# Theorembeweiser und ihre Anwendungen

Prof. Dr.-Ing. Gregor Snelting
Dipl.-Inf. Univ. Daniel Wasserrab

Lehrstuhl Programmierparadigmen
IPD Snelting
Universität Karlsruhe (TH)

Teil VI

# Typbasierte Information Flow Control

# Sprachbasierte Softwaresicherheit

*Language-based security* (LBS)

- Ziele:
  - Garantie von Sicherheitseigenschaften und
  - Auffinden von Sicherheitsverletzungen
- betrachtet Programmquelltexte oder Objektcodes
- verwendet Techniken aus
  - formaler Semantik
  - Typsystemen
  - Programmanalyse

# Informationsflusskontrolle

*Information flow control* (IFC)

- wichtigstes Teilgebiet der LBS
- Ziel: Garantie von zwei Zielen auf Daten

  Vertraulichkeit: vertrauliche Information (z.B. systeminterne Daten) darf nicht nach außen (z.B. Internet) gelangen

  Integrität: kritische Berechnungen dürfen nicht von außen beeinflusst werden
- zwei Haupttechniken:
  - Typsysteme
  - Programmanalyse

# Informationsflusskontrolle

*Information flow control* (IFC)

- wichtigstes Teilgebiet der LBS
- Ziel: Garantie von zwei Zielen auf Daten

  Vertraulichkeit: vertrauliche Information (z.B. systeminterne Daten) darf nicht nach außen (z.B. Internet) gelangen

  Integrität: kritische Berechnungen dürfen nicht von außen beeinflusst werden
- zwei Haupttechniken:

  Typsysteme

  Programmanalyse

# Informationsflusskontrolle

*Information flow control* (IFC)

- wichtigstes Teilgebiet der LBS
- Ziel: Garantie von zwei Zielen auf Daten

  Vertraulichkeit: vertrauliche Information (z.B. systeminterne Daten) darf nicht nach außen (z.B. Internet) gelangen

  Integrität: kritische Berechnungen dürfen nicht von außen beeinflusst werden
- zwei Haupttechniken:

  Typsysteme

  Programmanalyse

# IFC Typsysteme

- Variablen eingeteilt in Sicherheitstypen durch Typumgebung
- allgemein: Sicherheitstypen-Verband, meist aber nur Aufteilung in High und Low
- Vertraulichkeitsaussage als Typsicherheitsaussage:
  Werte in High-Variablen beeinflussen keine Werte von Low-Variablen
- statisch und effizient prüfbar
- jedoch nur konservative Approximation, u.a. da

| | |
|---|---|
| nicht flusssensitiv | d.h. beide Zweige eines `if` durchlaufen |
| nicht objektsensitiv | d.h. Felder unterschiedlicher Objekte als gleich betrachtet |
| nicht kontextsensitiv | d.h. verschiedene Aufrufstellen einer Methode nicht unterschieden |

# Das Volpano-Smith-Irvine Typsystem

D. Volpano, G. Smith, and C. Irvine.
A sound type system for secure flow analysis.
*Journal of Computer Security*, 4(2-3):167–187, IOS Press, 1996.
http://users.cs.fiu.edu/~smithg/papers/jcs96.pdf

