

An Isabelle Correctness Proof for the Volpano/Smith Security Typing System

Gregor Snelting and Daniel Wasserrab
IPD Snelting
Universität Karlsruhe (TH)

December 14, 2021

Abstract

The Volpano/Smith/Irvine security type systems [2] requires that variables are annotated as high (secret) or low (public), and provides typing rules which guarantee that secret values cannot leak to public output ports. This property of a program is called confidentiality.

For a simple while-language without threads, our proof shows that typeability in the Volpano/Smith system guarantees noninterference. Noninterference means that if two initial states for program execution are low-equivalent, then the final states are low-equivalent as well. This indeed implies that secret values cannot leak to public ports. For more details on noninterference and security typing systems, see [1].

The proof defines an abstract syntax and operational semantics for programs, formalizes noninterference, and then proceeds by rule induction on the operational semantics. The mathematically most intricate part is the treatment of implicit flows. Note that the Volpano/Smith system is not flow-sensitive and thus quite unprecise, resulting in false alarms. However, due to the correctness property, all potential breaks of confidentiality are discovered.

Contents

1	The Language	3
1.1	Variables and Values	3
1.2	Expressions and Commands	3
1.3	State	4
1.4	Small Step Semantics	4
2	Security types	7
2.1	Security definitions	7
2.2	Lemmas concerning expressions	7
2.3	Noninterference definitions	8
2.3.1	Low Equivalence	8
2.3.2	Non Interference	9
3	Executing the small step semantics	11
3.1	An example taken from Volpano, Smith, Irvine	12

```

theory Semantics
imports Main
begin

```

1 The Language

1.1 Variables and Values

type-synonym *vname* = *string* — names for variables

```

datatype val
  = Bool bool      — Boolean value
  | Intg int       — integer value

```

```

abbreviation true == Bool True
abbreviation false == Bool False

```

1.2 Expressions and Commands

datatype *bop* = *Eq* | *And* | *Less* | *Add* | *Sub* — names of binary operations

```

datatype expr
  = Val val                — value
  | Var vname              — local variable
  | BinOp expr bop expr    (- «-» - [80,0,81] 80) — binary operation

```

Note: we assume that only type correct expressions are regarded as later proofs fail if expressions evaluate to `None` due to type errors. However there is [yet] no typing system

```

fun binop :: bop ⇒ val ⇒ val ⇒ val option
where binop Eq v1 v2          = Some(Bool(v1 = v2))
  | binop And (Bool b1) (Bool b2) = Some(Bool(b1 ∧ b2))
  | binop Less (Intg i1) (Intg i2) = Some(Bool(i1 < i2))
  | binop Add (Intg i1) (Intg i2) = Some(Intg(i1 + i2))
  | binop Sub (Intg i1) (Intg i2) = Some(Intg(i1 - i2))

  | binop bop v1 v2          = Some(Intg(0))

```

```

datatype com
  = Skip
  | LAss vname expr      (-:=- [70,70] 70) — local assignment
  | Seq com com         (-;;/ - [61,60] 60)
  | Cond expr com com  (if '(-) -/ else - [80,79,79] 70)
  | While expr com     (while '(-) - [80,79] 70)

```

```

fun fv :: expr ⇒ vname set — free variables in an expression
where

```

$FVc: fv (Val V) = \{\}$
 $| FVv: fv (Var V) = \{V\}$
 $| FVe: fv (e1 \ll bop \gg e2) = fv e1 \cup fv e2$

1.3 State

type-synonym $state = vname \rightarrow val$

interpret silently assumes type correct expressions, i.e. no expression evaluates to None

fun *interpret* :: $expr \Rightarrow state \Rightarrow val option$ ($\llbracket - \rrbracket$)
where *Val*: $\llbracket Val v \rrbracket s = Some v$
 $|$ *Var*: $\llbracket Var V \rrbracket s = s V$
 $|$ *BinOp*: $\llbracket e1 \ll bop \gg e2 \rrbracket s = (case \llbracket e1 \rrbracket s of None \Rightarrow None$
 $| Some v_1 \Rightarrow (case \llbracket e2 \rrbracket s of None \Rightarrow None$
 $| Some v_2 \Rightarrow binop bop v_1 v_2))$

