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

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

March 17, 2025

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

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* v_1 v_2 = *Some*(*Bool*(v_1 = v_2))
| *binop And* (*Bool* b_1) (*Bool* b_2) = *Some*(*Bool*(b_1 ∧ b_2))
| *binop Less* (*Intg* i_1) (*Intg* i_2) = *Some*(*Bool*(i_1 < i_2))
| *binop Add* (*Intg* i_1) (*Intg* i_2) = *Some*(*Intg*(i_1 + i_2))
| *binop Sub* (*Intg* i_1) (*Intg* i_2) = *Some*(*Intg*(i_1 − i_2))

| *binop bop* v_1 v_2 = *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* «*bop*» *e2*) = *fv e1* ∪ *fv e2*

1.3 State

type-synonym *state* = *vname* ⇀ *val*

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

fun *interpret* :: *expr* ⇒ *state* ⇒ *val option* (‹⟦-⟧-›)
where *Val*: ⟦*Val v*⟧ *s* = *Some v*
| *Var*: ⟦*Var V*⟧ *s* = *s V*
| *BinOp*: ⟦e_1«*bop*»e_2⟧ *s* = (*case* ⟦e_1⟧ *s of None* ⇒ *None*
| *Some* v_1 ⇒ (*case* ⟦e_2⟧ *s of None* ⇒ *None*
| *Some* v_2 ⇒ *binop bop* v_1 v_2))

1.4 Small Step Semantics

inductive *red* :: *com* ∗ *state* ⇒ *com* ∗ *state* ⇒ *bool*
and *red′* :: *com* ⇒ *state* ⇒ *com* ⇒ *state* ⇒ *bool*
(‹((*1*⟨-,/-⟩) →/ (*1*⟨-,/-⟩))› [*0*,*0*,*0*,*0*] *81*)
where
⟨*c1*,*s1*⟩ → ⟨*c2*,*s2*⟩ == *red* (*c1*,*s1*) (*c2*,*s2*) |
RedLAss:
⟨*V*:=*e*,*s*⟩ → ⟨*Skip*,*s*(*V*:=(⟦*e*⟧ *s*))⟩

| *SeqRed*:
⟨c_1,*s*⟩ → ⟨c_1′,*s*′⟩ ⟹ ⟨c_1;;c_2,*s*⟩ → ⟨c_1′;;c_2,*s*′⟩

| *RedSeq*:
⟨*Skip*;;c_2,*s*⟩ → ⟨c_2,*s*⟩

| *RedCondTrue*:
⟦*b*⟧ *s* = *Some true* ⟹ ⟨*if* (*b*) c_1 *else* c_2,*s*⟩ → ⟨c_1,*s*⟩

| *RedCondFalse*:
⟦*b*⟧ *s* = *Some false* ⟹ ⟨*if* (*b*) c_1 *else* c_2,*s*⟩ → ⟨c_2,*s*⟩

| *RedWhileTrue*:
⟦*b*⟧ *s* = *Some true* ⟹ ⟨*while* (*b*) *c*,*s*⟩ → ⟨*c*;;*while* (*b*) *c*,*s*⟩

| *RedWhileFalse*:
⟦*b*⟧ *s* = *Some false* ⟹ ⟨*while* (*b*) *c*,*s*⟩ → ⟨*Skip*,*s*⟩

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

abbreviation *reds* ::*com* ⇒ *state* ⇒ *com* ⇒ *state* ⇒ *bool*
(‹((*1*⟨-,/-⟩) →∗/ (*1*⟨-,/-⟩))› [*0*,*0*,*0*,*0*] *81*) **where**
⟨*c*,*s*⟩ →∗ ⟨*c′*,*s′*⟩ == *red*∗∗ (*c*,*s*) (*c′*,*s′*)

lemma *Skip-reds*:
$\langle Skip,s\rangle \rightarrow* \langle c',s'\rangle \Longrightarrow s = s' \wedge c' = Skip$
⟨*proof*⟩

lemma *LAss-reds*:
$\langle V{:=}e,s\rangle \rightarrow* \langle Skip,s'\rangle \Longrightarrow s' = s(V{:=}[\![e]\!]\ s)$
⟨*proof*⟩

lemma *Seq2-reds*:
$\langle Skip;;c_2,s\rangle \rightarrow* \langle Skip,s'\rangle \Longrightarrow \langle c_2,s\rangle \rightarrow* \langle Skip,s'\rangle$
⟨*proof*⟩