# Zugrundeliegende While-Sprache

While-Sprache wie definiert in der “Semantik”-Vorlesung,
Small Step Semantik Regeln mit Zwischenstatements und -zuständen

```
inductive red :: "com * state ⇒ com * state ⇒ bool"
and red' :: "com ⇒ state ⇒ com ⇒ state ⇒ bool"
   ("((1⟨_,/_⟩) →/ (1⟨_,/_⟩))" [0,0,0,0] 81)
where "⟨c₁,σ₁⟩ → ⟨c₂,σ₂⟩ == red (c₁,σ₁) (c₂,σ₂)"

 | RedAssign: "⟨x:=e,σ⟩ → ⟨Skip,σ(x:=([[e]]σ))⟩"
 | SeqRed:    "⟨c₁,σ⟩ → ⟨c₁',σ'⟩ ⟹ ⟨c₁;;c₂,σ⟩ → ⟨c₁';;c₂,σ'⟩"
 | RedSeq:    "⟨Skip;;c₂,σ⟩ → ⟨c₂,σ⟩"
 | RedCondT:  "[[b]]σ = true ⟹ ⟨if (b) c₁ else c₂,σ⟩ → ⟨c₁,σ⟩"
 | RedCondF:  "[[b]]σ = false ⟹ ⟨if (b) c₁ else c₂,σ⟩ → ⟨c₂,σ⟩"
 | RedWhileT:
      "[[b]]σ = true ⟹ ⟨while (b) c,σ⟩ → ⟨c;;while (b) c,σ⟩"
 | RedWhileF: "[[b]]σ = false ⟹ ⟨while (b) c,σ⟩ → ⟨Skip,σ⟩"
```

# Zugrundeliegende While-Sprache

While-Sprache wie definiert in der "Semantik"-Vorlesung,
Small Step Semantik Regeln mit Zwischenstatements und -zuständen

**inductive** *red :: "com * state ⇒ com * state ⇒ bool"*
**and** *red' :: "com ⇒ state ⇒ com ⇒ state ⇒ bool"*
*("((1⟨_,/_⟩) →/ (1⟨_,/_⟩))" [0,0,0,0] 81)*
**where** *"⟨c₁,σ₁⟩ → ⟨c₂,σ₂⟩ == red (c₁,σ₁) (c₂,σ₂)"*

*| RedAssign: "⟨x:=e,σ⟩ → ⟨Skip,σ(x:=([[e]]σ))⟩"*
*| SeqRed: "⟨c₁,σ⟩ → ⟨c₁',σ'⟩ ⟹ ⟨c₁;;c₂,σ⟩ → ⟨c₁';;c₂,σ'⟩"*
*| RedSeq: "⟨Skip;;c₂,σ⟩ → ⟨c₂,σ⟩"*
*| RedCondT: "[[b]]σ = true ⟹ ⟨if (b) c₁ else c₂,σ⟩ → ⟨c₁,σ⟩"*
*| RedCondF: "[[b]]σ = false ⟹ ⟨if (b) c₁ else c₂,σ⟩ → ⟨c₂,σ⟩"*
*| RedWhileT:*
*"[[b]]σ = true ⟹ ⟨while (b) c,σ⟩ → ⟨c;;while (b) c,σ⟩"*
*| RedWhileF: "[[b]]σ = false ⟹ ⟨while (b) c,σ⟩ → ⟨Skip,σ⟩"*

# Sicherheitstypen von Ausdrücken

Sicherheitstypen als Datentyp mit zwei Elementen:
**datatype** *secType = Low | High*

Typumgebung weist Variablen initialen Sicherheitstyp zu

- Sicherheitstyp einer Variable ändert sich nie
- realisiert: partielle Funktion von Variablenname nach Sicherheitstyp
  - *None*: Wert undefiniert, *Some y*: Wert definiert, nämlich *y*
  - genannt Map, Symbol $\rightharpoonup$ statt $\Rightarrow$
  - **types** *typeEnv = "vname* $\rightharpoonup$ *secType"*

# Sicherheitstypen von Ausdrücken

jedem Ausdruck (Konstanten *Val*, Variablen *Var*, binäre Operation «*bop*») mittels Typumgebung Sicherheitstyp zuordnen, Syntax Γ ⊢ *e* : *T*

