technische universität
dortmund

Seminararbeit

JIF

Andreas Schober
25. Juni 2012

Betreuer: Christian Wessel

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

Andreas Schober
andreas.schober@udo.edu.de
Matrikelnummer: 126594
Studiengang: Master Informatik

Ausgewählte Themen des Modellbasierten Sicherheits-Engineerings
Thema: JIF

Eingereicht: 25. Juni 2012

Betreuer: Christian Wessel

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 25. Juni 2012

Andreas Schober

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation und Hintergrund

In der heutigen Zeit werden immer größere Ansprüche an Informations- und Datenverarbeitungssysteme gestellt. Aufgrund des hohen Kostenfaktors, den das Testen eines Systems hinterlässt, und dem Zeitdruck, der hinter der Entwicklung eines neuen Systems entsteht, werden oft sicherheitstechnische Aspekte ignoriert und nur „simpel“ oder gar nicht behandelt.
Nehmen wir als Beispiel eine Smartphone Banking App. Mit dieser hat der Benutzer die Möglichkeit, seine vertraulichen Bankinformationen von dem Server seiner Bank, über das mobile Internet, auf sein Handy zu laden und sich diese dort anzuschauen. Hier müssen also sensible und vertrauliche Daten von A nach B transportiert werden. Jedoch reicht es nicht, diese Daten einfach nur zu übermitteln. Sie müssen auch gegen Angreifer geschützt sein, damit sie nicht abgefangen und missbraucht werden können. Ein Angreifer könnte Lücken oder Fehler in diesen Informationsflüssen ausnutzen und die Logindaten des Bankkunden abgreifen und sich damit unbefugten Zugang zu seinem Konto verschaffen.
Ein weiteres Beispiel ist die Kommunikation zwischen zwei PCs in einem Intranet. Bei diesem könnte es sich um ein Firmenintranet handeln, über das vertrauliche Informationen (z.B. persönliche Angaben der Kunden) zwischen den PCs ausgetauscht werden. Auch hier müssen die Informationsflüsse abgesichert werden, da sonst, innerhalb der Firma, ein Dritter die Informationsflüsse abhören, die Daten abfangen und diese schlussendlich manipulieren könnte, obwohl er dazu keine Befugnisse hat.
Deshalb geht diese Ausarbeitung etwas genauer auf die Sicherheitsanforderungen an solche Datenübermittlungen ein und wie diese in der Praxis realisiert werden können. Daher soll der Leser lernen, wie er JIF für seine eigenen Projekte sinnvoll nutzen kann.

1.2 Aufbau der Arbeit

Zu Beginn dieser Arbeit wird der Begriff des Informationsfluss anhand eines Beispiels erläutert. Anschließend wird der Begriff des passiven und aktiven Angreifers eingeführt und erklärt, wie diese beiden unterschiedlichen Arten von Angreifern vorgehen und worauf es bei Sicherheitskonzepten gegen diese ankommt.
In Kapitel 2 wird die Java Erweiterung JIF vorgestellt. Zuerst wird darauf eingegangen, was JIF überhaupt ist und wo es Anwendung im Code findet. Im Anschluss werden die wesentlichen Features, wie Labels und Principals erläutert und anhand von kleinen Codebeispielen genauer beschrieben. Mit dieser Grundlage wird die Absicherung von Methoden und Klassen mit Hilfe von Labels erläutert. Zum Abschluss werden ein paar kurze Anwendungsgebiete beschrieben, bei denen JIF Verwendung findet.
Den Abschluss in Kapitel 3 bilden die Zusammenfassung sowie das Fazit des Autors.

2 Informationsfluss

In diesem Kapitel wird der Begriff des Informationsflusses eingeführt und erklärt. Anschließend geht der Text auf die Sicherheit in Informationsflüssen ein und mit welchen Arten von Angreifern zu rechnen ist. Zum Schluss wird der Spezialfall des impliziten Informationsflusses betrachtet.