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** $\langle c_2,s''\rangle \rightarrow* \langle Skip,s'\rangle$
⟨*proof*⟩

lemma *Cond-True-or-False*:
$\langle if\ (b)\ c_1\ else\ c_2,s\rangle \rightarrow* \langle Skip,s'\rangle \Longrightarrow [\![b]\!]\ s = Some\ true \vee [\![b]\!]\ s = Some\ false$
⟨*proof*⟩

lemma *CondTrue-reds*:
$\langle if\ (b)\ c_1\ else\ c_2,s\rangle \rightarrow* \langle Skip,s'\rangle \Longrightarrow [\![b]\!]\ s = Some\ true \Longrightarrow \langle c_1,s\rangle \rightarrow* \langle Skip,s'\rangle$
⟨*proof*⟩

lemma *CondFalse-reds*:
$\langle if\ (b)\ c_1\ else\ c_2,s\rangle \rightarrow* \langle Skip,s'\rangle \Longrightarrow [\![b]\!]\ s = Some\ false \Longrightarrow \langle c_2,s\rangle \rightarrow* \langle Skip,s'\rangle$
⟨*proof*⟩

lemma *WhileFalse-reds*:
$\langle while\ (b)\ cx,s\rangle \rightarrow* \langle Skip,s'\rangle \Longrightarrow [\![b]\!]\ s = Some\ false \Longrightarrow s = s'$
⟨*proof*⟩

lemma *WhileTrue-reds*:
$\langle while\ (b)\ cx,s\rangle \rightarrow* \langle Skip,s'\rangle \Longrightarrow [\![b]\!]\ s = Some\ true$
$\Longrightarrow \exists\, sx.\ \langle cx,s\rangle \rightarrow* \langle Skip,sx\rangle \wedge \langle while\ (b)\ cx,sx\rangle \rightarrow* \langle Skip,s'\rangle$
⟨*proof*⟩

lemma *While-True-or-False*:
$\langle while\ (b)\ com,s\rangle \rightarrow* \langle Skip,s'\rangle \Longrightarrow [\![b]\!]\ s = Some\ true \vee [\![b]\!]\ s = Some\ false$
⟨*proof*⟩

inductive *red-n* :: *com* ⇒ *state* ⇒ *nat* ⇒ *com* ⇒ *state* ⇒ *bool*
$(\langle((1\langle\text{-},/\text{-}\rangle) \rightarrow^{\text{-}} (1\langle\text{-},/\text{-}\rangle))\rangle\ [0,0,0,0,0]\ 81)$
where *red-n-Base*: $\langle c,s\rangle \rightarrow^{0} \langle c,s\rangle$
| *red-n-Rec*: $[\![\langle c,s\rangle \rightarrow \langle c'',s''\rangle;\ \langle c'',s''\rangle \rightarrow^{n} \langle c',s'\rangle]\!] \Longrightarrow \langle c,s\rangle \rightarrow^{Suc\ n} \langle c',s'\rangle$

