technische universität
dortmund

Proseminararbeit

Analysetechniken für Informationssicherheit: Einleitung & Grundlagen

Alexander Schäferdiek

31. Januar 2014

Gutachter: Prof. Dr. Jan Jürjens

Prof. Dr. Jan Jürjens Lehrstuhl 14 Software Engineering
Fakultät Informatik
Technische Universität Dortmund
Otto-Hahn-Straße 14
44227 Dortmund
http://www-jj.cs.uni-dortmund.de/secse

Alexander Schäferdiek
alexander.schaeferdiek@tu-dortmund.de
Matrikelnummer: 140177
Studiengang: Bachelor Informatik

Werkzeugunterstützung für sichere Software
Thema: Analysetechniken für Informationssicherheit: Einleitung & Grundlagen

Eingereicht: 29. Januar 2014

Betreuer:

Prof. Dr. Jan Jürjens Lehrstuhl 14 Software Engineering
Fakultät Informatik
Technische Universität Dortmund
Otto-Hahn-Straße 14
44227 Dortmund

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Dortmund, den 29. Januar 2014

Alexander Schäferdiek

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

1.1 Vorwort

Sicherheit spielt in der heutigen Zeit eine immer wichtigere Rolle in Informationssystemen. Mit der Entwicklung und vor allem Weiterentwicklung vieler dieser IT Systeme steigt die Komplexität. Folglich werden auch die Anforderungen an ihre Sicherheit höher. In fast allen Bereichen der aktuellen Wirtschaft und Dienstleistungsinformatik sind sichere Grundsysteme eine der wichtigsten und ersten Anforderungen, die ein Auftraggeber stellt, um seine eigene Infrastruktur zu schützen und den unbefugten Zugriff von Dritten auf die eigene Infrastruktur systematisch zu verhindern. Um diese Art von Sicherheit zu gewährleisten, bedarf es der theoretischen Analyse von Code und der Entwicklung und Einhaltung von universell anwendbaren Vorgehensweisen im Bezug auf Schutz der immer schwieriger zu analysierenden IT Infrastrukturen.
Die systematische Analyse von Code lässt sich in zwei verschiedene Herangehensweisen unterteilen [**Datta2010**]:

a) statische Analyse, d.h. ohne Ausführung des Codes

b) **Datenflussanalyse**, d.h. unter Abstrahierung der möglichen Zustände eines Programmes während der Laufzeit

Diese Ausarbeitung widmet sich der **Datenflussanalyse** und der theoretischen Methodik, Code zu analysieren und alle möglichen Zustände innerhalb eines Programmes zu abstrahieren, um mithilfe der Automatentheorie und der klassischen Logik einzigartige Herangehensweisen zu entwickeln, die universell auf jede Art von Code zur Sicherheitsanalyse beitragen.

1.2 Struktur

Die Arbeit ist in drei verschiedene Kapitel unterteilt. Beginnend mit dem Kapitel „Einleitung“, in welchem Hintergründe und die Motivation für das Präsentieren dieser theoretischen Sicherheitsmethoden dargestellt werden, folgt die Beschreibung der „Ziel- und Problemstellungen“. In diesem Kapitel werden Hintergründe und notwendige Erläuterungen zu bereits feststehenden Konstrukten aus der Literatur kurz erläutert, um ein Grundverständnis für die Problematik zu generieren.

Der nächste Abschnitt „Lösungsansätze“ beschäftigt sich mit der Methodik zur

Datenflusssicherheitsanalyse von Programmen. Dieses Kapitel schafft eine Brücke zwischen den aus der Literatur bereits bekannten Konstrukten, den Kontrollflussgraphen und erweitert diese mithilfe der Automatentheorie (**Weighted Pushdown Systems**, WPDS) auf Basis von [**Datta2010**]. Zunächst wird hier auf Kontrollflussgraphen eingegangen, welche den Datenfluss eines Programmes visuell darstellen. Anhand dieser ersten Abstraktion werden WPDS eingeführt. Hier liegt der Fokus vor allem auf der Boolean-Domäne. Es werden wichtige Begriffe wie *Join-Over-All-Valid-Paths* (JOVP) und *prestar* erläutert und anhand eines Beispieles verdeutlicht.

