Slicing Guarantees Information Flow Noninterference

Daniel Wasserrab

October 13, 2025

Abstract

In this contribution, we show how correctness proofs for intra- [8] and interprocedural slicing [9] can be used to prove that slicing is able to guarantee information flow noninterference. Moreover, we also illustrate how to lift the control flow graphs of the respective frameworks such that they fulfil the additional assumptions needed in the noninterference proofs. A detailed description of the intraprocedural proof and its interplay with the slicing framework can be found in [10].

1 Introduction

Information Flow Control (IFC) encompasses algorithms which determines if a given program leaks secret information to public entities. The major group are so called IFC type systems, where well-typed means that the respective program is secure. Several IFC type systems have been verified in proof assistants, e.g. see [1, 2, 5, 3, 7].

However, type systems have some drawbacks which can lead to false alarms. To overcome this problem, an IFC approach basing on slicing has been developed [4], which can significantly reduce the amount of false alarms. This contribution presents the first machine-checked proof that slicing is able to guarantee IFC noninterference. It bases on previously published machine-checked correctness proofs for slicing [8, 9]. Details for the intraprocedural case can be found in [10].

2 Slicing guarantees IFC Noninterference

theory *NonInterferenceIntra* **imports**
Slicing.Slice
Slicing.CFGExit-wf
begin

2.1 Assumptions of this Approach

Classical IFC noninterference, a special case of a noninterference definition using partial equivalence relations (per) [6], partitions the variables (i.e. locations) into security levels. Usually, only levels for secret or high, written H, and public or low, written L, variables are used. Basically, a program that is noninterferent has to fulfil one basic property: executing the program in two different initial states that may differ in the values of their H-variables yields two final states that again only differ in the values of their H-variables; thus the values of the H-variables did not influence those of the L-variables.

Every per-based approach makes certain assumptions: (i) all H-variables are defined at the beginning of the program, (ii) all L-variables are observed (or used in our terms) at the end and (iii) every variable is either H or L. This security label is fixed for a variable and can not be altered during a program run. Thus, we have to extend the prerequisites of the slicing framework in [8] accordingly in a new locale:

locale *NonInterferenceIntraGraph* =
BackwardSlice sourcenode targetnode kind valid-edge Entry Def Use state-val backward-slice +
CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (‹'('-Entry'-')›) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *backward-slice* :: *'node set* ⇒ *'node set*
and *Exit* :: *'node* (‹'('-Exit'-')›) +
fixes *H* :: *'var set*
fixes *L* :: *'var set*
fixes *High* :: *'node* (‹'('-High'-')›)
fixes *Low* :: *'node* (‹'('-Low'-')›)
assumes *Entry-edge-Exit-or-High*:
⟦*valid-edge a*; *sourcenode a* = (-*Entry*-)⟧
⟹ *targetnode a* = (-*Exit*-) ∨ *targetnode a* = (-*High*-)
and *High-target-Entry-edge*:
∃ *a*. *valid-edge a* ∧ *sourcenode a* = (-*Entry*-) ∧ *targetnode a* = (-*High*-) ∧
kind a = $(\lambda s.\ True)_{\surd}$
and *Entry-predecessor-of-High*:
⟦*valid-edge a*; *targetnode a* = (-*High*-)⟧ ⟹ *sourcenode a* = (-*Entry*-)
and *Exit-edge-Entry-or-Low*: ⟦*valid-edge a*; *targetnode a* = (-*Exit*-)⟧
⟹ *sourcenode a* = (-*Entry*-) ∨ *sourcenode a* = (-*Low*-)
and *Low-source-Exit-edge*:
∃ *a*. *valid-edge a* ∧ *sourcenode a* = (-*Low*-) ∧ *targetnode a* = (-*Exit*-) ∧
kind a = $(\lambda s.\ True)_{\surd}$
and *Exit-successor-of-Low*:
⟦*valid-edge a*; *sourcenode a* = (-*Low*-)⟧ ⟹ *targetnode a* = (-*Exit*-)
and *DefHigh*: *Def* (-*High*-) = *H*
and *UseHigh*: *Use* (-*High*-) = *H*