1.4 Small Step Semantics

inductive *red* :: $com * state \Rightarrow com * state \Rightarrow bool$

and *red'* :: $com \Rightarrow state \Rightarrow com \Rightarrow state \Rightarrow bool$

$((\llbracket (I \langle -, / - \rangle) \rightarrow / (I \langle -, / - \rangle) \rrbracket) [0,0,0,0] 81)$

where

$\langle c1, s1 \rangle \rightarrow \langle c2, s2 \rangle == red (c1, s1) (c2, s2) |$

RedLAss:

$\langle V := e, s \rangle \rightarrow \langle Skip, s (V := (\llbracket e \rrbracket s)) \rangle$

$|$ *SeqRed*:

$\langle c_1, s \rangle \rightarrow \langle c_1', s' \rangle \implies \langle c_1 ;; c_2, s \rangle \rightarrow \langle c_1' ;; c_2, s' \rangle$

$|$ *RedSeq*:

$\langle Skip ;; c_2, s \rangle \rightarrow \langle c_2, s \rangle$

$|$ *RedCondTrue*:

$\llbracket b \rrbracket s = Some true \implies \langle if (b) c_1 else c_2, s \rangle \rightarrow \langle c_1, s \rangle$

$|$ *RedCondFalse*:

$\llbracket b \rrbracket s = Some false \implies \langle if (b) c_1 else c_2, s \rangle \rightarrow \langle c_2, s \rangle$

$|$ *RedWhileTrue*:

$\llbracket b \rrbracket s = Some true \implies \langle while (b) c, s \rangle \rightarrow \langle c ;; while (b) c, s \rangle$

$|$ *RedWhileFalse*:

$\llbracket b \rrbracket s = Some false \implies \langle while (b) c, s \rangle \rightarrow \langle Skip, s \rangle$

lemmas *red-induct* = *red.induct*[*split-format* (*complete*)]

abbreviation *reds* :: $com \Rightarrow state \Rightarrow com \Rightarrow state \Rightarrow bool$

$((\llbracket (I \langle -, / - \rangle) \rightarrow * / (I \langle -, / - \rangle) \rrbracket) [0,0,0,0] 81) \text{ where}$

$\langle c, s \rangle \rightarrow * \langle c', s' \rangle == red^{**} (c, s) (c', s')$

lemma *Skip-reds*:

$\langle \text{Skip}, s \rangle \rightarrow^* \langle c', s' \rangle \implies s = s' \wedge c' = \text{Skip}$
 $\langle \text{proof} \rangle$

lemma *LAss-reds*:

$\langle V := e, s \rangle \rightarrow^* \langle \text{Skip}, s' \rangle \implies s' = s(V := \llbracket e \rrbracket s)$
 $\langle \text{proof} \rangle$

lemma *Seq2-reds*:

$\langle \text{Skip}; c_2, s \rangle \rightarrow^* \langle \text{Skip}, s' \rangle \implies \langle c_2, s \rangle \rightarrow^* \langle \text{Skip}, s' \rangle$
 $\langle \text{proof} \rangle$

lemma *Seq-reds*:

assumes $\langle c_1; c_2, s \rangle \rightarrow^* \langle \text{Skip}, s' \rangle$
obtains s'' **where** $\langle c_1, s \rangle \rightarrow^* \langle \text{Skip}, s'' \rangle$ **and** $\langle c_2, s'' \rangle \rightarrow^* \langle \text{Skip}, s' \rangle$
 $\langle \text{proof} \rangle$

lemma *Cond-True-or-False*:

$\langle \text{if } (b) \ c_1 \ \text{else } c_2, s \rangle \rightarrow^* \langle \text{Skip}, s' \rangle \implies \llbracket b \rrbracket s = \text{Some true} \vee \llbracket b \rrbracket s = \text{Some false}$
 $\langle \text{proof} \rangle$

lemma *CondTrue-reds*:

$\langle \text{if } (b) \ c_1 \ \text{else } c_2, s \rangle \rightarrow^* \langle \text{Skip}, s' \rangle \implies \llbracket b \rrbracket s = \text{Some true} \implies \langle c_1, s \rangle \rightarrow^* \langle \text{Skip}, s' \rangle$
 $\langle \text{proof} \rangle$

lemma *CondFalse-reds*:

$\langle \text{if } (b) \ c_1 \ \text{else } c_2, s \rangle \rightarrow^* \langle \text{Skip}, s' \rangle \implies \llbracket b \rrbracket s = \text{Some false} \implies \langle c_2, s \rangle \rightarrow^* \langle \text{Skip}, s' \rangle$
 $\langle \text{proof} \rangle$

lemma *WhileFalse-reds*:

$\langle \text{while } (b) \ cx, s \rangle \rightarrow^* \langle \text{Skip}, s' \rangle \implies \llbracket b \rrbracket s = \text{Some false} \implies s = s'$
 $\langle \text{proof} \rangle$

lemma *WhileTrue-reds*:

$\langle \text{while } (b) \ cx, s \rangle \rightarrow^* \langle \text{Skip}, s' \rangle \implies \llbracket b \rrbracket s = \text{Some true}$
 $\implies \exists sx. \langle cx, s \rangle \rightarrow^* \langle \text{Skip}, sx \rangle \wedge \langle \text{while } (b) \ cx, sx \rangle \rightarrow^* \langle \text{Skip}, s' \rangle$
 $\langle \text{proof} \rangle$

lemma *While-True-or-False*:

$\langle \text{while } (b) \ com, s \rangle \rightarrow^* \langle \text{Skip}, s' \rangle \implies \llbracket b \rrbracket s = \text{Some true} \vee \llbracket b \rrbracket s = \text{Some false}$
 $\langle \text{proof} \rangle$

inductive *red-n* :: *com* \Rightarrow *state* \Rightarrow *nat* \Rightarrow *com* \Rightarrow *state* \Rightarrow *bool*

$((1 \langle -, /- \rangle) \rightarrow^* (1 \langle -, /- \rangle)) [0, 0, 0, 0, 0] \ 81)$

where *red-n-Base*: $\langle c, s \rangle \rightarrow^0 \langle c, s \rangle$

| *red-n-Rec*: $\llbracket \langle c, s \rangle \rightarrow \langle c'', s'' \rangle; \langle c'', s'' \rangle \rightarrow^n \langle c', s' \rangle \rrbracket \implies \langle c, s \rangle \rightarrow^{\text{Suc } n} \langle c', s' \rangle$

lemma *Seq-red-nE*: **assumes** $\langle c_1; c_2, s \rangle \rightarrow^n \langle \text{Skip}, s' \rangle$
obtains $i\ j\ s''$ **where** $\langle c_1, s \rangle \rightarrow^i \langle \text{Skip}, s'' \rangle$ **and** $\langle c_2, s'' \rangle \rightarrow^j \langle \text{Skip}, s' \rangle$
and $n = i + j + 1$
 $\langle \text{proof} \rangle$

lemma *while-red-nE*:
 $\langle \text{while } (b)\ cx, s \rangle \rightarrow^n \langle \text{Skip}, s' \rangle$
 $\implies (\llbracket b \rrbracket s = \text{Some false} \wedge s = s' \wedge n = 1) \vee$
 $(\exists i\ j\ s''. \llbracket b \rrbracket s = \text{Some true} \wedge \langle cx, s \rangle \rightarrow^i \langle \text{Skip}, s'' \rangle \wedge$
 $\langle \text{while } (b)\ cx, s'' \rangle \rightarrow^j \langle \text{Skip}, s' \rangle \wedge n = i + j + 2)$
 $\langle \text{proof} \rangle$