2.1 Was ist ein Informationsfluss

Wie bereits in Kapitel 1 erwähnt, spielt der so genannte Informationsfluss in sicheren Systemen eine große Rolle. Jedoch wurde bisher nicht erwähnt, worum es sich hierbei handelt.

```
1 int x = 0;
2 int y = x;
```

Listing 2.1: ein einfacher Informationsfluss

In Listing 2.1 ist ein typisches Beispiel für einen Informationsfluss dargestellt. Dieses triviale Beispiel beschreibt den einfachen Transfer der Informationen von x zu y.
Da in größeren und sicherheitskritischen Projekten nicht nur solche einfachen Informationsflüsse mit vermeintlich unwichtigen Informationen verwendet werden, sondern auch sensible Daten, wie zum Beispiel die Kontodaten einer Bank, transferiert werden können, müssen auch Informationsflüsse abgesichert werden.

2.2 Sicherheit in Informationsflüssen

Gerade in Branchen und Firmen, die mit persönlichen und vertraulichen Daten arbeiten, ist es wichtig, dass diese Daten nicht angreifbar sind und somit kein Angreifer an diese Daten herankommt oder sie manipulieren kann.
Nach [**ZM01**] unterscheidet man bei Informationsflüssen zwischen einem passiven und aktiven Angreifer, welche beide im Folgenden etwas genauer beschrieben werden:

- der passive Angreifer:
  Der passive Angreifer beobachtet ein System während seiner Laufzeit. Ihm ist dabei klar, dass es Teile des Systems gibt, die er beobachten kann, und andere, die er nicht einsehen kann. Damit man ein System gegen einen passiven Angreifer absichern kann, versucht man sich in die Rolle des Angreifers hineinzuversetzen, um herauszufinden, an welche Informationen man durch das Beobachten eines Systems gelangen kann.

- der aktive Angreifer:
  Ein aktiver Angreifer hat im Gegensatz zu einem Passiven die Möglichkeit, Daten in einem System oder dessen Verhalten zu verändern anstatt nur das System zu beobachten (Weiterführung siehe Kapitel 2.2.1).

Im direkten Vergleich scheint es daher wichtiger zu sein, sich gegen aktive Angreifer zu schützen, da diese die Passiven implizieren. Durch die Modellierung von aktiven Angreifern soll es dem Entwickler möglich sein, das Systemverhalten bei einem (teilweise) kompromitiertem System vorherzusagen und die Informationsflüsse dementsprechend abzusichern.

2.2.1 Implizite Flüsse

Die so genannten impliziten Flüsse sind ein Spezialfall der Informationsflüsse. Ein Angreifer kann hierbei nicht direkt den Informationsfluss ausnutzen, um an interessante Daten heranzukommen. Viel mehr beobachtet er das Systemverhalten um Details über die Funktionsweise des Systems zu erhalten. Betrachten wir einmal das Listing 2.2:

```
1 int l;
2 boolean h = true;
3 if ( h == true ) then {
4   l := 3
5 }
6 else {
7   l := 42
8 }
```

Listing 2.2: ein impliziter Informationsfluss

Auf dem ersten Blick erkennt man keine Sicherheitslücke in diesem System. Betrachten wir nun den Fall, dass ein aktiver Angreifer das System beobachtet. Dieser wird feststellen, dass sich der Wert der Variable l nach mehreren Durchläufen des Systems unterscheiden kann. Nun könnte er Rückschlüsse auf den Wert, der in der Variable h gespeichert ist, ziehen. Aus diesem Grund muss bei der Sicherheit von Informationsflüssen auch speziell der Fall des impliziten Flusses betrachtet werden.

3 Java und Information Flow (JIF)

Dieses Kapitel soll dem Leser die Java Erweiterung JIF (Java und Information Flow) vorstellen. Nach einem allgemeinen Überblick werden dem Leser die einzelnen Aspekte von JIF genauer erläutert. So wird nach und nach die Verwendung von JIF dargestellt. Am Ende des Kapitels wird noch ein kleiner Einblick in zwei Anwendungsgebiet von JIF gegeben gegeben.