and *UseLow*: *Use* (*-Low-*) = *L*
and *HighLowDistinct*: $H \cap L = \{\}$
and *HighLowUNIV*: $H \cup L = UNIV$

begin

lemma *Low-neq-Exit*: **assumes** $L \neq \{\}$ **shows** (*-Low-*) $\neq$ (*-Exit-*)
⟨*proof*⟩

lemma *Entry-path-High-path*:
assumes (*-Entry-*) $-as\rightarrow*$ *n* **and** *inner-node n*
obtains a' as' **where** $as = a'\#as'$ **and** (*-High-*) $-as'\rightarrow*$ *n*
and *kind* $a' = (\lambda s.\ True)_{\surd}$
⟨*proof*⟩

lemma *Exit-path-Low-path*:
assumes *n* $-as\rightarrow*$ (*-Exit-*) **and** *inner-node n*
obtains a' as' **where** $as = as'@[a']$ **and** *n* $-as'\rightarrow*$ (*-Low-*)
and *kind* $a' = (\lambda s.\ True)_{\surd}$
⟨*proof*⟩

lemma *not-Low-High*: $V \notin L \Longrightarrow V \in H$
⟨*proof*⟩

lemma *not-High-Low*: $V \notin H \Longrightarrow V \in L$
⟨*proof*⟩

2.2 Low Equivalence

In classical noninterference, an external observer can only see public values, in our case the L-variables. If two states agree in the values of all L-variables, these states are indistinguishable for him. *Low equivalence* groups those states in an equivalence class using the relation $\approx_L$:

definition *lowEquivalence* :: *'state* $\Rightarrow$ *'state* $\Rightarrow$ *bool* (**infixl** ‹$\approx_L$› *50*)
where $s \approx_L s' \equiv \forall\ V \in L.\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V$

The following lemmas connect low equivalent states with relevant variables as necessary in the correctness proof for slicing.

lemma *relevant-vars-Entry*:
assumes $V \in rv\ S$ (*-Entry-*) **and** (*-High-*) $\notin$ *backward-slice S*
shows $V \in L$
⟨*proof*⟩

lemma *lowEquivalence-relevant-nodes-Entry*:
assumes $s \approx_L s'$ **and** $(\text{-}High\text{-}) \notin$ *backward-slice S*
shows $\forall\, V \in rv\ S\ (\text{-}Entry\text{-}).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V$
⟨*proof*⟩

lemma *rv-Low-Use-Low*:
assumes $(\text{-}Low\text{-}) \in S$
shows $[\![n\ {-as}{\rightarrow}{*}\ (\text{-}Low\text{-});\ n\ {-as'}{\rightarrow}{*}\ (\text{-}Low\text{-});$
$\forall\, V \in rv\ S\ n.\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V;$
$preds\ (slice\text{-}kinds\ S\ as)\ s;\ preds\ (slice\text{-}kinds\ S\ as')\ s']\!]$
$\Longrightarrow \forall\, V \in Use\ (\text{-}Low\text{-}).\ state\text{-}val\ (transfers\ (slice\text{-}kinds\ S\ as)\ s)\ V =$
$state\text{-}val\ (transfers\ (slice\text{-}kinds\ S\ as')\ s')\ V$
⟨*proof*⟩

2.3 The Correctness Proofs

In the following, we present two correctness proofs that slicing guarantees IFC noninterference. In both theorems, $(\text{-}High\text{-}) \notin$ *backward-slice S*, where $(\text{-}Low\text{-}) \in S$, makes sure that no high variable (which are all defined in $(\text{-}High\text{-})$) can influence a low variable (which are all used in $(\text{-}Low\text{-})$).

