunkt	PasswordManager.processPassword	1
Unsichere Senken	java.io.PrintStream.println	0
	java.lang.String.valueOf	0

TABELLE 4.8: Konfiguration der Anwendung.

Ergebnisse Zunächst zeigt Tabelle 4.9 Laufzeitdaten und Arbeitsspeicherverbrauch. Auch bei dieser Anwendung nimmt die Initialisierung die meiste Zeit in Anspruch. Da Swing im Hintergrund sehr viele Methoden aufruft, dauert diese sogar über neun Minuten. An dieser Stelle muss jedoch angemerkt werden, dass das Testsystem nur über 8 GB Arbeitsspeicher verfügt und aufgrund des hohen Bedarfs von über 15 GB Gebrauch von der Auslagerungsdatei machen musste, was den Prozess erheblich verlangsamt und die Laufzeiten verfälscht hat.

Name	Wert
Laufzeit (Initialisierung)	9m 37s
Laufzeit (Analyse)	1,102s
Laufzeit (EDFD Generierung)	8,766s
Arbeitsspeicherverbrauch	15,217 GB

TABELLE 4.9: Statistiken der Anwendung.

Aufgrund der Komplexität der Anwendung wird auf den EDFD verzichtet, da dieser nicht mehr kompakt und übersichtlich dargestellt werden kann. Nun werden die Datenflüsse betrachtet, die von einem Einstiegspunkt zu unsicheren Senken führen. Die Analyse hat 16 Datenflüsse erfolgreich abgebildet, sodass die Anfrage 16 Gefahren entdeckt hat. Unter den Ergebnissen finden sich fünf falsch-positive Datenflüsse, was aufgrund des Vorkommens von Arrays und Felder Sinn ergibt, da die Insensitivitäten der Analyse zum Tragen kommen. Abbildung 4.5 stellt die Ergebnisse in einer Konfusionsmatrix dar.

		Analyseergebnisse		
		p	**n**	**Total**
Erwartet	**p′**	16	4	20
	n′	5	X	5
	Total	21	4	

ABBILDUNG 4.5: Die Ergebnisse in einer Konfusionsmatrix.

Somit ergeben sich folgende Kennwerte (da die Analyse fünf falsch-positive Datenflüsse ermittelt hat, erhöht sich die Gesamtanzahl auf 25):

$$Accuracy = \frac{CorrectPredictions}{TotalPredictions} = \frac{16}{25} = 0,64$$

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} = \frac{16}{16+5} = \frac{16}{21} \approx 0,76$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} = \frac{16}{16+4} = \frac{16}{20} = 0,8$$

4.3 Vergleich mit LAPSE+

Bei LAPSE+ handelt es sich um eine Sicherheitssoftware in Form eines Plugins für die Eclipse IDE, die Schwachstellen in Java EE Anwendungen aufspüren kann und auf LAPSE basiert, welches 2006 von der SUIF Compiler Group von der Stanford University veröffentlicht wurde [11]. Im Gegensatz zu der in dieser Arbeit vorgestellten Analyse legt LAPSE+ den Fokus jedoch nicht auf die exakte Bestimmung von Datenflüssen und deren Visualisierung in EDFDs, sondern auf das Erkennen von Sicherheitsproblemen, also ob die Daten einer Quelle in eine unsichere Senke fließen. Nachdem in diesem Kapitel näher auf die Konfiguration von LAPSE+ eingegangen wird, werden die drei Testanwendungen getestet und die Ergebnisse mit den Analyseergebnissen verglichen. Bei der ersten Testanwendung werden zusätzlich die Ansichten von LAPSE+ erläutert, um die Funktionsweise zu verdeutlichen.

4.3.1 Konfiguration

Die Konfiguration erfolgt über drei verschiedene XML-Dateien: Kategorien, Quellen und Senken. Die Kategorien beschreiben die verschiedenen Schwachstellen, die auftreten können. LAPSE+ bietet mit **Information Leakage** eine passende Kategorie für unsichere Datenflüsse an, die im Folgenden für die Testanwendungen verwendet wird.

Quellen Während bei der entwickelten Analyse beliebige Parameter als Quellen festgelegt werden können, sind in LAPSE+ lediglich Methoden erlaubt, die Werte zurückgeben. Der Rückgabewert wird dann als sensitiv angesehen und verfolgt. Listing 4.3 zeigt die Angabe einer Quelle.

```
<source id="package.class.method()">
    <category>Information Leakage</category>
</source>
```

LISTING 4.3: Spezifikation einer Quelle in LAPSE+.