**inductive** *secExprTyping* ::
*"typeEnv ⇒ expr ⇒ secType ⇒ bool" ("_ ⊢ _ : _")*
**where** *typeVal: "Γ ⊢ Val V : T"*
— Konstanten haben beliebigen Sicherheitstypen

| *typeVar: "Γ Vn = Some T ⟹ Γ ⊢ Var Vn : T"*
— Variablen sehen Sicherheitstyp in Typumgebung nach

| *typeBinOp1: "⟦Γ ⊢ e1 : Low; Γ ⊢ e2 : Low⟧*
*⟹ Γ ⊢ e1 «bop» e2 : Low"*
| *typeBinOp2: "⟦Γ ⊢ e1 : High; Γ ⊢ e2 : T⟧*
*⟹ Γ ⊢ e1 «bop» e2 : High"*
| *typeBinOp3: "⟦Γ ⊢ e1 : T; Γ ⊢ e2 : High⟧*
*⟹ Γ ⊢ e1 «bop» e2 : High"*
— binäre Operation nur *Low*, wenn auch beide Teilausdrücke *Low*

# Sicherheitstypen von Ausdrücken

jedem Ausdruck (Konstanten *Val*, Variablen *Var*, binäre Operation «*bop*») mittels Typumgebung Sicherheitstyp zuordnen, Syntax Γ ⊢ *e* : *T*

**inductive** *secExprTyping* ::
*"typeEnv ⇒ expr ⇒ secType ⇒ bool" ("_ ⊢ _ : _")*
**where** *typeVal: "Γ ⊢ Val V : T"*
— Konstanten haben beliebigen Sicherheitstypen

*| typeVar: "Γ Vn = Some T ⟹ Γ ⊢ Var Vn : T"*
— Variablen sehen Sicherheitstyp in Typumgebung nach

*| typeBinOp1: "⟦Γ ⊢ e1 : Low; Γ ⊢ e2 : Low⟧*
*⟹ Γ ⊢ e1 «bop» e2 : Low"*
*| typeBinOp2: "⟦Γ ⊢ e1 : High; Γ ⊢ e2 : T⟧*
*⟹ Γ ⊢ e1 «bop» e2 : High"*
*| typeBinOp3: "⟦Γ ⊢ e1 : T; Γ ⊢ e2 : High⟧*
*⟹ Γ ⊢ e1 «bop» e2 : High"*
— binäre Operation nur *Low*, wenn auch beide Teilausdrücke *Low*

# Sicherheitstypen von Ausdrücken

jedem Ausdruck (Konstanten *Val*, Variablen *Var*, binäre Operation «*bop*») mittels Typumgebung Sicherheitstyp zuordnen, Syntax Γ ⊢ *e* : *T*

**inductive** *secExprTyping* ::
*"typeEnv ⇒ expr ⇒ secType ⇒ bool" ("_ ⊢ _ : _")*
**where** *typeVal: "Γ ⊢ Val V : T"*
— Konstanten haben beliebigen Sicherheitstypen

| *typeVar:* *"Γ Vn = Some T ⟹ Γ ⊢ Var Vn : T"*
— Variablen sehen Sicherheitstyp in Typumgebung nach

| *typeBinOp1: "⟦Γ ⊢ e1 : Low; Γ ⊢ e2 : Low⟧*
*⟹ Γ ⊢ e1 «bop» e2 : Low"*
| *typeBinOp2: "⟦Γ ⊢ e1 : High; Γ ⊢ e2 : T⟧*
*⟹ Γ ⊢ e1 «bop» e2 : High"*
| *typeBinOp3: "⟦Γ ⊢ e1 : T; Γ ⊢ e2 : High⟧*
*⟹ Γ ⊢ e1 «bop» e2 : High"*
— binäre Operation nur *Low*, wenn auch beide Teilausdrücke *Low*

# Sicherheitstypen von Ausdrücken

jedem Ausdruck (Konstanten *Val*, Variablen *Var*, binäre Operation «*bop*») mittels Typumgebung Sicherheitstyp zuordnen, Syntax Γ ⊢ *e* : *T*