First, a theorem regarding $(\text{-}Entry\text{-})\ {-as}{\rightarrow}{*}\ (\text{-}Exit\text{-})$ paths in the control flow graph (CFG), which agree to a complete program execution:

lemma *nonInterference-path-to-Low*:
assumes $s \approx_L s'$ **and** $(\text{-}High\text{-}) \notin$ *backward-slice S* **and** $(\text{-}Low\text{-}) \in S$
and $(\text{-}Entry\text{-})\ {-as}{\rightarrow}{*}\ (\text{-}Low\text{-})$ **and** $preds\ (kinds\ as)\ s$
and $(\text{-}Entry\text{-})\ {-as'}{\rightarrow}{*}\ (\text{-}Low\text{-})$ **and** $preds\ (kinds\ as')\ s'$
shows $transfers\ (kinds\ as)\ s \approx_L transfers\ (kinds\ as')\ s'$
⟨*proof*⟩

theorem *nonInterference-path*:
assumes $s \approx_L s'$ **and** $(\text{-}High\text{-}) \notin$ *backward-slice S* **and** $(\text{-}Low\text{-}) \in S$
and $(\text{-}Entry\text{-})\ {-as}{\rightarrow}{*}\ (\text{-}Exit\text{-})$ **and** $preds\ (kinds\ as)\ s$
and $(\text{-}Entry\text{-})\ {-as'}{\rightarrow}{*}\ (\text{-}Exit\text{-})$ **and** $preds\ (kinds\ as')\ s'$
shows $transfers\ (kinds\ as)\ s \approx_L transfers\ (kinds\ as')\ s'$
⟨*proof*⟩

end

The second theorem assumes that we have a operational semantics, whose evaluations are written $\langle c,s\rangle \Rightarrow \langle c',s'\rangle$ and which conforms to the CFG. The correctness theorem then states that if no high variable influenced a low variable and the initial states were low equivalent, the reulting states are again low equivalent:

locale *NonInterferenceIntra* =

NonInterferenceIntraGraph sourcenode targetnode kind valid-edge Entry Def Use state-val backward-slice Exit H L High Low +
BackwardSlice-wf sourcenode targetnode kind valid-edge Entry Def Use state-val backward-slice sem identifies
for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
and *kind* :: *'edge* ⇒ *'state edge-kind* **and** *valid-edge* :: *'edge* ⇒ *bool*
and *Entry* :: *'node* (‹'('-Entry'-')›) **and** *Def* :: *'node* ⇒ *'var set*
and *Use* :: *'node* ⇒ *'var set* **and** *state-val* :: *'state* ⇒ *'var* ⇒ *'val*
and *backward-slice* :: *'node set* ⇒ *'node set*
and *sem* :: *'com* ⇒ *'state* ⇒ *'com* ⇒ *'state* ⇒ *bool*
(‹((1⟨-,/-⟩) ⇒/ (1⟨-,/-⟩))› [0,0,0,0] 81)
and *identifies* :: *'node* ⇒ *'com* ⇒ *bool* (‹- ≜ -› [51, 0] 80)
and *Exit* :: *'node* (‹'('-Exit'-')›)
and *H* :: *'var set* **and** *L* :: *'var set*
and *High* :: *'node* (‹'('-High'-')›) **and** *Low* :: *'node* (‹'('-Low'-')›) +
fixes *final* :: *'com* ⇒ *bool*
assumes *final-edge-Low*: ⟦*final c*; *n* ≜ *c*⟧
⟹ ∃ *a*. *valid-edge a* ∧ *sourcenode a* = *n* ∧ *targetnode a* = (-*Low*-) ∧ *kind a* = ⇑*id*
begin

The following theorem needs the explicit edge from (-*High*-) to *n*. An approach using a *init* predicate for initial statements, being reachable from (-*High*-) via a $(\lambda s.\ True)_{\surd}$ edge, does not work as the same statement could be identified by several nodes, some initial, some not. E.g., in the program `while (True) Skip;;Skip` two nodes identify this inital statement: the initial node and the node within the loop (because of loop unrolling).