Senken Bei den Senken kommt in LAPSE+ genauso wie in der entwickelten Analyse jeder beliebige Methodenparameter in Frage. Listing 4.4 zeigt die Angabe einer Senke. Mit Hilfe des **vulnParam**-Tags kann ein Parameter über seine Position in der Methodensignatur als unsicher markiert werden.

```
<sink id="package.class.method(int,int,int)">
    <paramCount>3</paramCount>
    <vulnParam>0</vulnParam>
    <category>Information Leakage</category>
</sink>
```

LISTING 4.4: Spezifikation einer Senke in LAPSE+.

4.3.2 Vorbereitung der Testfälle

Um die Ergebnisse von LAPSE+ mit denen der Analyse vergleichen zu können, müssen die Testanwendungen angepasst werden, damit die sensitiven Parameter als Quellen konfiguriert werden können. Angenommen es existiert eine selbstgeschriebene **process**-Methode, deren erster Parameter sensitiv ist und verfolgt werden soll, so wird die aufgerufene Methode (siehe Listing 4.5) dahingehend verändert, dass der entsprechende Parameter unverändert aus einer Methode zurückgegeben und ersetzt wird (siehe Listing 4.6).

```
void process(int first, String second) {
    // ...
}
```

LISTING 4.5: Code vor der Anpassung.

```
void getFirst(int first) { return first; }

// ...

void process(int first, String second) {
    first = getFirst(first);

    // ...
}
```

LISTING 4.6: Code nach der Anpassung.

Handelt es sich um eine Bibliotheksfunktion ist dieses Vorgehen nicht möglich, da der Quellcode unter Umständen nicht zur Verfügung steht. In diesem Fall muss der Parameter bei jedem Aufruf mit der neu erstellten Methode ersetzt werden. Beide Anpassungen verändern die Semantik des Programms nicht und verfälschen daher auch nicht die ermittelten Datenflüsse. Anschließend werden die neu erstellten Methoden als Quellen definiert und die unsicheren Senken in die XML-Syntax überführt. Zum Schluss wird die Anwendung zusammen mit allen XML-Dateien als Projekt in der Eclipse IDE importiert, sodass LAPSE+ eingesetzt werden kann.

4.3.3 Testanwendung 1

Abbildung 4.6 zeigt alle Ansichten von LAPSE+ mit Ergebnissen der ersten Testanwendung. Während die **Vulnerability Sources**-Ansicht alle gefundenen Quellen im Quellcode anzeigt, werden in der **Vulnerability Sinks**-Ansicht alle gefundenen unsicheren Senken im Quellcode angezeigt. Die **Provenance Tracker**-Ansicht zeigt alle Datenflüsse, die mit einer Rückwärtspropagation von einer unsicheren Senke aus rekonstruiert wurden [11].