**inductive** *secExprTyping* ::
*"typeEnv ⇒ expr ⇒ secType ⇒ bool" ("_ ⊢ _ : _")*
**where** *typeVal:* "Γ ⊢ *Val V* : *T*"
— Konstanten haben beliebigen Sicherheitstypen

| *typeVar:* "Γ *Vn* = *Some T* ⟹ Γ ⊢ *Var Vn* : *T*"
— Variablen sehen Sicherheitstyp in Typumgebung nach

| *typeBinOp1:* "⟦Γ ⊢ *e1* : *Low*; Γ ⊢ *e2* : *Low*⟧
⟹ Γ ⊢ *e1* «*bop*» *e2* : *Low*"
| *typeBinOp2:* "⟦Γ ⊢ *e1* : *High*; Γ ⊢ *e2* : *T*⟧
⟹ Γ ⊢ *e1* «*bop*» *e2* : *High*"
| *typeBinOp3:* "⟦Γ ⊢ *e1* : *T*; Γ ⊢ *e2* : *High*⟧
⟹ Γ ⊢ *e1* «*bop*» *e2* : *High*"
— binäre Operation nur *Low*, wenn auch beide Teilausdrücke *Low*

# Typüberprüfung von Anweisungen

$\Gamma, T \vdash c$ gilt, falls $c$ mit Typumgebung $\Gamma$ unter Sicherheitstyp $T$ typbar

**inductive** *secComTyping* ::
*"typeEnv ⇒ secType ⇒ com ⇒ bool" ("_,_ ⊢ _")*
**where** *typeSkip:* *"Γ,T ⊢ Skip"* — *Skip* immer typbar

| *typeAssH:* *"Γ V = Some High ⟹ Γ,T ⊢ V := e"*
— Zuweisung an *High* Variable immer typbar

| *typeAssL:* *"⟦Γ ⊢ e : Low; Γ V = Some Low⟧*
*⟹ Γ,Low ⊢ V := e"*
— wenn Variable *Low* und rechte Seite unter *Low* typbar,
dann auch Zuweisung an Variable unter *Low* typbar

| *typeSeq:* *"⟦Γ,T ⊢ c1; Γ,T ⊢ c2⟧ ⟹ Γ,T ⊢ c1;;c2"*
— wenn zwei Anweisungen unter $T$ typbar, dann auch ihre Komposition

# Typüberprüfung von Anweisungen

$\Gamma, T \vdash c$ gilt, falls $c$ mit Typumgebung $\Gamma$ unter Sicherheitstyp $T$ typbar

**inductive** *secComTyping* ::
*"typeEnv ⇒ secType ⇒ com ⇒ bool" ("_,_ ⊢ _")*
**where** *typeSkip:* *"Γ,T ⊢ Skip"* — *Skip* immer typbar

| *typeAssH:* *"Γ V = Some High ⟹ Γ,T ⊢ V := e"*
— Zuweisung an *High* Variable immer typbar

| *typeAssL:* *"⟦Γ ⊢ e : Low; Γ V = Some Low⟧*
*⟹ Γ,Low ⊢ V := e"*
— wenn Variable *Low* und rechte Seite unter *Low* typbar,
dann auch Zuweisung an Variable unter *Low* typbar

| *typeSeq:* *"⟦Γ,T ⊢ c1; Γ,T ⊢ c2⟧ ⟹ Γ,T ⊢ c1;;c2"*
— wenn zwei Anweisungen unter $T$ typbar, dann auch ihre Komposition

# Typüberprüfung von Anweisungen

$\Gamma, T \vdash c$ gilt, falls $c$ mit Typumgebung $\Gamma$ unter Sicherheitstyp $T$ typbar