lemma *while-red-n-induct* [*consumes 1, case-names false true*]:
assumes *major*: $\langle \text{while } (b)\ cx, s \rangle \rightarrow^n \langle \text{Skip}, s' \rangle$
and *IHfalse*: $\bigwedge s. \llbracket b \rrbracket s = \text{Some false} \implies P\ s\ s'$
and *IHtrue*: $\bigwedge s\ i\ j\ s''. \llbracket b \rrbracket s = \text{Some true}; \langle cx, s \rangle \rightarrow^i \langle \text{Skip}, s'' \rangle;$
 $\langle \text{while } (b)\ cx, s'' \rangle \rightarrow^j \langle \text{Skip}, s' \rangle; P\ s''\ s' \implies P\ s\ s'$
shows $P\ s\ s'$
 $\langle \text{proof} \rangle$

lemma *reds-to-red-n*: $\langle c, s \rangle \rightarrow^* \langle c', s' \rangle \implies \exists n. \langle c, s \rangle \rightarrow^n \langle c', s' \rangle$
 $\langle \text{proof} \rangle$

lemma *red-n-to-reds*: $\langle c, s \rangle \rightarrow^n \langle c', s' \rangle \implies \langle c, s \rangle \rightarrow^* \langle c', s' \rangle$
 $\langle \text{proof} \rangle$

lemma *while-reds-induct* [*consumes 1, case-names false true*]:
 $\llbracket \langle \text{while } (b)\ cx, s \rangle \rightarrow^* \langle \text{Skip}, s' \rangle; \bigwedge s. \llbracket b \rrbracket s = \text{Some false} \implies P\ s\ s';$
 $\bigwedge s\ s''. \llbracket b \rrbracket s = \text{Some true}; \langle cx, s \rangle \rightarrow^* \langle \text{Skip}, s'' \rangle;$
 $\langle \text{while } (b)\ cx, s'' \rangle \rightarrow^* \langle \text{Skip}, s' \rangle; P\ s''\ s' \rrbracket \implies P\ s\ s'$
 $\implies P\ s\ s'$
 $\langle \text{proof} \rangle$

lemma *red-det*:
 $\llbracket \langle c, s \rangle \rightarrow \langle c_1, s_1 \rangle; \langle c, s \rangle \rightarrow \langle c_2, s_2 \rangle \rrbracket \implies c_1 = c_2 \wedge s_1 = s_2$
 $\langle \text{proof} \rangle$

theorem *reds-det*:
 $\llbracket \langle c, s \rangle \rightarrow^* \langle \text{Skip}, s_1 \rangle; \langle c, s \rangle \rightarrow^* \langle \text{Skip}, s_2 \rangle \rrbracket \implies s_1 = s_2$
 $\langle \text{proof} \rangle$

```

end
theory secTypes
imports Semantics
begin

```

2 Security types

2.1 Security definitions

```

datatype secLevel = Low | High

```

```

type-synonym typeEnv = vname  $\rightarrow$  secLevel

```

```

inductive secExprTyping :: typeEnv  $\Rightarrow$  expr  $\Rightarrow$  secLevel  $\Rightarrow$  bool (-  $\vdash$  - : -)
where typeVal:  $\Gamma \vdash \text{Val } V : \text{lev}$ 

```

```

| typeVar:  $\Gamma \ Vn = \text{Some } \text{lev} \Longrightarrow \Gamma \vdash \text{Var } Vn : \text{lev}$ 

```

```

| typeBinOp1:  $[\Gamma \vdash e1 : \text{Low}; \Gamma \vdash e2 : \text{Low}] \Longrightarrow \Gamma \vdash e1 \ll\text{bop}\gg e2 : \text{Low}$ 

```

```

| typeBinOp2:  $[\Gamma \vdash e1 : \text{High}; \Gamma \vdash e2 : \text{lev}] \Longrightarrow \Gamma \vdash e1 \ll\text{bop}\gg e2 : \text{High}$ 

```

```

| typeBinOp3:  $[\Gamma \vdash e1 : \text{lev}; \Gamma \vdash e2 : \text{High}] \Longrightarrow \Gamma \vdash e1 \ll\text{bop}\gg e2 : \text{High}$ 