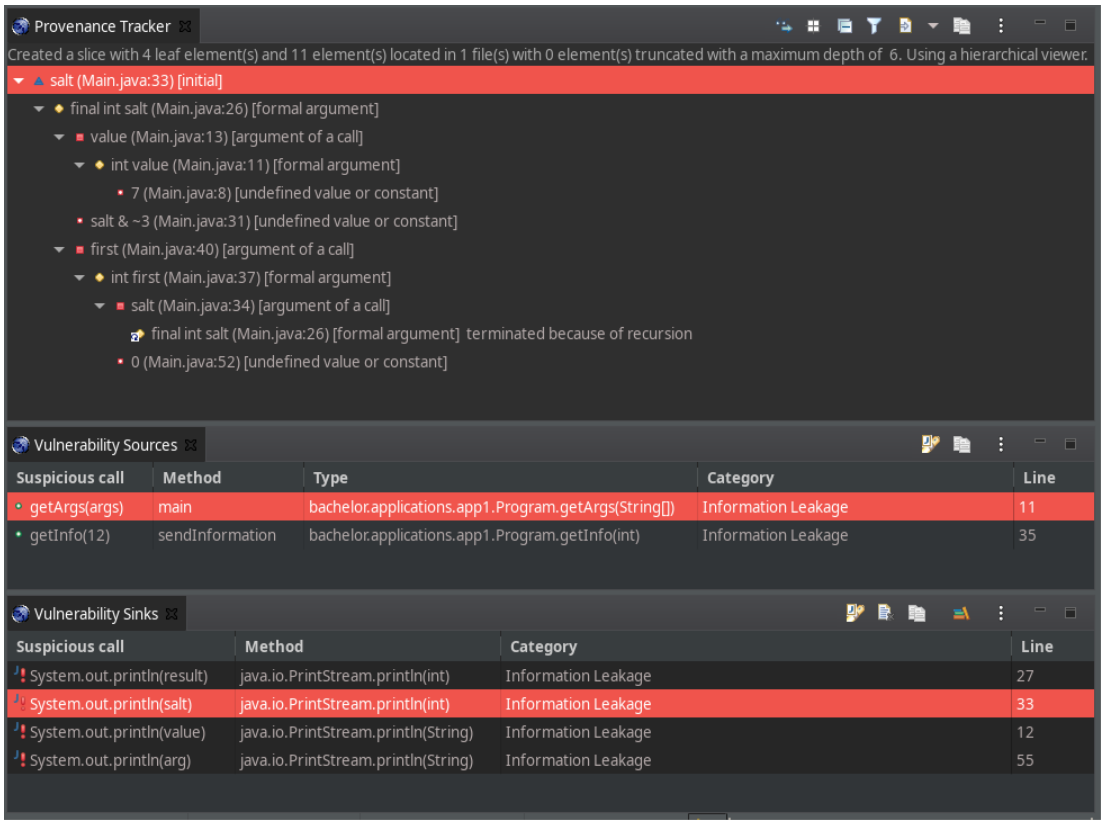


ABBILDUNG 4.6: Alle LAPSE+ Ansichten mit Ergebnissen.

Wie zuvor bei der Analyse werden auch die Ergebnisse von LAPSE+ in einer Konfusionsmatrix (siehe Abbildung 4.7) dargestellt, in der aus den gleichen Gründen auf die Angabe von wahr-negativen Datenflüssen verzichtet wird:

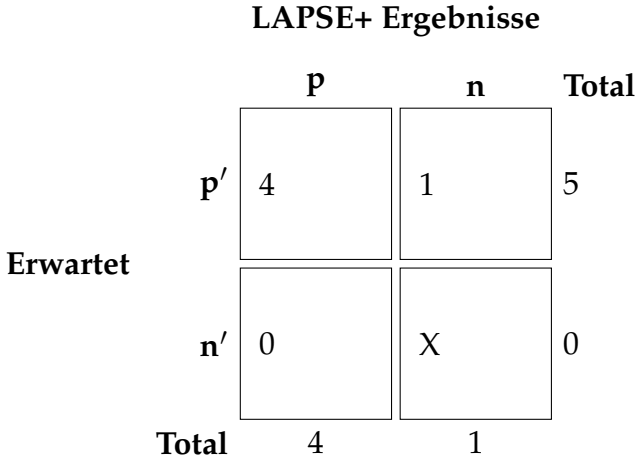


ABBILDUNG 4.7: Die Ergebnisse in einer Konfusionsmatrix.

Somit ergeben sich folgende Kennwerte:

$$Accuracy = \frac{CorrectPredictions}{TotalPredictions} = \frac{4}{5} = 0,8$$

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} = \frac{4}{4+0} = \frac{4}{4} = 1$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} = \frac{4}{4+1} = \frac{4}{5} = 0,8$$

Ein Vergleich mit den Ergebnissen der in dieser Arbeit entwickelten Analyse zeigt, dass sowohl die Konfusionsmatrix als auch alle Kennwerte identisch sind. Allerdings unterscheidet sich der falsch-negative Datenfluss. Im Gegensatz zur anderen Analyse hat LAPSE+ alle Datenflüsse zu den verschiedenen **println**-Aufrufen erfolgreich ermittelt. Dafür wurde der Datenfluss zur **send**-Methode nicht entdeckt.