Des Weiteren wird in diesem Kapitel der Begriff „Datalog“, eine an Prolog angelehnte und weniger mächtige klassische logische Programmiersprache, basierend auf der Prädikatenlogik, vorgestellt. In diesem Kapitel werden durch [**Schoening2000**] wichtige elementare Logikdefinitionen, wie z.B. das Herbrand Univerum, die Resolution und die Findung minimaler Modelle wiederholt und aufbereitet, welche z.B. für die Analyse von Sicherheitsstrategien benötigt werden [**Datta2010**].

Abschließend beinhaltet das letzte Kapitel „Zusammenfassung & Ausblick“ dieser Ausarbeitung einen kurzen Überblick bisheriger Ergebnisse und beschäftigt sich mit möglichen Anwendungsfällen der in dieser Arbeit enthaltenen Methoden zur Datenflusssicherheitsanalyse.

2 Ziel- und Problemstellungen

Das Kapitel „Ziel- und Problemstellungen“ beinhaltet Grundlagen zur Analyse der Datenflussproblematik. Die Frage, die sich hier stellt, ist:

„Welche möglichen Zustände kann ein Programm einnehmen?“

Um diese beantworten zu können, werden zunächst einige Grundlagen benötigt. Zur Visualisierung von Programmen werden in dieser Ausarbeitung Kontrollflussgraphen benutzt und im folgenden definiert.

2.1 Vom Programm zum Kontrollflussgraphen

Definition 2.1 *Kontrollflussgraph (CFG). Ein Kontrollflussgraph G ist eine visuelle Repräsentation eines Programms P, bestehend aus G = (V, E). Die Knotenmenge V enhält alle Anweisungen und die Kantenmenge E enhält alle möglichen Wege von* (v_1, v_2).

```
1  int x,y;
2  int example() {
3      x = 1;
4      y = x + 3;
5      return y;
6  }
```

Abbildung 2.1: Beispielprogramm P.
Quelle: Eigene Darstellung.

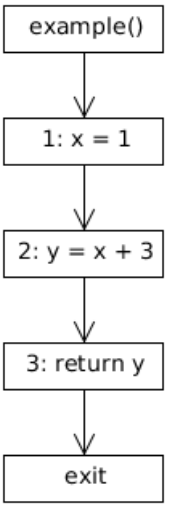

Abbildung 2.2: CFG zu Programm P.
Quelle: Eigene Darstellung.

Wie in Abbildung 2.2 zu sehen, ist der CFG des Programmes P (Abbildung 2.1) einfach zu generieren. Jede Kante e stellt einen Übergang zu einer anderen Anweisung

$(v \in V)$ dar.

CFGs für einzelne Programmprozeduren lassen sich mithilfe von Interprozeduralen Kontrollflussgraphen erweitern und bieten so die Möglichkeit, jede Prozedur in einem Programm P zu modellieren. So können innerhalb einer Anwendung verschiedene Funktionen mithilfe von einzelnen CFGs realisiert werden. Nach [**Datta2010**] sind ICFGs wie folgt definiert.

Definition 2.2 *Interprozeduraler Kontrollflussgraph (ICFG). Ein interprozeduraler Kontrollflussgraph ist die Zusammensetzung mehrerer CFGs, wobei jeder einzelne CFG eine Prozedur darstellt und somit einen call und return Knoten besitzt. G besitzt außerdem einen einzigartigen enter und exit Knoten.*