theorem *nonInterference*:
assumes $s_1 \approx_L s_2$ **and** (-*High*-) ∉ *backward-slice S* **and** (-*Low*-) ∈ *S*
and *valid-edge a* **and** *sourcenode a* = (-*High*-) **and** *targetnode a* = *n*
and *kind a* = $(\lambda s.\ True)_{\surd}$ **and** *n* ≜ *c* **and** *final c'*
and $\langle c,s_1\rangle \Rightarrow \langle c',s_1'\rangle$ **and** $\langle c,s_2\rangle \Rightarrow \langle c',s_2'\rangle$
shows $s_1' \approx_L s_2'$
⟨*proof*⟩

end

end

3 Framework Graph Lifting for Noninterference

theory *LiftingIntra*
imports *NonInterferenceIntra Slicing.CDepInstantiations*
begin

In this section, we show how a valid CFG from the slicing framework in [8] can be lifted to fulfil all properties of the *NonInterferenceIntraGraph* locale. Basically, we redefine the hitherto existing *Entry* and *Exit* nodes as new *High* and *Low* nodes, and introduce two new nodes *NewEntry* and *NewExit*. Then, we have to lift all functions to operate on this new graph.

3.1 Liftings

3.1.1 The datatypes

datatype *'node LDCFG-node* = *Node 'node*
| *NewEntry*
| *NewExit*

type-synonym (*'edge*,*'node*,*'state*) *LDCFG-edge* =
'node LDCFG-node $\times$ (*'state edge-kind*) $\times$ *'node LDCFG-node*

3.1.2 Lifting *valid-edge*

inductive *lift-valid-edge* :: (*'edge* $\Rightarrow$ *bool*) $\Rightarrow$ (*'edge* $\Rightarrow$ *'node*) $\Rightarrow$ (*'edge* $\Rightarrow$ *'node*) $\Rightarrow$
(*'edge* $\Rightarrow$ *'state edge-kind*) $\Rightarrow$ *'node* $\Rightarrow$ *'node* $\Rightarrow$ (*'edge*,*'node*,*'state*) *LDCFG-edge* $\Rightarrow$
bool
for *valid-edge*::*'edge* $\Rightarrow$ *bool* **and** *src*::*'edge* $\Rightarrow$ *'node* **and** *trg*::*'edge* $\Rightarrow$ *'node*
and *knd*::*'edge* $\Rightarrow$ *'state edge-kind* **and** *E*::*'node* **and** *X*::*'node*

where *lve-edge*:
$[\![$*valid-edge a*; *src a* $\neq$ *E* $\vee$ *trg a* $\neq$ *X*;
e = (*Node* (*src a*),*knd a*,*Node* (*trg a*))$]\!]$
$\Longrightarrow$ *lift-valid-edge valid-edge src trg knd E X e*

| *lve-Entry-edge*:
e = (*NewEntry*,(λ*s. True*)$_{\surd}$,*Node E*)
$\Longrightarrow$ *lift-valid-edge valid-edge src trg knd E X e*

| *lve-Exit-edge*:
e = (*Node X*,(λ*s. True*)$_{\surd}$,*NewExit*)
$\Longrightarrow$ *lift-valid-edge valid-edge src trg knd E X e*

| *lve-Entry-Exit-edge*:
e = (*NewEntry*,(λ*s. False*)$_{\surd}$,*NewExit*)
$\Longrightarrow$ *lift-valid-edge valid-edge src trg knd E X e*

lemma [*simp*]:$\neg$ *lift-valid-edge valid-edge src trg knd E X* (*Node E*,*et*,*Node X*)
$\langle$*proof*$\rangle$

3.1.3 Lifting *Def* and *Use* sets

inductive-set *lift-Def-set* :: (*'node* $\Rightarrow$ *'var set*) $\Rightarrow$ *'node* $\Rightarrow$ *'node* $\Rightarrow$