4.3.4 Testanwendung 2

Zusammenfassend hat LAPSE+ folgende Ergebnisse geliefert:

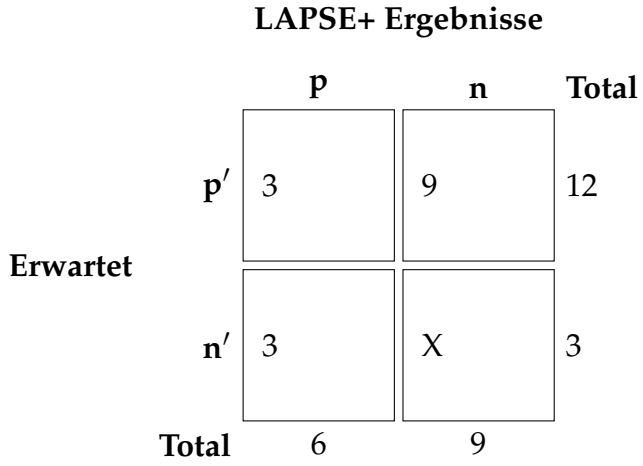


ABBILDUNG 4.8: Die Ergebnisse in einer Konfusionsmatrix.

Somit ergeben sich folgende Kennwerte:

$$Accuracy = \frac{CorrectPredictions}{TotalPredictions} = \frac{3}{15} = 0,2$$

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} = \frac{3}{3+3} = \frac{3}{6} = 0,5$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} = \frac{3}{3+9} = \frac{3}{12} = 0,25$$

Im Vergleich mit der Analyse schneidet LAPSE+ in dieser Testanwendung deutlich schlechter ab, da es nur 25% der unsicheren Datenflüsse vollständig ermittelt hat. Zwar hat es weitere Datenflüsse ermittelt, die allerdings bei rekursiven Aufrufen oder bei Berechnungen von Argumenten stoppten und somit unvollständig und nicht mehr mit der Quelle in Verbindung gebracht werden können. Darüber hinaus wurden Datenflüsse ermittelt, die nicht von einer spezifizierten Quelle ausgehen.

4.3.5 Testanwendung 3

Zusammenfassend hat LAPSE+ folgende Ergebnisse geliefert:

LAPSE+ Ergebnisse

Erwartet	**p**	**n**	**Total**
p′	8	12	20
n′	5	X	5
Total	13	12	

ABBILDUNG 4.9: Die Ergebnisse in einer Konfusionsmatrix.

Somit ergeben sich folgende Kennwerte:

$$Accuracy = \frac{CorrectPredictions}{TotalPredictions} = \frac{8}{25} = 0,32$$

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive} = \frac{8}{8+5} = \frac{8}{13} \approx 0,62$$

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative} = \frac{8}{8+12} = \frac{8}{20} = 0,4$$

Im Vergleich mit der Analyse schneidet LAPSE+ in dieser Testanwendung erneut deutlich schlechter ab, da es nur 40% der unsicheren Datenflüsse fehlerfrei ermittelt hat. Weitere Datenflüsse, die zum Teil korrekt waren, endeten wieder bei Berechnungen abrupt. Auch hier wurden wieder mehrere falschpositive Datenflüsse erkannt.

4.4 Auswertung

In diesem Abschnitt werden alle Ergebnisse von der entwickelten Analyse sowie von LAPSE+ ausgewertet und verglichen. Bei Betrachtung der Laufzeiten (siehe Abbildung 4.10) und dem Arbeitsspeicherverbrauch (siehe Abbildung 4.11) aller Testanwendungen fällt auf, dass die Initialisierung gefolgt von der EDFD-Generierung den größten Teil der Gesamtlaufzeit ausmacht und der Arbeitsspeicherverbrauch bei größeren Anwendungen enorm zunimmt. Interessant wäre es, die Laufzeiten im Hinblick auf die SLOC zu betrachten und zu untersuchen, wie diese korrelieren. Allerdings sind die angegebenen SLOC ein ungeeignetes Maß für diesen Zweck, da lediglich die Zeilen des Anwendungscodes und nicht von Bibliotheksfunktionen berücksichtigt sind. Gerade bei der dritten Testanwendung werden innerhalb des Swing-Frameworks Tausende zusätzliche Zeilen im Hintergrund ausgeführt, welche der eigentliche Grund für die hohen Werte sein dürften. Dass das Testsystem nur 8 GB Arbeitsspeicher verbaut hat und auf die Auslagerungsdatei zugreifen musste, darf im Hinblick auf die Laufzeit auch nicht vernachlässigt werden.