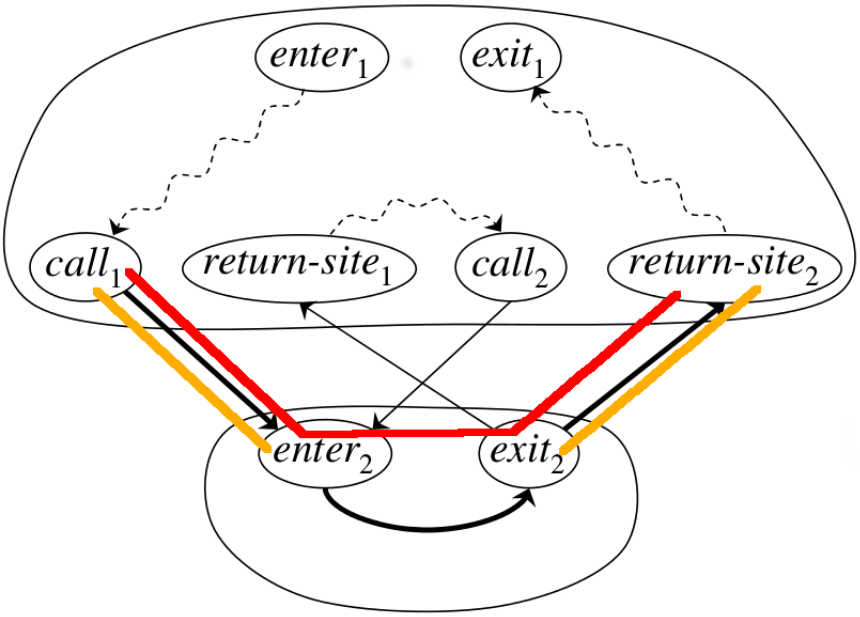

Abbildung 2.3: ICFG. **Quelle:** In Anlehnung an [**Datta2010**].

Wie im ICFG 2.3 zu sehen ist, werden call mit enter Knoten verbunden und return zu exit Knoten. ICFGs beantworten die oben gestellte Frage. Das tun sie. Ein typischer Algorithmus möchte für alle Knoten $v \in V$ approximieren, welche möglichen Zustände man erreichen kann, wenn man gerade im Knoten v ist. Man erhält folglich alle erreichbaren Knoten im gesamten ICFG. Solche Algorithmen berücksichtigen ebenfalls unerreichbare Knoten, durchlaufen den Graphen und reduzieren die Problematik auf das Erreichbarkeitsproblem in Graphen.
In Abbildung 2.3 ist ein solcher fehlerhafter und unerreichbarer Pfad sichtbar. Der Pfad $[call_1, enter_2, exit_2, return_2]$ (hier rot markiert) ist ein falscher Pfad. Die return $(exit_2 \rightarrow return_2)$ und call Kante $(call_1 \rightarrow enter_2)$ passen nicht zu einander (orange markiert), da diese in unterschiedlichen Knoten münden.

2.2 Vom ICFG zu allen möglichen Speicherzuständen

Wie zuvor beschrieben, können Algorithmen Obermengen von Knoten berechnen, die von einem Knoten $v \in V$ erreichbar sind. Diese Mengen beschreiben mögliche Speicherinhalte, wenn der Knoten v erreicht wurde. Die formale Bestimmung aller Pfade soll in diesem Abschnitt durch die JOIN-Operation in einem ICFG definiert werden.

```
1   int y;
2
3   void firstFunction() {
4       int x = 1;
5       int y = 2;
6
7       secondFunction(x);
8       ...
9   }
10  void secondFunction(int p) {
11      if (...)
12          y = 3;
13      else
14          y = p;
15  }
```

Abbildung 2.4: Programm X. **Quelle:** Eigene Darstellung.

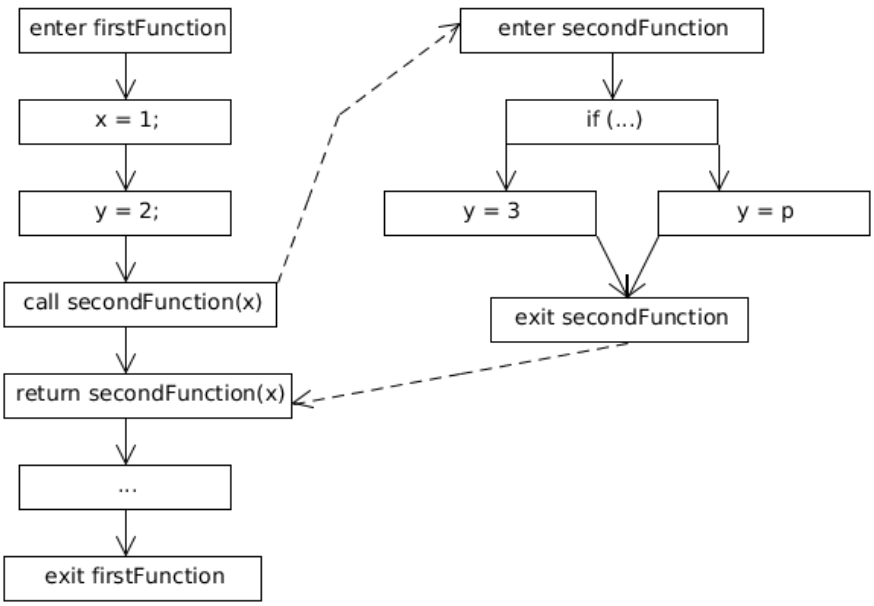

Abbildung 2.5: ICFG des Programms X. **Quelle:** In Anlehnung an [**Datta2010**].