3.1 Was ist JIF

In dieser Ausarbeitung gehen wir speziell auf Informationsflüsse in der Programmiersprache Java ein. Bei der Verwendung von Java ohne besondere Bibliotheken, realisiert man die Sicherheit von Informationsflüssen meist durch geeignete Methoden und Abfragen. Da dies aber schnell kompliziert werden kann, stellen wir nun die Java Erweiterung JIF (Java and Information Flow) vor [**JIF**].
JIF ist eine sicherheitstypische Erweiterung für die Programmiersprache Java. Mit ihrer Hilfe kann man Informationsflüsse sicherer gestalten und die Integrität von Daten verbessern. Hierbei findet JIF eine Verwendung sowohl bei der Kompilierung, als auch während der Laufzeit. Man kann einen JIF in etwa mit einem erweiterten Type-Checker vergleichen, der in alle vorhandenen Klassen mit eingebunden werden kann.
Zusätzlich zu der JIF-Bibliothek existiert ein Plugin für die Entwicklungsumgebung Eclipse. Das Plugin soll dem Entwickler dabei helfen, Fehlernachrichten JIF zu analysieren und es erweitert die Outline von Eclipse um die verwendeten Labels oder es bietet einem die Möglichkeit sich die Richtlinien anzusehen.[**JIFC**]. Jifclipse wird in Kapitel 3.3 noch etwas genauer beschrieben.

3.2 Realisierung von Sicherheit mittels JIF

In diesem Unterkapitel werden wir einen ersten Eindruck vermitteln, wie JIF verwendet werden kann, um Informationsflüsse in Java-Code sicherer zu gestalten. Es werden die wichtigsten Features von JIF vorgestellt und welchen Zweck sie erfüllen.

3.2.1 Labels

Betrachten wir zu Beginn einmal den primitiven Informationsfluss aus Listing 3.1.

```
1 int x;
2 int y;
3 x = y;
4 y = x;
```

Listing 3.1: primitiver Informationsfluss nach [**JIF**]

In diesem Codebeispiel befinde sich keine sicherheitstechnische Maßnahme. Die Informationen werden einfach transportiert. An dieser Stelle können die so genannten Labels aus JIF Verwendung finden.

```
1 int {Alice->Bob} x;
2 int {Alice->Bob, Chuck} y;
3 x = y; // OK: policy on x is stronger
4 y = x; // BAD: policy on y is not as strong as x
```

Listing 3.2: sicherer Informationsfluss aus [**JIF**]

Das Listing 3.2 zeigt eine bespielhafte Verwendung von JIF. Die Ausdrücke in den geschweiften Klammern stellen hier eine so gennante Sicherheitsinformation dar. Durch diese „Annotation“ kann die Verwendung einer Information eingeschränkt werden. Die Notation „$Alice \rightarrow Bob$“ in der ersten Zeile sagt hierbei aus, dass die Information der Variable x von *Alice* verwaltet wird und *Alice* *Bob* die Erlaubnis erteilt, diese Information anzusehen. Analog gibt *Alice* die Variable y für *Bob* und *Chuck* zum Ansehen frei.
Der Code in Zeile drei ist immer noch sicher, da niemand neues die Werte der Variable y einsehen kann. Beim Komplilieren der vierten Zeile wird JIF jedoch einen Fehler ausgeben. Hier würde es *Chuck* über Umwege gelingen, den Wert der Variable x zu lesen, obwohl *Alice* diesen nur für *Bob* freigegeben hätte. Daher würde hier der Type-Checker einen Fehler signalisieren und die Sicherheit des Codes gewährleisten. Die zugrunde liegende Theorie wird in dem Kapitel 3.2.3 genauer erläutert.