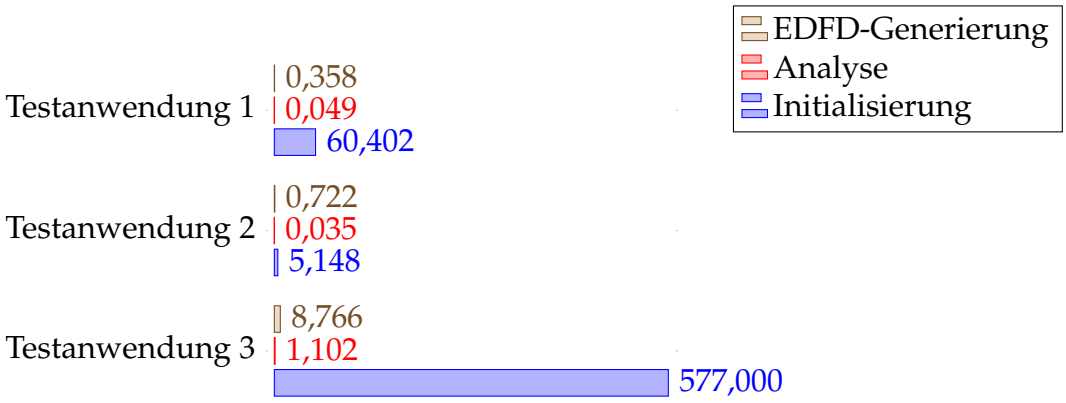


ABBILDUNG 4.10: Alle Laufzeiten in Sekunden.

Der hohe Speicherverbrauch lässt sich durch die Architektur von Soot und VASCO erklären. Schließlich müssen alle Informationen über jede Klasse, Methode und Jimple-Anweisung während der gesamten Analyse und der EDFD-Generierung zur Verfügung stehen. Im Analyseprozess werde darüber hinaus Werte-Kontexte für jeden unterschiedlichen Methodenaufruf und Ein- sowie Ausgabeverbände für jede Anweisung instanziiert.

Tabelle 4.10 präsentiert eine Übersicht über die wichtigsten Kennwerte der Ergebnisse von der Analyse sowie LAPSE+ und zeigt die auf ganze Zahlen

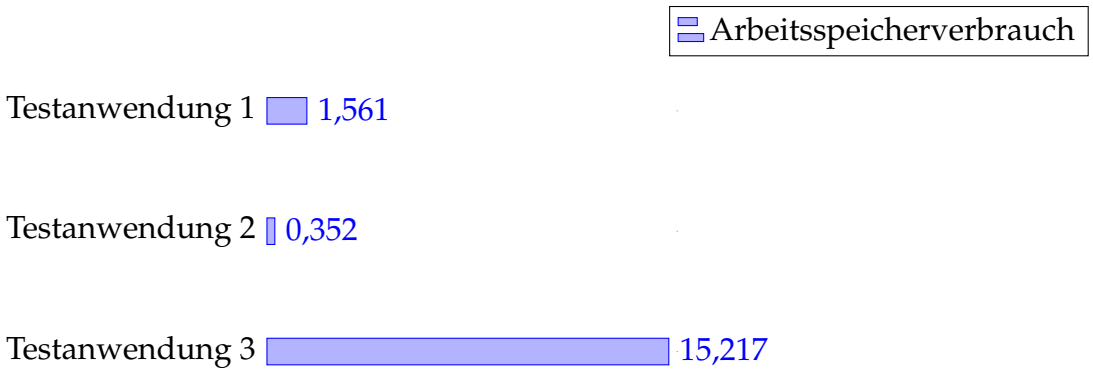


ABBILDUNG 4.11: Alle Arbeitsspeicherverbräuche in GB.

gerundeten gewichteten arithmetischen Mittelwerte (siehe Gleichung 4.1) der einzelnen Kennwerte. Die Gewichtung ist notwendig, da die Testanwendungen einen unterschiedlichen Umfang und somit unterschiedlich viele Datenflüsse aufweisen. Weil drei Testanwendungen betrachtet werden, beträgt die Anzahl der Stichproben $n = 3$. Ein Stichprobenumfang w_i entspricht dem ungekürzten Nenner des jeweiligen Kennwerts der i-ten Testanwendung. Gleichung 4.2 verdeutlicht die Rechnung anhand von Accuracy der Analyse. Bei dieser ist ein Stichprobenumfang die Anzahl aller Datenflüsse einer Testanwendung, außer die wahr-negativen Datenflüsse.

$$\bar{x} = \frac{\sum_{i=1}^{n} w_i * x_i}{\sum_{i=1}^{n} w_i} \tag{4.1}$$

$$\bar{x}_{ACCURACY} = \frac{5 \cdot 0,8 + 12 \cdot 1 + 25 \cdot 0,64}{5 + 12 + 25} = \frac{32}{42} \approx 0,76 \tag{4.2}$$

Bei dem Vergleich der beiden Analysewerkzeuge fällt direkt auf, dass die Analyse deutlich besser abschneidet als LAPSE+. Obwohl LAPSE+ nahezu die gleiche Anzahl an Bedrohungen erkennt, ist es nicht in der Lage die Datenflüsse vollständig von einer Quelle zur entsprechenden unsicheren Senke zu

rekonstruieren und beendet die Rekonstruktion bei dem Auftreten von Rekursion oder anderen Berechnungen.