**inductive** *secComTyping* ::
*"typeEnv ⇒ secType ⇒ com ⇒ bool" ("_,_ ⊢ _")*
**where** *typeSkip:* *"Γ,T ⊢ Skip"* — *Skip* immer typbar

| *typeAssH:* *"Γ V = Some High ⟹ Γ,T ⊢ V := e"*
— Zuweisung an *High* Variable immer typbar

| *typeAssL:* *"⟦Γ ⊢ e : Low; Γ V = Some Low⟧*
*⟹ Γ,Low ⊢ V := e"*
— wenn Variable *Low* und rechte Seite unter *Low* typbar,
dann auch Zuweisung an Variable unter *Low* typbar

| *typeSeq:* *"⟦Γ,T ⊢ c1; Γ,T ⊢ c2⟧ ⟹ Γ,T ⊢ c1;;c2"*
— wenn zwei Anweisungen unter $T$ typbar, dann auch ihre Komposition

# Typüberprüfung von Anweisungen

$\Gamma, T \vdash c$ gilt, falls $c$ mit Typumgebung $\Gamma$ unter Sicherheitstyp $T$ typbar

**inductive** *secComTyping* ::
"*typeEnv* ⇒ *secType* ⇒ *com* ⇒ *bool*" ("_,_ ⊢ _")
**where** *typeSkip:* "Γ,*T* ⊢ *Skip*" — *Skip* immer typbar

| *typeAssH:* "Γ *V* = *Some High* ⟹ Γ,*T* ⊢ *V* := *e*"
— Zuweisung an *High* Variable immer typbar

| *typeAssL:* "⟦Γ ⊢ *e* : *Low*; Γ *V* = *Some Low*⟧
⟹ Γ,*Low* ⊢ *V* := *e*"
— wenn Variable *Low* und rechte Seite unter *Low* typbar,
dann auch Zuweisung an Variable unter *Low* typbar

| *typeSeq:* "⟦Γ,*T* ⊢ *c1*; Γ,*T* ⊢ *c2*⟧ ⟹ Γ,*T* ⊢ *c1*;;*c2*"
— wenn zwei Anweisungen unter *T* typbar, dann auch ihre Komposition

# Typüberprüfung von Anweisungen

| *typeIf:* "⟦Γ ⊢ *b* : *T*; Γ,*T* ⊢ *c1*; Γ,*T* ⊢ *c2*⟧
⟹ Γ,*T* ⊢ *if (b) c1 else c2*"
— *if* unter *T* typbar, falls Prädikat Typ *T* hat
und beide Zweige unter *T* typbar

| *typeWhile:* "⟦Γ ⊢ *b* : *T*; Γ,*T* ⊢ *c*⟧ ⟹ Γ,*T* ⊢ *while (b) c*"
— *while* unter *T* typbar, falls Prädikat Typ *T* hat
und Rumpf unter *T* typbar

| *typeConvert:* "Γ,*High* ⊢ *c* ⟹ Γ,*Low* ⊢ *c*"
— wenn *c* unter *High* typbar, dann auch unter *Low*

# Typüberprüfung von Anweisungen

| *typeIf:* "⟦Γ ⊢ *b* : *T*; Γ,*T* ⊢ *c1*; Γ,*T* ⊢ *c2*⟧
⟹ Γ,*T* ⊢ *if (b) c1 else c2*"
— *if* unter *T* typbar, falls Prädikat Typ *T* hat
und beide Zweige unter *T* typbar

| *typeWhile:* "⟦Γ ⊢ *b* : *T*; Γ,*T* ⊢ *c*⟧ ⟹ Γ,*T* ⊢ *while (b) c*"
— *while* unter *T* typbar, falls Prädikat Typ *T* hat
und Rumpf unter *T* typbar

| *typeConvert:* "Γ,*High* ⊢ *c* ⟹ Γ,*Low* ⊢ *c*"
— wenn *c* unter *High* typbar, dann auch unter *Low*