Bei jedem Funktionsaufruf wird eine JOIN-Operation ausgeführt, welche die möglichen Zustände repräsentiert. Nach [**Datta2010**] können alle Pfade innerhalb eines ICFG wie folgt bestimmt werden. Dazu werden zunächst die Definitionen eines Pfades und des JOP benötigt.

Definition 2.3 *Sei ein Pfad p = [e_1,..,e_k] eine Hintereinanderreihung von k Kanten $e_i \in E$ eines ICFG, wobei der Endknoten der Kante e_i der Startknoten der Kante e_{i+1} ist, die Bedingung $k >= 1$ erfüllt und ein Pfad von Knoten v nach v die Länge 0 hat, dann lässt sich mithilfe einer Zuweisung $M(e) \in V \rightarrow V$, einer Datenflussfunktion zu jedem verbundenen Knotenpaar, eine* ***Pfadfunktion*** *bestimmen:*

$pf_q = M(e_k) \circ ... \circ M(e_1)$.

Diese Funktion pf_q ist für den Pfad der Länge $k = 0$ eine leere Hintereinanderreihung [] (die Identitätsfunktion am Knoten v).

Definition 2.4 *Join-Over-All-Paths (JOP). Für einen Startknoten $v_0 \in V$ zu einem Zielknoten n seien alle Pfade definiert durch:*

$JOP_n = \cup_{q \in Pfade(enter,n)} pf_q(v_0)$

Mithilfe dieser Definitionen lässt sich zur Laufzeit des Programmes bestimmen, welche Zustände (Pfade von einem zu einem anderen Knoten) im Speicher vorhanden sein können.
Es bleiben allerdings die folgende Fragen offen, welche für die weitere Vorgehensweise zur Sicherung des Wissens über den Speicher unverzichtbar sind:

1. Ist die Bestimmung aller Pfade mithilfe der JOP Definition effizient oder existieren Verbesserungen?

2. Lässt sich ein System oder endlicher Automat bestimmen, der durch Kenntnis eines ICFG die JOP Menge extrahieren kann und somit alle Speicherzustände kennt?

3. Existieren Wege in einem ICFG, die wichtiger sind als andere?

4. Existieren andere Wege Wissen zu repräsentieren?

3 Lösungsansätze

3.1 Automatentheorie (WPDS)

WPDS sind *weighted pushdown systems*, welche mithilfe eines Algorithmus' in ω-Automaten umgewandelt werden. Sie repräsentieren das Verhalten und Wissen, welches der Speicher (und somit mögliche Angreifer) haben können.

3.1.1 Pfadverbesserungen

Zur effizienteren Gestaltung dieser Systeme wird in diesem Abschnitt der in Kapitel 2 genannte Begriff JOP erweitert. Die Idee hinter dieser effizienteren Variante bildet die Möglichkeit, dass in einem Programm nicht alle Pfade wirklich ausgeführt werden. Im Folgenden sollen alle *call-enter* Kanten mit „(i“ und alle *exit-return* Kanten mit „i)“ notiert werden, wobei i maximal die Anzahl an Aufrufen (calls) repräsentiert. So lässt sich mit nachfolgender kontextfreien Grammatik durch Hintereinanderverbindung der Pfade, herausfinden, ob der Pfad gültig ist [**Datta2010**].

valid $\rightarrow$ matched valid
| (i valid
| ϵ

matched $\rightarrow$ matched matched
| (i matched i)
| Kantenname
| ϵ

Ziel ist es, die Aufrufe innerhalb eines Programmes im ICFG und in den Pfaden zu berücksichtigen. Der JOIN über diese Pfade wird nach [**Datta2010**] wie folgt erweitert, um die oben beschriebene Verbesserung zu übernehmen.