Anwendung	Analyse			LAPSE+		
	Accuracy	Precision	Recall	Accuracy	Precision	Recall
1	80%	100%	80%	80%	100%	80%
2	100%	100%	100%	20%	50%	25%
3	64%	76%	80%	32%	62%	40%
Mittelwert	76%	86%	86%	33%	65%	41%

TABELLE 4.10: Übersicht sämtlicher Ergebnisse der Analyse und LAPSE+.

Im Hinblick auf das Ziel dieser Arbeit, nämlich unsichere Datenflüsse zu erkennen, ist Recall der wichtigste Wert, da er angibt, wie viele unsichere Datenflüsse erkannt wurden. Mit einem Durchschnitt von 86% erreicht die Analyse einen Prozentsatz, der befriedigend ist. Allerdings heißt das im Umkehrschluss, dass 14% der unsicheren Datenflüsse nicht erkannt wurden, was bei sicherheitskritischen Anwendungen gefährlich ist. Ein Sicherheitsreview, welches von Experten durchgeführt wird, ersetzt die Analyse somit nicht. Der Precision-Kennwert sagt aus, wie viel von den erkannten unsicheren Datenflüssen in der Tat unsicher sind. Aufgrund der Array- und Objektinsensitivität entstehen häufig falsch-positive Ergebnisse, was zwar Mehraufwand für den Gutachter bedeutet, aber keine übersehene Schwachstelle bedeutet. Weil lediglich die dritte Testanwendung sowohl zyklische Datenflüsse als auch für Datenflüsse relevante Arrays und Felder besitzt, sind die Kennwerte von dieser am aussagekräftigsten. Die Ergebnisse von Anwendungen aus der realen Welt pendeln sich vermutlich in diesem Bereich ein. Abschließend sei erwähnt, dass die durchgeführte Evaluation nur auf Fallstudien basiert und zusätzlich mit einer groß angelegten Evaluation ergänzt werden sollte, die auf möglichst vielen realen Desktop-, Web- sowie Enterprise-Anwendungen aus verschiedenen Domänen von verschiedenen Entwicklern basiert und so ein breites Spektrum an Java-Konstrukten umfasst. Auch die Untersuchung der Laufzeit und des Arbeitsspeicherverbrauchs anhand eines einzigen Systems ist nicht aussagekräftig und sollte sowohl Desktop- als auch Notebook-Systeme aus verschiedenen Preissegmenten umfassen. Dann kann die Evaluation mit wissenschaftlich fundierten Aussagen abgeschlossen werden.

Kapitel 5

Fazit und Ausblick

In dieser Arbeit wurde eine statische Datenflussanalyse zur Extraktion von Datenflüssen in Java-basierten Anwendungen für Sicherheitsüberprüfungen vorgestellt. Zunächst wurden elementare Themen wie Programmanalysen, Datenflussanalysen und Datenflussdiagramme erläutert, die das Fundament dieser Arbeit bilden. Im Anschluss wurden die Ziele und das Verhalten der Analyse beschrieben, bevor ein Blick auf die verwendeten Bibliotheken Soot, VASCO und ArchSec geworfen wurde. Eine detaillierte Erläuterung der Implementierung der interprozeduralen Datenflussanalyse sowie der EDFD-Generierung verdeutlichte die wesentlichen Algorithmen. Die implementierte Analyse wurde daraufhin anhand von automatisierten Tests sowie drei konstruierten Testanwendungen hinsichtlich der Erkennung von unsicheren Datenflüssen evaluiert und mit LAPSE+, einem ähnlichen Analysewerkzeug, verglichen.