```

```

inductive secComTyping :: typeEnv  $\Rightarrow$  secLevel  $\Rightarrow$  com  $\Rightarrow$  bool (-, -  $\vdash$  -)
where typeSkip:  $\Gamma, T \vdash \text{Skip}$ 

```

```

| typeAssH:  $\Gamma \ V = \text{Some } \text{High} \Longrightarrow \Gamma, T \vdash V := e$ 

```

```

| typeAssL:  $[\Gamma \vdash e : \text{Low}; \Gamma \ V = \text{Some } \text{Low}] \Longrightarrow \Gamma, \text{Low} \vdash V := e$ 

```

```

| typeSeq:  $[\Gamma, T \vdash c1; \Gamma, T \vdash c2] \Longrightarrow \Gamma, T \vdash c1;;c2$ 

```

```

| typeWhile:  $[\Gamma \vdash b : T; \Gamma, T \vdash c] \Longrightarrow \Gamma, T \vdash \text{while } (b) \ c$ 

```

```

| typeIf:  $[\Gamma \vdash b : T; \Gamma, T \vdash c1; \Gamma, T \vdash c2] \Longrightarrow \Gamma, T \vdash \text{if } (b) \ c1 \ \text{else } \ c2$ 

```

```

| typeConvert:  $\Gamma, \text{High} \vdash c \Longrightarrow \Gamma, \text{Low} \vdash c$ 

```

2.2 Lemmas concerning expressions

```

lemma exprTypeable:

```

```

  assumes  $fv \ e \subseteq \text{dom } \Gamma$  obtains  $T$  where  $\Gamma \vdash e : T$ 
  <proof>

```

lemma *exprBinopTypeable*:
assumes $\Gamma \vdash e1 \llbracket \text{bop} \rrbracket e2 : T$
shows $(\exists T1. \Gamma \vdash e1 : T1) \wedge (\exists T2. \Gamma \vdash e2 : T2)$
 $\langle \text{proof} \rangle$

lemma *exprTypingHigh*:
assumes $\Gamma \vdash e : T$ **and** $x \in \text{fv } e$ **and** $\Gamma x = \text{Some High}$
shows $\Gamma \vdash e : \text{High}$
 $\langle \text{proof} \rangle$

lemma *exprTypingLow*:
assumes $\Gamma \vdash e : \text{Low}$ **and** $x \in \text{fv } e$ **shows** $\Gamma x = \text{Some Low}$
 $\langle \text{proof} \rangle$

lemma *typeableFreevars*:
assumes $\Gamma \vdash e : T$ **shows** $\text{fv } e \subseteq \text{dom } \Gamma$
 $\langle \text{proof} \rangle$

lemma *exprNotNone*:
assumes $\Gamma \vdash e : T$ **and** $\text{fv } e \subseteq \text{dom } s$
shows $\llbracket e \rrbracket s \neq \text{None}$
 $\langle \text{proof} \rangle$

2.3 Noninterference definitions

2.3.1 Low Equivalence

Low Equivalence is reflexive even if the involved states are undefined. But in non-reflexive situations low variables must be initialized (i.e. $\in \text{dom } \text{state}$), otherwise the proof will not work. This is not a restriction, but a natural requirement, and could be formalized as part of a standard type system.

Low equivalence is also symmetric and transitive (see lemmas) hence an equivalence relation.

definition *lowEquiv* :: $\text{typeEnv} \Rightarrow \text{state} \Rightarrow \text{state} \Rightarrow \text{bool}$ ($- \vdash - \approx_L -$)
where $\Gamma \vdash s1 \approx_L s2 \equiv \forall v \in \text{dom } \Gamma. \Gamma v = \text{Some Low} \longrightarrow (s1 v = s2 v)$

lemma *lowEquivReflexive*: $\Gamma \vdash s1 \approx_L s1$
 $\langle \text{proof} \rangle$

lemma *lowEquivSymmetric*:
 $\Gamma \vdash s1 \approx_L s2 \implies \Gamma \vdash s2 \approx_L s1$

<proof>

lemma *lowEquivTransitive*:

$\llbracket \Gamma \vdash s1 \approx_L s2; \Gamma \vdash s2 \approx_L s3 \rrbracket \implies \Gamma \vdash s1 \approx_L s3$

<proof>

2.3.2 Non Interference

definition *nonInterference* :: *typeEnv* \Rightarrow *com* \Rightarrow *bool*

where *nonInterference* Γ *c* \equiv

$(\forall s1\ s2\ s1'\ s2'. (\Gamma \vdash s1 \approx_L s2 \wedge \langle c, s1 \rangle \rightarrow^* \langle Skip, s1' \rangle \wedge \langle c, s2 \rangle \rightarrow^* \langle Skip, s2' \rangle) \rightarrow \Gamma \vdash s1' \approx_L s2')$

lemma *nonInterferenceI*:

$\llbracket \bigwedge s1\ s2\ s1'\ s2'. (\Gamma \vdash s1 \approx_L s2; \langle c, s1 \rangle \rightarrow^* \langle Skip, s1' \rangle; \langle c, s2 \rangle \rightarrow^* \langle Skip, s2' \rangle) \rrbracket \implies \Gamma \vdash s1' \approx_L s2' \rrbracket \implies \text{nonInterference } \Gamma\ c$

<proof>

lemma *interpretLow*:

assumes $\Gamma \vdash s1 \approx_L s2$ **and** *all*: $\forall V \in \text{fv } e. \Gamma\ V = \text{Some Low}$

shows $\llbracket e \rrbracket s1 = \llbracket e \rrbracket s2$

<proof>

lemma *interpretLow2*:

assumes $\Gamma \vdash e : \text{Low}$ **and** $\Gamma \vdash s1 \approx_L s2$ **shows** $\llbracket e \rrbracket s1 = \llbracket e \rrbracket s2$

<proof>

lemma *assignNIhighlemma*:

assumes $\Gamma \vdash s1 \approx_L s2$ **and** $\Gamma\ V = \text{Some High}$ **and** $s1' = s1(V := \llbracket e \rrbracket s1)$

and $s2' = s2(V := \llbracket e \rrbracket s2)$

shows $\Gamma \vdash s1' \approx_L s2'$

<proof>

lemma *assignNIlowlemma*:

assumes $\Gamma \vdash s1 \approx_L s2$ **and** $\Gamma\ V = \text{Some Low}$ **and** $\Gamma \vdash e : \text{Low}$

and $s1' = s1(V := \llbracket e \rrbracket s1)$ **and** $s2' = s2(V := \llbracket e \rrbracket s2)$

shows $\Gamma \vdash s1' \approx_L s2'$

<proof>

Sequential Compositionality is given the status of a theorem because compositionality is no longer valid in case of concurrency

theorem *SeqCompositionality*:

assumes *nonInterference* $\Gamma\ c1$ **and** *nonInterference* $\Gamma\ c2$

shows *nonInterference* $\Gamma\ (c1;;c2)$

$\langle proof \rangle$

lemma *WhileStepInduct*:

assumes $while: \langle while (b) c, s1 \rangle \rightarrow^* \langle Skip, s2 \rangle$

and $body: \bigwedge s1 s2. \langle c, s1 \rangle \rightarrow^* \langle Skip, s2 \rangle \implies \Gamma \vdash s1 \approx_L s2$ **and** $\Gamma, High \vdash c$

shows $\Gamma \vdash s1 \approx_L s2$

$\langle proof \rangle$

In case control conditions from if/while are high, the body of an if/while must not change low variables in order to prevent implicit flow. That is, start and end state of any if/while body must be low equivalent.

theorem *highBodies*:

assumes $\Gamma, High \vdash c$ **and** $\langle c, s1 \rangle \rightarrow^* \langle Skip, s2 \rangle$

shows $\Gamma \vdash s1 \approx_L s2$

— all intermediate states must be well formed, otherwise the proof does not work for uninitialized variables. Thus it is propagated through the theorem conclusion