Definition 3.1 *Join-Over-All-Valid-Paths (JOVP). Für einen Startknoten $v_0 \in V$ zu einem Zielknoten n, seien alle Pfade definiert durch:*

$JOVP_n = \cup_{q \in GueltigePfade(enter,n)} pf_q(v_0)$

3.1.2 (W)PDS-Erstellung

WPDS sind automatenähnliche, gewichtete Systeme, die alle möglichen Zustände innerhalb eines Programmes, der in Form eines ICFG gegeben ist, kennen. Nachfolgend

sei nach [**Datta2010**] ein WPDS und die dazugehörigen Konfigurationen, welche die Zustandsänderungen bzw. Transitionen des Systems theoretisieren, definiert. Die Konfigurationen werden für die weitere Erstellung der Automaten benötigt, da diese die akzeptierenden Zustände des Systems über Pfade definieren.

Definition 3.2 *Weighted Pushdown System (WPDS). Ein WPDS ist ein Tripel W = (P, S, f), wobei P = (Q, Γ, Δ) ein Pushdown System ist und Q eine endliche Menge an Zuständen (Kontrollzustände),* Γ *eine endliche Menge an Kellersymbolen (Kelleralphabet) und* Δ*, mit* $\Delta \subseteq P \times \Gamma \times Q \times \Gamma^*$*, eine endliche Menge an Regeln ist. Eine Regel wird definiert als* $r \in \Delta$*, geschrieben* $(q, \gamma) \rightarrow (q', u)$*, wobei* $q, q' \in Q, \gamma \in \Gamma$ *und* $u \in \Gamma^*$*. Die Funktion f bildet ein Gewicht auf jede Regel aus P ab. S = (d ∈ {Gewichte}, ⊕, ⊗, 0', 1'), ist ein idempotenter (mit sich selbst verknüpfter) Halbring, welcher unterschiedliche Datenabstraktionen darstellt.*

Definition 3.3 *WPDS Konfiguration. Eine Konfiguration eines WPDS ist ein Paar* (p, u)*, wobei* $p \in P$ *und* $u \in \Gamma^*$*.*

Die Abbildung 2.5 zeigt einen ICFG, der nachfolgend in ein *Pushdown System* (PDS) umgewandelt wird. Um den Graphen zu kodieren, seien folgende Kodierungsregeln angegeben (nach [**Datta2010**]).

Regel	ICFG Modell
$(p, u) \rightarrow (p, v)$	interprozedurale Kante $u \rightarrow v$
$(p, c) \rightarrow (p, e_f r)$	ruft f von c auf und geht zu r
$(p, x_f) \rightarrow (p, \epsilon)$	return von f zu exit Knoten x_f

Abbildung 3.1: Kodierung von ICFGs. **Quelle:** In Anlehnung an [**Datta2010**].

Der dazugehörige PDS sieht nach Anwendung o.g. Kodierungsvorschriften wie folgt aus:

(p, enter FirstFunc) → (p, x=1)
(p, x=1) → (p, y=2)
(p, call SecFunc) → (p, e_f return SecFunc)
(p, ...) → (p, exit FirstFunc)
(p, exit FirstFunc) → (p, ϵ)
(p, enter SecFunc) → (p, if (...))
(p, if(...)) → (p, y=3)
(p, y=3) → (p, exit SecFunc)
(p, if (...)) → (p, y=p)
(p, y=p) → (p, exit SecFunc)
(p, exit SecFunc) → (p, ϵ)

Abbildung 3.2: PDS extrahiert aus ICFG. **Quelle:** In Anlehnung an [**Datta2010**].

Wenn die Kanten im ICFG ein Gewicht hätten, könnte man daraus nach denselben Regeln einen WPDS konstruieren, vorausgesetzt es existiert eine Domäne,

ein Halbring für den WPDS. In der Tabelle 3.2 gibt es eine Domäne. Es ist die Boolean-Domäne und soll nun weiter beschrieben werden. Weitere Domänen werden in [**Datta2010**] thematisiert.

Der in der obigen Definition genannte Halbring ist ({true, false}, $\wedge$, $\vee$, F, T), d.h. jede Regel ist entweder vorhanden (*true*) oder nicht (*false*). *false* wertet mit der Konjunktion $\wedge$ verknüpfte Elemente aus den Gewichten {true, false} wiederum zu *false* um, da die Konjunktion nur wahr ist, wenn beide Wahrheitsbelegungen *true* sind. Für alle Regeln r $\in \Delta$ ist f(r) = true.