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

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

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

lemma *reds-to-red-n*:$\langle c,s\rangle \to* \langle c',s'\rangle \Longrightarrow \exists\, n.\ \langle c,s\rangle \to^n \langle c',s'\rangle$
⟨*proof*⟩

lemma *red-n-to-reds*:$\langle c,s\rangle \to^n \langle c',s'\rangle \Longrightarrow \langle c,s\rangle \to* \langle c',s'\rangle$
⟨*proof*⟩

lemma *while-reds-induct*[*consumes 1*, *case-names false true*]:
$\llbracket\langle while\ (b)\ cx,s\rangle \to* \langle Skip,s'\rangle;\ \bigwedge s.\ \llbracket b\rrbracket\ s = Some\ false \Longrightarrow P\ s\ s;$
$\bigwedge s\ s''.\ \llbracket\ \llbracket b\rrbracket\ s = Some\ true;\ \langle cx,s\rangle \to* \langle Skip,s''\rangle;$
$\langle while\ (b)\ cx,s''\rangle \to* \langle Skip,s'\rangle;\ P\ s''\ s'\rrbracket \Longrightarrow P\ s\ s'\rrbracket$
$\Longrightarrow P\ s\ s'$
⟨*proof*⟩

lemma *red-det*:
$\llbracket\langle c,s\rangle \to \langle c_1,s_1\rangle;\ \langle c,s\rangle \to \langle c_2,s_2\rangle\rrbracket \Longrightarrow c_1 = c_2 \wedge s_1 = s_2$
⟨*proof*⟩

theorem *reds-det*:
$\llbracket\langle c,s\rangle \to* \langle Skip,s_1\rangle;\ \langle c,s\rangle \to* \langle Skip,s_2\rangle\rrbracket \Longrightarrow s_1 = s_2$
⟨*proof*⟩

end
theory *secTypes*
imports *Semantics*
begin

2 Security types

2.1 Security definitions

datatype *secLevel* = *Low* | *High*

type-synonym *typeEnv* = *vname* ⇀ *secLevel*

inductive *secExprTyping* :: *typeEnv* ⇒ *expr* ⇒ *secLevel* ⇒ *bool* (‹- ⊢ - : -›)
where *typeVal*: Γ ⊢ *Val V* : *lev*

| *typeVar*: Γ *Vn* = *Some lev* ⟹ Γ ⊢ *Var Vn* : *lev*

| *typeBinOp1*: ⟦Γ ⊢ *e1* : *Low*; Γ ⊢ *e2* : *Low*⟧ ⟹ Γ ⊢ *e1* «*bop*» *e2* : *Low*

| *typeBinOp2*: ⟦Γ ⊢ *e1* : *High*; Γ ⊢ *e2* : *lev*⟧ ⟹ Γ ⊢ *e1* «*bop*» *e2* : *High*

| *typeBinOp3*: ⟦Γ ⊢ *e1* : *lev*; Γ ⊢ *e2* : *High*⟧ ⟹ Γ ⊢ *e1* «*bop*» *e2* : *High*

inductive *secComTyping* :: *typeEnv* ⇒ *secLevel* ⇒ *com* ⇒ *bool* (‹-,- ⊢ -›)
where *typeSkip*: Γ,*T* ⊢ *Skip*

| *typeAssH*: Γ *V* = *Some High* ⟹ Γ,*T* ⊢ *V* := *e*

| *typeAssL*: ⟦Γ ⊢ *e* : *Low*; Γ *V* = *Some Low*⟧ ⟹ Γ,*Low* ⊢ *V* := *e*

| *typeSeq*: ⟦Γ,*T* ⊢ *c1*; Γ,*T* ⊢ *c2*⟧ ⟹ Γ,*T* ⊢ *c1*;;*c2*

| *typeWhile*: ⟦Γ ⊢ *b* : *T*; Γ,*T* ⊢ *c*⟧ ⟹ Γ,*T* ⊢ *while* (*b*) *c*

| *typeIf*: ⟦Γ ⊢ *b* : *T*; Γ,*T* ⊢ *c1*; Γ,*T* ⊢ *c2*⟧ ⟹ Γ,*T* ⊢ *if* (*b*) *c1 else c2*

| *typeConvert*: Γ,*High* ⊢ *c* ⟹ Γ,*Low* ⊢ *c*

2.2 Lemmas concerning expressions

lemma *exprTypeable*:
assumes *fv e* ⊆ *dom* Γ **obtains** *T* **where** Γ ⊢ *e* : *T*
⟨*proof*⟩