$\langle proof \rangle$

lemma *CondHighCompositionality*:

assumes $\Gamma, High \vdash if (b) c1 else c2$

shows $nonInterference \Gamma (if (b) c1 else c2)$

$\langle proof \rangle$

lemma *CondLowCompositionality*:

assumes $nonInterference \Gamma c1$ **and** $nonInterference \Gamma c2$ **and** $\Gamma \vdash b : Low$

shows $nonInterference \Gamma (if (b) c1 else c2)$

$\langle proof \rangle$

lemma *WhileHighCompositionality*:

assumes $\Gamma, High \vdash while (b) c'$

shows $nonInterference \Gamma (while (b) c')$

$\langle proof \rangle$

lemma *WhileLowStepInduct*:

assumes $while1: \langle while (b) c', s1 \rangle \rightarrow^* \langle Skip, s1' \rangle$

and $while2: \langle while (b) c', s2 \rangle \rightarrow^* \langle Skip, s2' \rangle$

and $\Gamma \vdash b : Low$

and $body: \bigwedge s1 s1' s2 s2'. \llbracket \langle c', s1 \rangle \rightarrow^* \langle Skip, s1' \rangle; \langle c', s2 \rangle \rightarrow^* \langle Skip, s2' \rangle; \Gamma \vdash s1 \approx_L s2 \rrbracket \implies \Gamma \vdash s1' \approx_L s2'$

and $le: \Gamma \vdash s1 \approx_L s2$

shows $\Gamma \vdash s1' \approx_L s2'$
 ⟨proof⟩

lemma *WhileLowCompositionality*:
assumes *nonInterference* $\Gamma \vdash c'$ **and** $\Gamma \vdash b : Low$ **and** $\Gamma, Low \vdash c'$
shows *nonInterference* $\Gamma \vdash (while\ (b)\ c')$
 ⟨proof⟩

and now: the main theorem:

theorem *secTypeImpliesNonInterference*:
 $\Gamma, T \vdash c \implies nonInterference\ \Gamma\ c$
 ⟨proof⟩

end

theory *Execute*
imports *secTypes*
begin

3 Executing the small step semantics

code-pred (*modes: $i \implies o \implies bool$ as exec-red, $i \implies i * o \implies bool$, $i \implies o * i \implies bool$, $i \implies i \implies bool$*) *red* ⟨proof⟩

thm *red.equation*

definition [*code*]: *one-step $x = Predicate.the\ (exec-red\ x)$*

lemmas [*code-pred-intro*] = *typeVal[where lev = Low] typeVal[where lev = High]*
typeVar
typeBinOp1 typeBinOp2[where lev = Low] typeBinOp2[where lev = High] type-
BinOp3[where lev = Low]

code-pred (*modes: $i \implies i \implies o \implies bool$ as compute-secExprTyping,*
 $i \implies i \implies i \implies bool$ as check-secExprTyping) *secExprTyping*
 ⟨proof⟩

lemmas [*code-pred-intro*] = *typeSkip[where T=Low] typeSkip[where T=High]*
typeAssH[where T = Low] typeAssH[where T=High]
typeAssL typeSeq typeWhile typeIf typeConvert

code-pred (*modes: $i \implies o \implies i \implies bool$ as compute-secComTyping,*
 $i \implies i \implies i \implies bool$ as check-secComTyping) *secComTyping*
 ⟨proof⟩

thm *secComTyping.equation*

3.1 An example taken from Volpano, Smith, Irvine

definition $com = \text{if } (Var \text{ ''}x'' \ll Eq \gg Val (Intg 1)) \text{ ''}y'' := Val (Intg 1) \text{ else } (''y'' := Val (Intg 0))$

definition $Env = \text{map-of } [(''x'', High), (''y'', High)]$

values $\{T. Env \vdash (Var \text{ ''}x'' \ll Eq \gg Val (Intg 1)): T\}$

value $Env, High \vdash com$

value $Env, Low \vdash com$

values $1 \{T. Env, T \vdash com\}$

definition $Env' = \text{map-of } [(''x'', Low), (''y'', High)]$

value $Env', Low \vdash com$

value $Env', High \vdash com$

values $1 \{T. Env, T \vdash com\}$

end

References

- [1] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
- [2] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, 1996.