Die zu suchende Menge JOVP für ein Programm P unter einer Boolean-Domäne definiert sich durch den Weg, welcher über die Kanten bestritten wird. Seien c und c' zwei Konfigurationen. Kommt auf dem Pfad von c zu c' ein σ vor, so ist der JOVP Wert für Konfigurationsmengen S, T $\subseteq P \times \Gamma^*$ nach [**Datta2010**] definiert mit: JOVP(S, T) = $\oplus\{s \Rightarrow^{\sigma}, s \in S, t \in T\}$.

3.1.3 ω-Automatenkonstruktion

Aus einem ICFG kann ein WPDS konstruiert werden, wobei dazu die Definition des *Join-Over-All-Valid-Paths* und die Kodierungstabelle aus Abbildung 3.1 verwendet werden.
Im Folgenden sei nun beispielhaft ein WPDS gegeben (in Anlehnung an [**Reps2005**]), um weitere Eigenschaften zu zeigen. Für alle Regeln gilt, dass das Gewicht = *true* ist, da eine Boolean-Domäne verwendet wird.

Q = {p,q}, Γ = {a,b,c,d}
$r_1 = (p, a) \rightarrow (q, b)$
$r_2 = (p, a) \rightarrow (p, c)$
$r_3 = (q, b) \rightarrow (p, d)$
$r_4 = (p, c) \rightarrow (p, ad)$
$r_5 = (p, d) \rightarrow (p, \epsilon)$

Aus diesem WPDS wird als nächstes ein ω-Automat konstruiert, der genau in diesem Fall eine Konfiguration c = (p, u) mit Gewicht w akzeptiert, wenn w die Kombination von allen akzeptierenden Pfaden von p nach u ist. Wenn dieser ω-Automat eine Konfiguration akzeptiert, so lassen sich auch Automaten konstruieren, die $pre^*(C)$ (alle Konfigurationen und jede Konfiguration zu sich selbst, in Vorwärtsrichtung) und $post^*(C)$ (alle Konfigurationen und jede Konfiguration zu sich selbst, in Rückwärtsrichtung) konstruieren, wobei C alle möglichen Konfigurationen eines Programmes beinhaltet.

Nach [**Datta2010**] geschieht dies in den folgenden Schritten für den **pre*-Algorithmus**:

1. Für jedes Paar (p', γ'), sodass der PDS minimal eine Regel der Form (p, γ) $\rightarrow$ (p', $\gamma'\gamma''$) enhält, füge einen neuen Zustand $q_{p',\gamma'}$ hinzu.

2. Wenn eine Regel der Form (p, γ) $\rightarrow$ (p', ϵ) existiert und γ einen (auch ϵ) Übergang zu q beschreibt, dann füge eine Transition (p', ϵ, q) hinzu.

3. Wenn eine Regel der Form (p, γ) $\rightarrow$ (p', γ') existiert und γ einen (auch ϵ) Übergang zu q beschreibt, dann füge eine Transition (p', γ', q) hinzu.

4. Wenn eine Regel der Form (p, γ) $\rightarrow$ (p', γ'γ") existiert und γ einen (auch ϵ) Übergang zu q beschreibt, dann füge eine Transition (p', γ', $q_{p',\gamma'}$) und ($q_{p',\gamma'}$, γ'',q) hinzu.

So ergibt sich für das Beispiel der folgende Automat:

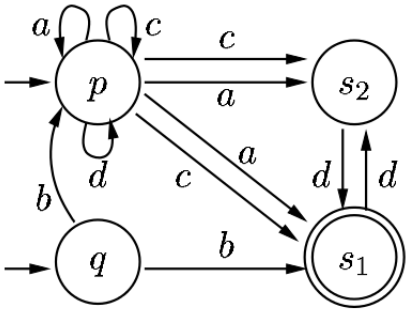

Abbildung 3.3: Automat pre* des Beispiels. **Quelle:** [**Reps2005**].

3.2 klassische Logik (Datalog)

Wissen kann auf mehrere Arten repräsentiert werden. Bei den bereits vorgestellten ω-Automaten kennt der Automat zu jedem Zeitpunkt alle möglichen Speicherzustände und weiß, welcher Pfad den meisten Einfluss auf eine Veränderung der Zustände hat.
Dieser Abschnitt widmet sich einer völlig unterschiedlichen Repräsentation solchen Wissens zu: der logischen Programmierung, namentlich Datalog.