lemma *exprBinopTypeable*:
assumes $\Gamma \vdash e1 \ «bop» \ e2 : T$
shows $(\exists T1.\ \Gamma \vdash e1 : T1) \wedge (\exists T2.\ \Gamma \vdash e2 : T2)$
⟨*proof*⟩

lemma *exprTypingHigh*:
assumes $\Gamma \vdash e : T$ **and** $x \in fv\ e$ **and** $\Gamma\ x = Some\ High$
shows $\Gamma \vdash e : High$
⟨*proof*⟩

lemma *exprTypingLow*:
assumes $\Gamma \vdash e : Low$ **and** $x \in fv\ e$ **shows** $\Gamma\ x = Some\ Low$
⟨*proof*⟩

lemma *typeableFreevars*:
assumes $\Gamma \vdash e : T$ **shows** $fv\ e \subseteq dom\ \Gamma$
⟨*proof*⟩

lemma *exprNotNone*:
assumes $\Gamma \vdash e : T$ **and** $fv\ e \subseteq dom\ s$
shows $[\![e]\!]\ s \neq None$
⟨*proof*⟩

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. $\in dom\ state$), 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 transitiv (see lemmas) hence an equivalence relation.

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

lemma *lowEquivReflexive*: $\Gamma \vdash s1 \approx_L s1$
⟨*proof*⟩

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

⟨*proof*⟩

lemma *lowEquivTransitive*:
⟦Γ ⊢ *s1* ≈$_L$ *s2*; Γ ⊢ *s2* ≈$_L$ *s3*⟧ ⟹ Γ ⊢ *s1* ≈$_L$ *s3*
⟨*proof*⟩

2.3.2 Non Interference

definition *nonInterference* :: *typeEnv* ⇒ *com* ⇒ *bool*
where *nonInterference* Γ *c* ≡
(∀ *s1 s2 s1′ s2′*. (Γ ⊢ *s1* ≈$_L$ *s2* ∧ ⟨*c*,*s1*⟩ →∗ ⟨*Skip*,*s1′*⟩ ∧ ⟨*c*,*s2*⟩ →∗ ⟨*Skip*,*s2′*⟩)
⟶ Γ ⊢ *s1′* ≈$_L$ *s2′*)

lemma *nonInterferenceI*:
⟦⋀*s1 s2 s1′ s2′*. ⟦Γ ⊢ *s1* ≈$_L$ *s2*; ⟨*c*,*s1*⟩ →∗ ⟨*Skip*,*s1′*⟩; ⟨*c*,*s2*⟩ →∗ ⟨*Skip*,*s2′*⟩⟧
⟹ Γ ⊢ *s1′* ≈$_L$ *s2′*⟧ ⟹ *nonInterference* Γ *c*
⟨*proof*⟩

lemma *interpretLow*:
assumes Γ ⊢ *s1* ≈$_L$ *s2* **and** *all*:∀ *V*∈*fv e*. Γ *V* = *Some Low*
shows ⟦*e*⟧ *s1* = ⟦*e*⟧ *s2*
⟨*proof*⟩

lemma *interpretLow2*:
assumes Γ ⊢ *e* : *Low* **and** Γ ⊢ *s1* ≈$_L$ *s2* **shows** ⟦*e*⟧ *s1* = ⟦*e*⟧ *s2*
⟨*proof*⟩

lemma *assignNIhighlemma*:
assumes Γ ⊢ *s1* ≈$_L$ *s2* **and** Γ *V* = *Some High* **and** *s1′* = *s1*(*V*:= ⟦*e*⟧ *s1*)
and *s2′* = *s2*(*V*:= ⟦*e*⟧ *s2*)
shows Γ ⊢ *s1′* ≈$_L$ *s2′*
⟨*proof*⟩

lemma *assignNIlowlemma*:
assumes Γ ⊢ *s1* ≈$_L$ *s2* **and** Γ *V* = *Some Low* **and** Γ ⊢ *e* : *Low*
and *s1′* = *s1*(*V*:= ⟦*e*⟧ *s1*) **and** *s2′* = *s2*(*V*:= ⟦*e*⟧ *s2*)
shows Γ ⊢ *s1′* ≈$_L$ *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* Γ *c1* **and** *nonInterference* Γ *c2*
shows *nonInterference* Γ (*c1*;;*c2*)