$'var\ set \Rightarrow 'var\ set \Rightarrow ('node\ LDCFG\text{-}node \times 'var)\ set$
for Def::$('node \Rightarrow 'var\ set)$ **and** E::$'node$ **and** X::$'node$
and H::$'var\ set$ **and** L::$'var\ set$

where *lift-Def-node*:
$V \in Def\ n \Longrightarrow (Node\ n, V) \in lift\text{-}Def\text{-}set\ Def\ E\ X\ H\ L$

| *lift-Def-High*:
$V \in H \Longrightarrow (Node\ E, V) \in lift\text{-}Def\text{-}set\ Def\ E\ X\ H\ L$

abbreviation *lift-Def* :: $('node \Rightarrow 'var\ set) \Rightarrow 'node \Rightarrow 'node \Rightarrow$
$'var\ set \Rightarrow 'var\ set \Rightarrow 'node\ LDCFG\text{-}node \Rightarrow 'var\ set$
where $lift\text{-}Def\ Def\ E\ X\ H\ L\ n \equiv \{V.\ (n,V) \in lift\text{-}Def\text{-}set\ Def\ E\ X\ H\ L\}$

inductive-set *lift-Use-set* :: $('node \Rightarrow 'var\ set) \Rightarrow 'node \Rightarrow 'node \Rightarrow$
$'var\ set \Rightarrow 'var\ set \Rightarrow ('node\ LDCFG\text{-}node \times 'var)\ set$
for Use::$'node \Rightarrow 'var\ set$ **and** E::$'node$ **and** X::$'node$
and H::$'var\ set$ **and** L::$'var\ set$

where
lift-Use-node:
$V \in Use\ n \Longrightarrow (Node\ n, V) \in lift\text{-}Use\text{-}set\ Use\ E\ X\ H\ L$

| *lift-Use-High*:
$V \in H \Longrightarrow (Node\ E, V) \in lift\text{-}Use\text{-}set\ Use\ E\ X\ H\ L$

| *lift-Use-Low*:
$V \in L \Longrightarrow (Node\ X, V) \in lift\text{-}Use\text{-}set\ Use\ E\ X\ H\ L$

abbreviation *lift-Use* :: $('node \Rightarrow 'var\ set) \Rightarrow 'node \Rightarrow 'node \Rightarrow$
$'var\ set \Rightarrow 'var\ set \Rightarrow 'node\ LDCFG\text{-}node \Rightarrow 'var\ set$
where $lift\text{-}Use\ Use\ E\ X\ H\ L\ n \equiv \{V.\ (n,V) \in lift\text{-}Use\text{-}set\ Use\ E\ X\ H\ L\}$

3.2 The lifting lemmas

3.2.1 Lifting the basic locales

abbreviation *src* :: $('edge,'node,'state)\ LDCFG\text{-}edge \Rightarrow 'node\ LDCFG\text{-}node$
where $src\ a \equiv fst\ a$

abbreviation *trg* :: $('edge,'node,'state)\ LDCFG\text{-}edge \Rightarrow 'node\ LDCFG\text{-}node$
where $trg\ a \equiv snd(snd\ a)$

definition *knd* :: $('edge,'node,'state)\ LDCFG\text{-}edge \Rightarrow 'state\ edge\text{-}kind$
where $knd\ a \equiv fst(snd\ a)$

lemma *lift-CFG*:

assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
shows *CFG src trg* (*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
⟨*proof*⟩

lemma *lift-CFG-wf*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
shows *CFG-wf src trg knd* (*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry* (*lift-Def Def Entry Exit H L*) (*lift-Use Use Entry Exit H L*) *state-val*
⟨*proof*⟩

lemma *lift-CFGExit*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
shows *CFGExit src trg knd* (*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry NewExit*
⟨*proof*⟩

lemma *lift-CFGExit-wf*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
shows *CFGExit-wf src trg knd* (*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry* (*lift-Def Def Entry Exit H L*) (*lift-Use Use Entry Exit H L*) *state-val NewExit*
⟨*proof*⟩