# Typüberprüfung von Anweisungen

| *typeIf:* "⟦Γ ⊢ *b* : *T*; Γ,*T* ⊢ *c1*; Γ,*T* ⊢ *c2*⟧
⟹ Γ,*T* ⊢ *if (b) c1 else c2*"
— *if* unter *T* typbar, falls Prädikat Typ *T* hat
und beide Zweige unter *T* typbar

| *typeWhile:* "⟦Γ ⊢ *b* : *T*; Γ,*T* ⊢ *c*⟧ ⟹ Γ,*T* ⊢ *while (b) c*"
— *while* unter *T* typbar, falls Prädikat Typ *T* hat
und Rumpf unter *T* typbar

| *typeConvert:* "Γ,*High* ⊢ *c* ⟹ Γ,*Low* ⊢ *c*"
— wenn *c* unter *High* typbar, dann auch unter *Low*

# Low Equivalence

zwei Zustände low equivalent, wenn alle Werte in Low-Variablen gleich

**definition** *lowEquiv ::*
*"typeEnv ⇒ state ⇒ state ⇒ bool" ("_ ⊢ _ ≈L _")* **where**
*"Γ ⊢ s1 ≈L s2 ≡ ∀ v∈dom Γ. Γ v = Some Low ⟶ (s1 v = s2 v)"*

$\approx_L$ Äquivalenzrelation:

- **lemma** *lowEquivReflexive: "Γ ⊢ s1 ≈L s1"*
  **by** *(simp add:lowEquiv_def)*
- **lemma** *lowEquivSymmetric: "Γ ⊢ s1 ≈L s2 ⟹ Γ ⊢ s2 ≈L s1"*
  **by** *(simp add:lowEquiv_def)*
- **lemma** *lowEquivTransitive:*
  *"⟦Γ ⊢ s1 ≈L s2; Γ ⊢ s2 ≈L s3⟧ ⟹ Γ ⊢ s1 ≈L s3"*
  **by** *(simp add:lowEquiv_def)*

# Low Equivalence

zwei Zustände low equivalent, wenn alle Werte in Low-Variablen gleich

**definition** *lowEquiv ::*
*"typeEnv ⇒ state ⇒ state ⇒ bool" ("_ ⊢ _ ≈L _") **where***
*"Γ ⊢ s1 ≈L s2 ≡ ∀ v∈dom Γ. Γ v = Some Low ⟶ (s1 v = s2 v)"*

$\approx_L$ Äquivalenzrelation:

- **lemma** *lowEquivReflexive: "Γ ⊢ s1 ≈L s1"*
  **by**(*simp add:lowEquiv_def*)
- **lemma** *lowEquivSymmetric: "Γ ⊢ s1 ≈L s2 ⟹ Γ ⊢ s2 ≈L s1"*
  **by**(*simp add:lowEquiv_def*)
- **lemma** *lowEquivTransitive:*
  *"⟦Γ ⊢ s1 ≈L s2; Γ ⊢ s2 ≈L s3⟧ ⟹ Γ ⊢ s1 ≈L s3"*
  **by**(*simp add:lowEquiv_def*)

# Low Deterministic Security

Programm $c$ nichtinterferent nach Low Deterministic Security, falls Auswertung von $c$ gestartet in zwei "low equivalent" Zuständen $s_1$ und $s_2$ in zwei "low equivalent" Endzuständen $s_1'$ und $s_2'$ resultiert

**definition** *nonInterference :: "typeEnv ⇒ com ⇒ bool"*
**where** *"nonInterference Γ c ≡*
$(\forall s_1\ s_2\ s_1'\ s_2'.$
$(\Gamma \vdash s_1 \approx_L s_2 \wedge \langle c, s_1\rangle \rightarrow* \langle Skip, s_1'\rangle \wedge \langle c, s_2\rangle \rightarrow* \langle Skip, s_2'\rangle)$
$\longrightarrow \Gamma \vdash s_1' \approx_L s_2')"$

- $\rightarrow*$ reflexiv-transitive Hülle von $\rightarrow$, also beliebig viele Schritte
- komplette Auswertung von $c$ ist Auswertung bis *Skip*, da *Skip* nicht weiter auswertbar