Datalog ist eine *First-Order Logic* (FOL), die vor allem in der Informationssicherheit benutzt wird, um Regeln für Sicherheitszugriffe zu kontrollieren und durchzusetzen. Datalog ist eine beschränkte Variante von Prolog [**Datta2010**].
In Anlehnung an [**Schoening2000**] wird die Syntax von Datalog ähnlich zu Prolog definiert. Eine Ausnahme bildet hier *:-*, welches mit einem $\leftarrow$ ersetzt wird.

Exemplarisch sei P ein Datalogprogramm in äquivalenter Schreibweise zu Prolog.

```
1    connected(pete, oscar).
2    connected(oscar, mary).
3    connected(X, Y) <- connected(X, Y), connected(Y, Z).
```

Abbildung 3.4: Datalogprogramm P. **Quelle:** Eigene Darstellung.

Wie in in Abbildung 3.4 zu sehen, ähnelt die Syntax und Semantik Prologprogrammen. Um auf weitere Eigenschaften einzugehen, seien nun nach [**Datta2010**] folgende Begriffe definiert:

1. die linke Seite (vor $\leftarrow$) nennt sich *head*
2. die rechte Seite (nach $\leftarrow$) nennt sich body
3. eine *Regel* r ist die Kombination von head X und body Y der Form $X \leftarrow Y$. Falls body(r) leer ist, ist head(r) ein Fakt und demnach immer mit wahr belegt.

Eine Anfrage F an das Programm P, wie z.B. F = $\exists x_1, ..x_n(connected(X, mary) \wedge X \neq oscar)$ (gibt es jemanden, der mit *mary* verbunden ist und nicht *oscar* heißt?) kann mit wahr beantwortet werden, wenn F in jedem Modell von P den Wahrheitswert *wahr* annimmt, d.h. wenn P $\vDash$ F gilt, also F aus P geschlussfolgert werden kann.

Offensichtlich ist die Anfrage F wahr, da F logisch aus P geschlussfolgert werden kann und direkte Konsequenz aus P ist. Es gilt mit Anwendung der Regel r (Zeile 3) auf die Fakten (Zeile 1 und 2): connected(pete, oscar), connected(oscar, mary), demnach auch **connected(pete, mary)**.

Es existieren zwei Lösungswege, um bei komplexeren Programmen auf die Anfrage F reagieren zu können:

a) Fixpunktberechnung mit Herbranduniversen (bottom-up)

b) Resolution (top-down)

3.2.1 Fixpunktberechnung

Mithilfe von minimalen *Herbrandmodellen* über P kann die Anfrage F beantwortet werden [**Datta2010**].
Sei A ein Eingabealphabet, welches mindestens eine Konstante enthält, dann ist ein(e) Literal, Fakt, Regel oder Klausel ein(e) *Grundliteral, Grundfakt, Grundregel oder Grundklausel*, wenn es keine Variablen enthält.
Ein *Herbranduniversum* ist eine Menge U_A, welche nur aus gegrundeten Objekten konstruiert wird und alle Prädikate (grundatomaren Formeln), welche mit einer Konstantenkombination aus A belegt sind, bilden die *Herbrandbasis* H_B.

Im oben genannten Beispiel zu P existieren folgende Herbrandobjekte (Prädikat *connected*/2 mit „c“ abgekürzt):

U_A = {pete, oscar, mary}
H_B= {c(pete, oscar), c(oscar, pete), c(pete, mary), c(mary, pete, c(pete, oscar), c(oscar, mary), c(mary, oscar)}

Wie zu sehen, wurden alle Konstanten $c \in U_A$ in die Prädikate des Programmes

P in allen möglichen Kombinationen eingesetzt. Ein *Herbrandmodell* ist eine Teilmenge von H_B. Das minimale Herbrandmodell ist das, was durch das Programm mithilfe von Fakten und Regelanwendung bestimmt wird.
Mit dem Konsequenzoperator T_p kann nun ein Fixpunkt, also die Frage, ob P $\models$ F gilt, beantwortet werden.
Sei nach [**Datta2010**] der kleinste Fixpunkt wie folgt definiert:

$T_P \uparrow^\omega = \cup_{i=0}^{\infty}(T_P \uparrow^i)$, wobei $T_P \uparrow^0 =$,$T_P \uparrow^{i+1} = T_P(T_P \uparrow^i)$ und $i \geq 0$

$T_P \uparrow^\omega$ ist demnach monoton. Die Menge beinhaltet alle Ergebnisse aus vorherigen Iterationen. Die Iterationen über dem Programm P zur Fixpunktberechnung enthält folgende Schritte:

1. {c(pete, oscar), c(oscar, mary)}
2. {c(pete, oscar), c(oscar, mary), c(pete, mary)}

In Schritt 2 wird deutlich, dass die Fixpunktberechnung bereits die Antwort enthält: **connected(pete, mary)**.