3.2.2 Principals

Gehen wir nun etwas genauer auf unsere Darsteller *Alice*, *Bob* und *Chuck* aus den vorrausgegangenen Beispielen ein. Bei diesen handelt es sich um so genannten Principals, welche vergleichbar mit Entitäten sind. Der Entwickler kann mit diesen Benutzer, Rollen, Prozesse oder ähnliches modellieren und mit diesen Sicherheitsrichtlinien (siehe Kapitel 3.2.3) im Bezug auf Beobachtung und Veränderung des Systemverhaltens festlegen.
Dabei ist es möglich, dass man einen Principal q für einen Principal p handeln lässt. Dies wäre so, als wenn man einen eigenen Auftrag an jemand anderen weiterdelegiert. Man spricht davon, dass *„q für p handelt“*. Dies wird auch mit $q \geq p$ notiert. Da es sich hierbei um reflexive und transitive Beziehung handelt, kann man damit das Verhalten von Principals untereinander modellieren. Weiterhin bietet JIF die Möglichkeit Principals mittels Konjunktion und Disjunktion miteinander zu verbinden [**LABW91**].
Ein weiterer Vorteil von JIF ist, dass man einen Top-Principal $\top$ und einen Bottom-Principal $\bot$ zu definieren. Ein Top-Principal könnte zum Beispiel ein Gruppenführer sein, der im Namen aller Gruppenmitglieder handeln könnte. Dadurch lässt sich eine Art Rollenhierarchie innerhalb der Principals festlegen [**SAN96**].

3.2.3 Richtlinien

In den beiden vorherigen Kapiteln sind wir bereits auf die Grundlagen für die Benutzung von JIF eingegangen. Mit diesen können wir so genannte Richtlininen in unseren Code einpflegen um dessen Sicherheit zu erhöhen. Im Allgemeinen wird zwischen zwei Arten von Richtlinien unterschieden. Diese werden nun genauer vorgestellt.

Vertraulichkeitsrichtlinien

Mit einer Vertraulichkeitsrichtlinie kann ein Principal einem Anderen den Zugriff auf eine Information gestatten, damit dieser die Information lesen darf. Betrachten wir noch einmal das Beispiel aus Listing 3.2. *Alice* hat die Variable x für *Bob* zum Lesen freigegeben. Formal definieren wir so eine Richtlinie mit $o \rightarrow r$, wobei o für den Besitzer und r für den Leser steht. Hierbei muss jedoch auch die *„handelt für“* Beziehung mit berücksichtigt werden, bei der zum Beispiel ein Principal q für den Leser r handeln könnte. Daher wird nach [**JIF**] eine *readers(p, c)* - Funktion definiert, die allen zugelassenen Lesern einer Information des Principals p nach der Richtlinie c ausgibt. Die Funktion sieht dabei wie folgt aus:

$$readers(p,\ o \rightarrow r) \equiv \{q \mid if\ o \geq p\ then\ (q \geq o\ or\ q \geq r)\}$$

Durch diese Definition wird die Transitivität bei der Vertraulichkeit mit einbezogen. Ein Leser q, der für o oder r handelt, gilt als vertraulich und hat Zugriff auf die Daten, die o bereitstellt.
Wie bereits im Abschnitt 3.2.2 erwähnt, kann man Principals auch mittels Konjunk-

tion und Disjunktion verbinden. Daher wird der Operator $\sqcup$ für die Konjunktion von Vertraulichkeitsrichtlinien und $\sqcap$ für die Disjunktion verwendet. Eine Richtlinie der Form $c \sqcup d$ besagt, dass ein Principal eine Information nur lesen darf, wenn die Principals c und d es erlauben. Die Richtinline $c \sqcap d$ hingegen, besagt, dass entweder c oder d die Erlaubins zum Lesen der Information gegeben muss. Folglich wird die Funktion *readers(p, c)* erweitert:

$$readers(p, c \sqcup d) \equiv readers(p, c) \cap readers(p, d)$$
$$readers(p, c \sqcap d) \equiv readers(p, c) \cup readers(p, d)$$