# Low Deterministic Security

Programm $c$ nichtinterferent nach Low Deterministic Security, falls Auswertung von $c$ gestartet in zwei "low equivalent" Zuständen $s_1$ und $s_2$ in zwei "low equivalent" Endzuständen $s_1'$ und $s_2'$ resultiert

**definition** *nonInterference :: "typeEnv ⇒ com ⇒ bool"*
**where** *"nonInterference Γ c ≡*
$(\forall s_1\ s_2\ s_1'\ s_2'.$
$(\Gamma \vdash s_1 \approx_L s_2 \wedge \langle c, s_1\rangle \rightarrow* \langle Skip, s_1'\rangle \wedge \langle c, s_2\rangle \rightarrow* \langle Skip, s_2'\rangle)$
$\longrightarrow \Gamma \vdash s_1' \approx_L s_2')$"

- $\rightarrow*$ reflexiv-transitive Hülle von $\rightarrow$, also beliebig viele Schritte
- komplette Auswertung von $c$ ist Auswertung bis *Skip*, da *Skip* nicht weiter auswertbar

# Der Beweis

benötigt noch ein paar Hilfslemmas wie

Aufteilungslemmas der Semantik, z.B.

**lemma** *Seq_reds:* **assumes** "$\langle c_1;;c_2,s\rangle \rightarrow* \langle Skip,s'\rangle$"
**obtains** *s''* **where** "$\langle c_1,s\rangle \rightarrow* \langle Skip,s''\rangle$"
**and** "$\langle c_2,s''\rangle \rightarrow* \langle Skip,s'\rangle$"

Determinismuslemma der Semantik

**theorem** *reds_det:*
"$[\![\langle c,s\rangle \rightarrow* \langle Skip,s_1\rangle;\ \langle c,s\rangle \rightarrow* \langle Skip,s_2\rangle]\!] \Longrightarrow s_1 = s_2$"

Kompositionalitätslemmas für Nichtinterferenz, z.B.

**lemma** *CondLowCompositionality:*
**assumes** "*nonInterference* Γ *c1*" **and** "*nonInterference* Γ *c2*"
**and** "Γ ⊢ *b : Low*"
**shows** "*nonInterference* Γ *(if (b) c1 else c2)*"

Induktionslemma für While, sowohl auf Semantik-, als auch auf Nichtinterferenzebene

# Der Beweis

benötigt noch ein paar Hilfslemmas wie

Aufteilungslemmas der Semantik, z.B.

**lemma** *Seq_reds:* **assumes** "$\langle c_1;;c_2,s\rangle \rightarrow* \langle Skip,s''\rangle$"
**obtains** *s''* **where** "$\langle c_1,s\rangle \rightarrow* \langle Skip,s''\rangle$"
**and** "$\langle c_2,s''\rangle \rightarrow* \langle Skip,s'\rangle$"

Determinismuslemma der Semantik

**theorem** *reds_det:*
"$[\![\langle c,s\rangle \rightarrow* \langle Skip,s_1\rangle;\ \langle c,s\rangle \rightarrow* \langle Skip,s_2\rangle]\!] \Longrightarrow s_1 = s_2$"

Kompositionalitätslemmas für Nichtinterferenz, z.B.

**lemma** *CondLowCompositionality:*
**assumes** "*nonInterference* Γ *c1*" **and** "*nonInterference* Γ *c2*"
**and** "Γ ⊢ *b* : *Low*"
**shows** "*nonInterference* Γ *(if (b) c1 else c2)*"

Induktionslemma für While, sowohl auf Semantik-, als auch auf Nichtinterferenzebene

# Der Beweis

benötigt noch ein paar Hilfslemmas wie

Aufteilungslemmas der Semantik, z.B.