⟨*proof*⟩

lemma *WhileStepInduct*:
assumes *while*:⟨*while* (*b*) *c*,*s1*⟩ →∗ ⟨*Skip*,*s2*⟩
and *body*:⋀*s1 s2*. ⟨*c*,*s1*⟩ →∗ ⟨*Skip*,*s2*⟩ ⟹ Γ ⊢ *s1* ≈$_L$ *s2* **and** Γ,*High* ⊢ *c*
shows Γ ⊢ *s1* ≈$_L$ *s2*
⟨*proof*⟩

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 Γ,*High* ⊢ *c* **and** ⟨*c*,*s1*⟩ →∗ ⟨*Skip*,*s2*⟩
shows Γ ⊢ *s1* ≈$_L$ *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
⟨*proof*⟩

lemma *CondHighCompositionality*:
assumes Γ,*High* ⊢ *if* (*b*) *c1 else c2*
shows *nonInterference* Γ (*if* (*b*) *c1 else c2*)
⟨*proof*⟩

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

lemma *WhileHighCompositionality*:
assumes Γ,*High* ⊢ *while* (*b*) *c′*
shows *nonInterference* Γ (*while* (*b*) *c′*)
⟨*proof*⟩

lemma *WhileLowStepInduct*:
assumes *while1*: ⟨*while* (*b*) *c′*,*s1*⟩ →∗ ⟨*Skip*,*s1′*⟩
and *while2*: ⟨*while* (*b*) *c′*,*s2*⟩→∗⟨*Skip*,*s2′*⟩
and Γ ⊢ *b* : *Low*
and *body*:⋀*s1 s1′ s2 s2′*. ⟦⟨*c′*,*s1*⟩ →∗ ⟨*Skip*,*s1′*⟩; ⟨*c′*,*s2*⟩ →∗ ⟨*Skip*,*s2′*⟩;
Γ ⊢ *s1* ≈$_L$ *s2*⟧ ⟹ Γ ⊢ *s1′* ≈$_L$ *s2′*
and *le*: Γ ⊢ *s1* ≈$_L$ *s2*

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

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

and now: the main theorem:

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

end

theory *Execute*
imports *secTypes*
begin

3 Executing the small step semantics

code-pred (*modes*: *i* => *o* => *bool as exec-red*, *i* => *i* ∗ *o* => *bool*, *i* => *o* ∗ *i* => *bool*, *i* => *i* => *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*] *typeBinOp3*[**where** *lev* = *Low*]

code-pred (*modes*: *i* => *i* => *o* => *bool as compute-secExprTyping*,
i => *i* => *i* => *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* => *o* => *i* => *bool as compute-secComTyping*,
i => *i* => *i* => *bool as check-secComTyping*) *secComTyping*
⟨*proof*⟩

thm *secComTyping.equation*

3.1 An example taken from Volpano, Smith, Irvine

definition *com* = *if* (*Var* ''*x*'' «*Eq*» *Val* (*Intg 1*)) (''*y*'' := *Val* (*Intg 1*)) *else* (''*y*'' := *Val* (*Intg 0*))
definition *Env* = *map-of* [(''*x*'', *High*), (''*y*'', *High*)]

values {*T*. *Env* ⊢ (*Var* ''*x*'' «*Eq*» *Val* (*Intg 1*)): *T*}
value *Env*, *High* ⊢ *com*
value *Env*, *Low* ⊢ *com*
values *1* {*T*. *Env*, *T* ⊢ *com*}

definition *Env'* = *map-of* [(''*x*'', *Low*), (''*y*'', *High*)]

value *Env'*, *Low* ⊢ *com*
value *Env'*, *High* ⊢ *com*
values *1* {*T*. *Env*, *T* ⊢ *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.