3.2.2 Lifting *wod-backward-slice*

lemma *lift-wod-backward-slice*:
fixes *valid-edge* **and** *sourcenode* **and** *targetnode* **and** *kind* **and** *Entry* **and** *Exit* **and** *Def* **and** *Use* **and** *H* **and** *L*
defines *lve*:*lve* ≡ *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*
and *lDef*:*lDef* ≡ *lift-Def Def Entry Exit H L*
and *lUse*:*lUse* ≡ *lift-Use Use Entry Exit H L*
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
and $H \cap L = \{\}$ **and** $H \cup L = UNIV$
shows *NonInterferenceIntraGraph src trg knd lve NewEntry lDef lUse state-val* (*CFG-wf.wod-backward-slice src trg lve lDef lUse*) *NewExit H L* (*Node Entry*) (*Node Exit*)
⟨*proof*⟩

3.2.3 Lifting *PDG-BS* with *standard-control-dependence*

lemma *lift-Postdomination*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
and *pd*:*Postdomination sourcenode targetnode kind valid-edge Entry Exit*
and *inner*:*CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx*
shows *Postdomination src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry NewExit*
⟨*proof*⟩

lemma *lift-PDG-scd*:
assumes *PDG*:*PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
(*Postdomination.standard-control-dependence sourcenode targetnode valid-edge Exit*)
and *pd*:*Postdomination sourcenode targetnode kind valid-edge Entry Exit*
and *inner*:*CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx*
shows *PDG src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
(*lift-Def Def Entry Exit H L*) (*lift-Use Use Entry Exit H L*) *state-val NewExit*
(*Postdomination.standard-control-dependence src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewExit*)
⟨*proof*⟩

lemma *lift-PDG-standard-backward-slice*:
fixes *valid-edge* **and** *sourcenode* **and** *targetnode* **and** *kind* **and** *Entry* **and** *Exit*
and *Def* **and** *Use* **and** *H* **and** *L*
defines *lve*:*lve* ≡ *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*
and *lDef*:*lDef* ≡ *lift-Def Def Entry Exit H L*
and *lUse*:*lUse* ≡ *lift-Use Use Entry Exit H L*
assumes *PDG*:*PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
(*Postdomination.standard-control-dependence sourcenode targetnode valid-edge Exit*)
and *pd*:*Postdomination sourcenode targetnode kind valid-edge Entry Exit*
and *inner*:*CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx*
and $H \cap L = \{\}$ **and** $H \cup L = UNIV$
shows *NonInterferenceIntraGraph src trg knd lve NewEntry lDef lUse state-val*
(*PDG.PDG-BS src trg lve lDef lUse*
(*Postdomination.standard-control-dependence src trg lve NewExit*))
NewExit H L (*Node Entry*) (*Node Exit*)
⟨*proof*⟩

3.2.4 Lifting *PDG-BS* with *weak-control-dependence*

lemma *lift-StrongPostdomination*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use*

state-val Exit
and *spd*:*StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit*
and *inner*:*CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx*
shows *StrongPostdomination src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry NewExit*
⟨*proof*⟩

lemma *lift-PDG-wcd*:
assumes *PDG*:*PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
(*StrongPostdomination.weak-control-dependence sourcenode targetnode valid-edge Exit*)
and *spd*:*StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit*
and *inner*:*CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx*
shows *PDG src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
(*lift-Def Def Entry Exit H L*) (*lift-Use Use Entry Exit H L*) *state-val NewExit*
(*StrongPostdomination.weak-control-dependence src trg*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewExit*)
⟨*proof*⟩

lemma *lift-PDG-weak-backward-slice*:
fixes *valid-edge* **and** *sourcenode* **and** *targetnode* **and** *kind* **and** *Entry* **and** *Exit*
and *Def* **and** *Use* **and** *H* **and** *L*
defines *lve*:*lve* ≡ *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*
and *lDef*:*lDef* ≡ *lift-Def Def Entry Exit H L*
and *lUse*:*lUse* ≡ *lift-Use Use Entry Exit H L*
assumes *PDG*:*PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit*
(*StrongPostdomination.weak-control-dependence sourcenode targetnode valid-edge Exit*)
and *spd*:*StrongPostdomination sourcenode targetnode kind valid-edge Entry Exit*
and *inner*:*CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx*
and $H \cap L = \{\}$ **and** $H \cup L = UNIV$
shows *NonInterferenceIntraGraph src trg knd lve NewEntry lDef lUse state-val*
(*PDG.PDG-BS src trg lve lDef lUse*
(*StrongPostdomination.weak-control-dependence src trg lve NewExit*))
NewExit H L (*Node Entry*) (*Node Exit*)
⟨*proof*⟩