Mit Hilfe der readers-Funktion lässt sich eine *„beschränkt nicht mehr als“* Relation $\sqsubseteq$c zwischen Richtlinien definieren. Dabei sagt man nach [**JIF**], dass für zwei Sicherheitsrichtlinien m und n der Ausdruck $m \sqsubseteq c\ n$ gilt, wenn für alle Principals p $readers(p, m) \supseteq readers(p, n)$ gilt. Formal bedeutet dies, dass p glaubt, dass m genauso vielen Lesern Zugriff gestattet, wie n.

Integritätsrichtlinien

Nach [**JIF**] werden Vertraulichkeit und Integrität oft dual betrachtet. Dies ist auch bei Richtlinien der Fall. So gibt es neben der Leser Richtlinie auch eine Richtlinie zum Schreiben. Wir sagen, dass bei einer Regel $o \leftarrow w$ der Principal o dem Principal w erlaubt, die gegebenen Informationen zu verändern. Aufgrund der Dualität definieren wird die Funktion *writers(p, c)* wie folgt definiert:

$$writers(p, o \leftarrow w) \equiv \{q \mid \textit{if } o \geq p \textit{ then } (q \geq o \textit{ or } q \geq w)\}$$
$$writers(p, c \sqcap d) \equiv writers(p, c) \cap writers(p, d)$$
$$writers(p, c \sqcup d) \equiv writers(p, c) \cup writers(p, d)$$

Wie man bereits an der Definition der *writers*-Funktion sehen kann, ist diese analog zu der *readers*-Funktion definiert und besitzt die gleichen Regeln unter Konjunktion und Disjunktion.
Die Dualität lässt sich hierbei auch auf die *„beschränkt nicht mehr als“* Relation anwenden und daher gilt für die *writers*-Funktion: für zwei Principals c und d gilt $c \sqsubseteq i\ d$, wenn für alle p $writers(p, c) \subseteq writers(p, d)$ gilt.

3.2.4 Methoden und Klassen

```
1  int {Alice->Bob} x;
2  public class Vector[label L] extends
3  AbstractList[L] { private int{L} length;
4      private Object{L}[]{L} elements;
5
6      public Vector() ...
7      public Object elementAt(int i):{L; i}
8                      throws IndexOutOfBoundsException {
9          ...
10         return elements[i];
11     }
12     public void setElementAt{L}(Object{L} o, int{L} i) ...
13     public int{L} size() { return length; }
14     public void clear{L}() ...
15     ...
16 }
```

Listing 3.3: JIF Version der Java Vector Klasse [**JIF**]

Betrachten wir nun, wie Labels und Principals verwendet werden können. Hierzu werfen wir einen Blick auf das Listing 3.3. Die Klasse wurde so gestaltet, dass sie mit unterschiedlichen Labeln erzeugt werden kann. Sie ist also generisch.
Interessanter ist die Annotation an der Methode setElementAt(...). Die Beschränkung auf das Label L hat die Folgen, dass zum Einen die Methode nur von Callern aufgerufen werden kann, die mindestens die Rechte haben, um Informationen mit dem Label L zu lesen (siehe Kapitel 3.2.2). Zum Anderen sorgt sie dafür, dass die Methode nur Informationen verändern darf, die maximal mit dem Label L abgesichert sind. Dieses Label wird im Allgemeinen als Anfangslabel bezeichnet.
Weiterhin besteht die Möglichkeit, die Herkunft der Parameter mittels Labels zu schützen. Man kann zum Beispiel als Parameter nur Objekte zulassen, die von bestimmten Principals verändert werden dürfen.
Analog zum Anfangslabel gibt es noch das End- und das Rückgabelabel. Das Endlabel hilft dabei, eine korrekte Terminierung der Methode zu überprüfen. Als Beispiel wird die elementAt(...) betrachtet. Mit dem Endlabel $L; i$ ist es möglich zu steuern, wer ein Recht darauf hat, die IndexOutOfBoundsException zu „beobachten“.
Rückgabelabels lassen sich in einen ähnlichen Kontext einbetten. Hier kann man festlegen, wer die Rückgabewerte einer Methode lesen beziehungsweise verändern darf.