Im Großen und Ganzen hat die Analyse im Durchschnitt 86% der unsicheren Datenflüsse von einer sensitiven Quelle zu einer unsicheren Senke aufgespürt, während LAPSE+ lediglich auf 41% kommt. Zwar hat LAPSE+ auch weitere unsichere Datenflüsse erkannt, konnte sie aber im Gegensatz zur Analyse nicht vollständig zu einer sensitiven Quellen zurück verfolgen und rekonstruieren. Bei Datenflüssen, die nicht von der Analyse erkannt wurden sind häufig Rückgabewerte von Methoden oder Arrays involviert. Die Probleme mit Arrays gehen auf die Architektur zurück und haben ihren Ursprung in der Alias-Funktionalität des Verbands. Die genaue Ursache ist jedoch unklar und kann in weiteren Tests untersucht werden.

Die Analyse hat die EDFDs für die Testanwendungen erfolgreich generiert und stellt die ermittelten Datenflüsse zu Parametern oder Feldern größtenteils

korrekt dar. Allerdings existieren einige Schönheitsfehler. Zum einen ist ArchSecs EDFD-Ansicht zurzeit nicht in der Lage eine Kante eines Knoten auf sich selbst anzuzeigen, wodurch rekursive Aufrufe nicht erkennbar sind und zum anderen wird der Datenfluss von Feldern zu Parametern nicht korrekt dargestellt, was auf einen Architekturfehler der Analyse zurück zu führen ist, da lediglich Parameter als Vorgänger vorgesehen sind. Dies könnte elegant mittels Vererbung und Polymorphie gelöst werden, sodass verschiedene Arten von Vorgängern zugewiesen werden können. Ein weiterer unschöner Aspekt ist, dass der Datenfluss über Instanz- und Klassenfelder sehr unpräzise ermittelt wird. So führt eine Kante von dem Feld direkt zur Methode, die dieses als Argument übergeben bekommt, ohne dass die aufrufende Methode erkennbar ist. Hier könnte die Analyse die lokalen Variablen speichern, die die Felder vor einem Aufruf beinhalten und in den EDFD mit aufnehmen.

Der Arbeitsspeicherverbrauch der Analyse nimmt mit größeren Anwendungen deutlich zu. Das liegt zum großen Teil daran, dass Soot und VASCO alle Informationen über jede Klasse, Methode sowie Jimple-Anweisung über die gesamte Berechnungsdauer zwischenspeichern. Der komplexe Verband der Analyse belastet jedoch ebenfalls den Arbeitsspeicher, da für jede Anweisung zwei Verbände erstellt werden. Haben zwei Anweisungen theoretisch den gleichen Verband, so wird der Verband inklusive seiner Daten trotzdem komplett kopiert. Diese Operation ist unnötig und könnte vermieden werden. Eine Optimierungsmöglichkeit wäre hier, dass Verbandinstanzen von mehreren Anweisungen referenziert werden und nur dann für eine Anweisung kopiert werden, wenn diese Änderungen vornimmt.

Eine sinnvolle zukünftige Ergänzung der Analyse wäre die sogenannte Desinfektion, die von LAPSE+ ebenfalls unterstützt wird. Mather et al. [8] definieren Desinfektion als den Prozess, Daten von einem Medium zu entfernen, bevor dieses in einer Umgebung zum Einsatz kommt, die keinen akzeptablen Schutz für die entfernten Daten geboten hätte. Dies kann auf die Analyse übertragen werden. Angenommen ein Parameter, der ein Passwort in Klartext repräsentiert, wird als sensitiv festgelegt, verfolgt und landet schließlich in einer Methode, die das Passwort verschlüsselt, sodass die Rückgabe nicht mehr sensitiv ist. Dann ergibt es keinen Sinn, den Rückgabewert weiterhin als sensitiv zu betrachten, da dies falsch-positive Datenflüsse zur Folge hat. Eine mögliche Vorgehensweise dafür wäre die Erweiterung der Konfiguration,

sodass Desinfizierer-Methoden mit betroffenen Argumenten definiert werden können, die später bei dem interprozeduralen Datenfluss aus den Verbänden entfernt werden. Abgesehen von der Desinfektion wäre es sinnvoll in Betracht zu ziehen das Verhalten der Analyse anzupassen, sodass sie sich sowohl array- als auch objektsensitiv verhält. Beide Insensitivitäten sorgen momentan für vermehrt falsch-positiv erkannte Datenflüsse, was die Arbeit bei der manuellen Überprüfung erschwert.