**lemma** *Seq_reds:* **assumes** "$\langle c_1;;c_2,s\rangle \rightarrow* \langle Skip,s'\rangle$"
**obtains** *s''* **where** "$\langle c_1,s\rangle \rightarrow* \langle Skip,s''\rangle$"
**and** "$\langle c_2,s''\rangle \rightarrow* \langle Skip,s'\rangle$"

Determinismuslemma der Semantik

**theorem** *reds_det:*
"$[\![\langle c,s\rangle \rightarrow* \langle Skip,s_1\rangle;\ \langle c,s\rangle \rightarrow* \langle Skip,s_2\rangle]\!] \Longrightarrow s_1 = s_2$"

Kompositionalitätslemmas für Nichtinterferenz, z.B.

**lemma** *CondLowCompositionality:*
**assumes** "*nonInterference* Γ *c1*" **and** "*nonInterference* Γ *c2*"
**and** "Γ ⊢ *b : Low*"
**shows** "*nonInterference* Γ *(if (b) c1 else c2)*"

Induktionslemma für While, sowohl auf Semantik-, als auch auf Nichtinterferenzebene

# Der Beweis

benötigt noch ein paar Hilfslemmas wie

Aufteilungslemmas der Semantik, z.B.

**lemma** *Seq_reds:* **assumes** "$\langle c_1;;c_2,s\rangle \rightarrow* \langle Skip,s'\rangle$"
**obtains** *s''* **where** "$\langle c_1,s\rangle \rightarrow* \langle Skip,s''\rangle$"
**and** "$\langle c_2,s''\rangle \rightarrow* \langle Skip,s'\rangle$"

Determinismuslemma der Semantik

**theorem** *reds_det:*
"$[\![\langle c,s\rangle \rightarrow* \langle Skip,s_1\rangle;\ \langle c,s\rangle \rightarrow* \langle Skip,s_2\rangle]\!] \Longrightarrow s_1 = s_2$"

Kompositionalitätslemmas für Nichtinterferenz, z.B.

**lemma** *CondLowCompositionality:*
**assumes** "*nonInterference* Γ *c1*" **and** "*nonInterference* Γ *c2*"
**and** "Γ ⊢ *b : Low*"
**shows** "*nonInterference* Γ *(if (b) c1 else c2)*"

Induktionslemma für While, sowohl auf Semantik-, als auch auf Nichtinterferenzebene

# Der Beweis

Das Haupttheorem:

**theorem** *secTypeImpliesNonInterference:*
*"Γ,T ⊢ c ⟹ nonInterference Γ c"*

bewiesen durch

- strukturelle Induktion nach *c* und
- Fallunterscheidung nach *T* für *Seq*, *Cond* und *While*


http://afp.sf.net/entries/VolpanoSmith.shtml

- gut lesbarer, kürzerer Beweis ($\sim$ 1300 Zeilen insgesamt)
- verwendet allerdings etwas kompliziertere *lowEquiv* Definition

# Der Beweis

Das Haupttheorem:

**theorem** *secTypeImpliesNonInterference:*
*"Γ,T ⊢ c ⟹ nonInterference Γ c"*

bewiesen durch

- strukturelle Induktion nach *c* und
- Fallunterscheidung nach *T* für *Seq*, *Cond* und *While*


`http://afp.sf.net/entries/VolpanoSmith.shtml`

- gut lesbarer, kürzerer Beweis ($\sim$ 1300 Zeilen insgesamt)
- verwendet allerdings etwas kompliziertere *lowEquiv* Definition

# weiterführende Literatur


A. Sabelfeld and A. C. Myers.
Language-Based Information-Flow Security.
*Journal on Selected Areas in Communications*, 21(1):5–19. IEEE, 2003
http://www.cs.chalmers.se/~andrei/jsac.pdf


L. Beringer and M. Hofman.
Secure information flow and program logics.
In *Proc. of Computer Security Foundations Symposium*. IEEE, 2007
http://afp.sf.net/entries/SIFPL.shtml

F. Kammüller.
Formalizing non-interference for a simple bytecode language in Coq.
*Formal Aspects of Computing*, 20(3):259–275, Springer, 2008.
http://dx.doi.org/10.1007/s00165-007-0055-2