3.3 Anwendungsgebiete

In diesem Kapitel werden kurz und knapp ein paar weiterführende Arbeiten zum Thema Java und Informationsflüsse vorgestellt.

- Jifclipse (nach [**JIFC**]):
  Bei Jifclipse handelt es sich um eine Erweiterung für die Entwicklungsumgebung Eclipse. Es soll dem Entwickler beim Verwenden von JIF in seinem Code unterstützen.
  Jifclipse soll dem Entwickler im Wesentlichen bei den folgenden drei Aufgaben helfen:

  – Die Bestimmung von Principals und deren Interaktionen mit den Informationsflüssen
  – Sinnvolle Darstellung von Labels und Richtlinien
  – Unterstützung bei der Behebung von Fehlern in Informationsflüssen

  Im Fall von Jifclipse wurde zum Beispiel die so genannte Outline von Eclipse, welche den Aufbau einer Java-Klasse darstellt, noch um die Darstellung von Labels und Principals erweitert. Des Weiteren macht die Erweiterung Vorschläge bei falschen Labels und markiert diese im Programmcode. Genauer wird Jifclipse in [**JIFC**] beschrieben.

- JPmail (nach [**JPMail**]):
  Bei JPmail handelt es sich um den Versuch, eine Anwendung komplett mit der Unterstützung von JIF [**JIF**] zu entwickeln. Die Idee ist nach der Publizierung von Sicherheitslücken in US Insituten entstanden, da dort vertrauliche Dokumente per Mail versendet und dann abgefangen und veröffentlicht worden sind. Das Ziel von JPmail ist es dem Nutzen einen sicheren EMail Client zur Verfügung zu stellen, der ihm einen sicheren Transport von Daten gewährleistet [**JPMail**]. Dies soll durch das Markieren der Daten mit so genannten Sicherheitsleveln geschehen. Dadurch sollen zum Beispiel „Daten mit hoher Sicherheit“ nicht mit „Daten mit niedriger Sicherheit“ interagieren dürfen. Hier findet das Label-System von JIF Verwendung.
  Da der Nachrichtentransfer über einen Server abläuft, müssen eventuelle Sicherheitsrichtlinien abgeschwächt werden, damit der Server die Nachricht verarbeiten kann. Die genaue Funktionsweise von JPmail ist in [**JPMail**] zu finden.

4 Zusammenfassung und Fazit

4.1 Zusammenfassung

Ein normaler Informationsfluss kann als nicht sicher bezeichnet werden, da passive und aktive Angreifer eine Bedrohung für ihn, durch das Beobachten und Verändern der Informationen, darstellen. Außerdem stellen implizite Flüsse eine Sicherheitsbedrohung dar, die meist gar nicht als solche wahrgenommen wird.
Die Java Erweiterung JIF greift an dieser Stelle ein. Durch so genannte Labels kann der Entwickler die Vertraulichkeit seiner Informationen einschränken. Hierbei wird eine Notation verwendet, die man mit Annotationen vergleichen kann.
Ergänzend zu den Labels gibt es die Principals. Sie nehmen Rollen oder Sicherheitsstufen bei der Modellierung ein. Mit ihnen könnte eine Hierarchie einer Firma nachgebildet werden. Dies ist möglich, da ein Principal auch für einen anderen handeln kann. Dadurch lässt sich eine transitive und reflexive Relation zwischen allen Principals aufbauen.
Durch das Zusammenspiel beider Komponenten ist es dem Entwickler möglich zu kontrollieren, wer Zugriff auf seine bereitgestellten Informationen hat oder wer diese verändern darf. Dies geschieht durch Richtlinien. Wir unterscheiden dabei einmal zwischen Vertraulichkeitsrichtlinien, mit deren Hilfe der Entwickler definieren kann, wer wem den Zugriff auf Informationen gestattet und Integritätsrichtlinien, um den Kreis der Principals, die bestimmte Informationen verändern dürfen, einzuschränken.
Methoden bieten die Möglichkeit mit Anfangs-, End-, Parameter- und Rückgabelabels versehen zu werden. Ein Anfangslabel beschränkt den Kreis der Principals, die diese Methode verwenden dürfen. Mit dem Endlabel kann man Richtlinien einführen, wer zum Beispiel Exceptions, die geworfen worden sind, sehen darf. Die Rückgabe- und Parameterlabels bieten noch weitere Möglichkeiten die Methode abzusichern.
Weiterhin können Klassen mit Labels versehen werden, um direkt bei ihrer Erzeugung, Richtlinien zu übergeben und bereits vorhandene Richtlinien weiterzugeben.