3.2.2 Resolution

In [**Schoening2000**] wird beschrieben, wie eine SLD-Resolution in der klassischen Logik funktioniert. Auch das Beantworten der Anfrage F ist ähnlich. Falls $P \cup \neg F$ *kein* Modell hat, ist die Anfrage wahr. Für das oben genannte Beispiel gilt:

F $= \exists x_1, ..x_n(connected(X, mary))$

$\neg F = \forall x_1, ..x_n(\neg connected(X, mary))$

Bereits aus der Generierung von $\neg F$ wird ein direkter Widerspruch sichtbar. $\neg F$ würde wegen $\neg connected(X, mary)$ bedeuten, dass niemand *mary* kennen darf, da $\neg F$ mit dem Allquantor quantifiziert ist. Das widerspricht dem Fakt connected(oscar, mary) $\in P$. Die Notwendigkeit der Findung eines Modells für $P \cup \neg F$ ist in diesem Beispiel nicht vorhanden und die Resolution würde direkt abbrechen. Nicht in allen logischen Datalogprogrammen ergibt sich eine derart simple Konsequenz aus der Bildung der für die Resolution wichtigen Mengen, da Datalogprogramme normalerweise weitaus komplexer sind.

Zusammenfassend ist Datalog eine zu Prolog sehr ähnliche, aber dennoch untergeordnete logische Programmierspache, die sowohl bottom-up als auch topdown *immer terminierend ausgewertet werden kann*. Datalog bildet den Kern der logischen Programmiersprache. Prolog hingegen ist durch programmiertechnische Erweiterungen demnach mächtiger und benutzt Datalog als logische Kernkomponente.

4 Zusammenfassung & Ausblick

In der Gesamtheit betrachtet bieten Weighted Pushdown Systems eine effiziente Möglichkeit den Speicher, welcher durch die Ausführung eines Programmes belegt wird, zu analysieren und unzulässige Zustände bzw. Zustandsveränderungen zu entdecken. Gerade durch diese vorgestellte Erweiterung mithilfe von Gewichtungen und den unterschiedlichen Domänen, je nach Datentyp, existiert zu jedem Programm eine sichere Datenflussanalyse. Die in der Arbeit genannten Algorithmen („pre- und poststar“) stellen automatisierte Möglichkeiten dar, endliche Automaten zur Erfüllung dieser Aufgabe zu generieren. Sie sind intelligent und berücksichtigen durch die oben genannten Erweiterungen der PDS den Einfluss eines Pfades auf den gesamten Datenfluss.
Auch Datalog als Einschränkung von Prolog zur Repräsentation von Wissen (z.B. in Datenbanken) findet Anwendung in vielen Gebieten. Was kann das Programm an einer Stelle wissen und womöglich schlussfolgern? Was kann demnach ein potenzieller Angreifer aus eben dem Wissen (dem Speicher) herausfiltern und mutwillig gegen die Software benutzen?

Mit diesem Sicherheitsverständnis rund um ein geschriebenes Programm könnten viele Sicherheitsfehler vermieden werden. Anwendung finden diese Konzepte z.B. beim Thema *„analyzing security policies“* [**Datta2010**] und sind nicht Gegenstand dieser Ausarbeitung.

Literatur

[Datta2010] Anupam Datta, Somesh Jha, Ninghui Li, David Melski, Thomas Reps: Analysis Techniques for Information Security. 2010.

[Schoening2000] Uwe Schöning: Logik für Informatiker (5. Ausgabe). 2000.

[Reps2005] Thomas Reps, Stefan Schwoon, Somesh Jha, David Melski: Weighted Pushdown Systems and their Application to Interprocedural Dataflow Analysis. 2005.