Die ursprüngliche Idee dieser Arbeit war es, unsichere Datenflüsse in Java-basierten Anwendungen erkennen zu können und auf diese Weise insbesondere kleinere und mittelgroße Unternehmen bei der architekturellen Risikoanalyse zu unterstützen. Mit einer durchschnittlichen Erkennungsquote von 86% bei den drei Testanwendungen erreicht die entwickelte Analyse das Ziel durchaus befriedigend, wobei die Effektivität mit einer groß angelegten Evaluation mit möglichst vielen verschiedenen Anwendungen weiter untersucht werden muss. Durch eine Integration der Analyse in gängige IDEs wie Eclipse, könnten die unsicheren Datenflüsse farblich hervorgehoben werden, um den Umgang weiter zu erleichtern. Zwar ersetzen die Analyse und ihre ermittelten Ergebnisse keine fachkundigen Reviews von Experten, allerdings bilden sie eine gute Grundlage für erste manuelle Sicherheitsüberprüfungen und bieten Unterstützung bei der architekturellen Risikoanalyse.

Literatur

[1] Bernhard J. Berger, Karsten Sohr und Rainer Koschke. „Automatically Extracting Threats from Extended Data Flow Diagrams“. In: *Engineering Secure Software and Systems*. Hrsg. von Juan Caballero, Eric Bodden und Elias Athanasopoulos. Cham: Springer International Publishing, 2016, S. 56–71. ISBN: 978-3-319-30806-7.

[2] Claus Brabrand u. a. „Intraprocedural Dataflow Analysis for Software Product Lines“. In: *Proceedings of the 11th Annual International Conference on Aspect-Oriented Software Development*. AOSD '12. Potsdam, Germany: Association for Computing Machinery, 2012, 13–24. ISBN: 9781450310925. DOI: 10.1145/2162049.2162052. URL: https://doi.org/10.1145/2162049.2162052.

[3] B. Chess und G. McGraw. „Static analysis for security“. In: *IEEE Security Privacy* 2.6 (2004), S. 76–79. DOI: 10.1109/MSP.2004.111.

[4] Riccardo Focardi und Roberto Gorrieri. „Non interference: Past, present and future“. In: *Proceedings of DARPA Workshop on Foundations for Secure Mobile Code*. Citeseer. 1997, S. 26–28.

[5] Nadime Francis u. a. „Cypher: An Evolving Query Language for Property Graphs“. In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: Association for Computing Machinery, 2018, 1433–1445. ISBN: 9781450347037. DOI: 10.1145/3183713.3190657. URL: https://doi.org/10.1145/3183713.3190657.

[6] Vladimir Khorikov. *Unit Testing Principles, Practices, and Patterns - Effective testing styles, patterns, and reliable automation for unit testing, mocking, and integration testing with examples in C#*. Birmingham: Manning Publications, 2020. ISBN: 978-1-617-29627-7.

[7] Rainer Koschke. *Software-Reengineering*. Techn. Ber. Universität Bremen, 2019.

[8] Tim Mather, Subra Kumaraswamy und Shahed Latif. *Cloud Security and Privacy - An Enterprise Perspective on Risks and Compliance*. Sebastopol: Ö'Reilly Media, Inc.", 2009. ISBN: 978-1-449-37951-3.

[9] Nurzhan Nurseitov u. a. „Comparison of JSON and XML data interchange formats: a case study." In: *Caine* 9 (2009), S. 157–162.

[10] Rohan Padhye und Uday P. Khedker. „Interprocedural Data Flow Analysis in Soot Using Value Contexts". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*. SOAP '13. Seattle, Washington: Association for Computing Machinery, 2013, 31–36. ISBN: 9781450322010. DOI: 10.1145/2487568.2487569. URL: https://doi.org/10.1145/2487568.2487569.

[11] Pablo Martín Pérez, Joanna Filipiak und José María Sierra. „Lapse+ static analysis security software: Vulnerabilities detection in java ee applications". In: *Future Information Technology*. Springer, 2011, S. 148–156.

[12] Raja Vallée-Rai u. a. „Soot - a Java Bytecode Optimization Framework". In: *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, S. 13.

[13] Wolfgang Wögerer. *A survey of static program analysis techniques*. Techn. Ber. Citeseer, 2005.

[14] Saurabh Zunke und Veronica D'Souza. „Json vs xml: A comparative performance analysis of data exchange formats". In: *Int J Comput Sci Netw* 3.4 (2014), S. 257–261.