4.2 Fazit

Die Bedrohung, die durch Informationsflüsse oder implizite Flüsse ausgeht wird von vielen Entwicklern unterschätzt. An dieser Stelle sind Erweiterungen wie zum Beispiel JIF eine gute Ergänzung um dem Entwickler die Sicherheit seines Codes zu gewährleisten.
Durch die Ähnlichkeit zu den regulären Annotationen schafft JIF es den Einstieg einfach zu machen. Schon durch die einfache Benutzung von Labels und Principals kann man seine Informationsflüsse sicherer gestalten. Mit Hilfe der transitiven und reflexiven Eigenschaft von Principals kann man zum Beispiel einfach und leicht ein vorhandenes Modell (von einem Geschäftsprozess o.Ä.) nachbilden und die Sicherheitsrichtlinien dazu aufstellen.
Die vierfach mögliche Absicherung von Methoden sowie die Beschriftung von Klassen mittels Labels tragen ebenso dazu bei.
Ein Nachteil ist der unübersichtlich und abschreckend wirkende Code, wenn man ihn zum ersten Mal sieht. Eine voll beschriftete Methode, mit Anfangs-, End-, Rückgabe- und Paramterlabeln wirkt schnell überladen und kompliziert. Schaut man sich jedoch ein einfaches Einsteigsbeispiel an, kann man sich schnell in die Materie einarbeiten.
Für Programmcodes, bei denen es um vertrauliche Informationen und sensible Daten geht, ist JIF durchaus eine Bereicherung. Daher könnte ich mir vorstellen, bei sicherheitskritschen Anwendungen, an deren Entwicklung ich beteiligt bin, JIF zum Absichern der Informationsflüsse zu verwenden.

Literatur

[JIF] Danfeng Zhang, Owen Arden, K. Vikram, Stephen Chong, Andrew Myers: *http://www.cs.cornell.edu/jif/* , zuletzt gesehen am 25.06.2012 um 11:44 Uhr

[JIFC] Boniface Hicks, Dave King, Patrick McDaniel: Jifclipse: *Development Tools for Security-Typed Languages*. 2nd Workshop on Programming languages and Analysis for Security (PLAS), pages $1 - 10$, June 2007

[JPMail] Patrick McDaniel, Boniface Hicks, Dave King, Kiyan Ahmadizadehs: *http://siis.cse.psu.edu/jpmail/index.html* , zuletzt gesehen am 25.06.2012 um 11:44 Uhr

[ZM01] Steve Zdancewic, Andrew C. Myers: *Robust Declassification*. Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW), June 2001

[LABW91] Butler Lampson, Martin Abadi, Michael Burrows, Edward Wobber: *Authentication in Distributed Systems: Theory and Practice*. Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP), October 1991

[SAN96] Ravi S. Sandhu: *Role Hierarchies and Constraints for Lattice-based Access Controls*. Proceedings of the 4th European Symposium on Research in Computer Security, Septemeber 1996