end

4 Information Flow for While

theory *NonInterferenceWhile* **imports**
Slicing.SemanticsWellFormed
Slicing.StaticControlDependences
LiftingIntra
begin

locale *SecurityTypes* =
fixes *H* :: *vname set*
fixes *L* :: *vname set*
assumes *HighLowDistinct*: $H \cap L = \{\}$
and *HighLowUNIV*: $H \cup L = UNIV$
begin

4.1 Lifting *labels-nodes* and Defining *final*

fun *labels-LDCFG-nodes* :: *cmd* ⇒ *w-node LDCFG-node* ⇒ *cmd* ⇒ *bool*
where *labels-LDCFG-nodes prog* (*Node n*) *c* = *labels-nodes prog n c*
| *labels-LDCFG-nodes prog n c* = *False*

lemmas *WCFG-path-induct*[*consumes 1*, *case-names empty-path Cons-path*]
= *CFG.path.induct*[*OF While-CFG-aux*]

lemma *lift-valid-node*:
assumes *CFG.valid-node sourcenode targetnode* (*valid-edge prog*) *n*
shows *CFG.valid-node src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind* (*-Entry-*) (*-Exit-*))
(*Node n*)
⟨*proof*⟩

lemma *lifted-CFG-fund-prop*:
assumes *labels-LDCFG-nodes prog n c* **and** ⟨*c*,*s*⟩ →∗ ⟨*c'*,*s'*⟩
shows ∃ *n' as*. *CFG.path src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind* (*-Entry-*) (*-Exit-*))
n as n' ∧ *transfers* (*CFG.kinds knd as*) *s* = *s'* ∧
preds (*CFG.kinds knd as*) *s* ∧ *labels-LDCFG-nodes prog n' c'*
⟨*proof*⟩

fun *final* :: *cmd* ⇒ *bool*
where *final Skip* = *True*
| *final c* = *False*

lemma *final-edge*:
labels-nodes prog n Skip $\Longrightarrow$ *prog* $\vdash$ *n* $-\Uparrow id\rightarrow$ (*-Exit-*)
⟨*proof*⟩

4.2 Semantic Non-Interference for Weak Order Dependence

lemmas *WODNonInterferenceGraph* =
lift-wod-backward-slice[*OF While-CFGExit-wf-aux HighLowDistinct HighLowUNIV*]

lemma *WODNonInterference*:
NonInterferenceIntra src trg knd
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*))
NewEntry (*lift-Def* (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*lift-Use* (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L*) *id*
(*CFG-wf.wod-backward-slice src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*))
(*lift-Def* (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*lift-Use* (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L*))
reds (*labels-LDCFG-nodes prog*)
NewExit H L (*LDCFG-node.Node* (*-Entry-*)) (*LDCFG-node.Node* (*-Exit-*)) *final*
⟨*proof*⟩

4.3 Semantic Non-Interference for Standard Control Dependence

lemma *inner-node-exists*:$\exists$ *n. CFGExit.inner-node sourcenode targetnode*
(*valid-edge prog*) (*-Entry-*) (*-Exit-*) *n*
⟨*proof*⟩

lemmas *SCDNonInterferenceGraph* =
lift-PDG-standard-backward-slice[*OF WStandardControlDependence.PDG-scd*
WhilePostdomination-aux - HighLowDistinct HighLowUNIV]

lemma *SCDNonInterference*:
NonInterferenceIntra src trg knd
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*))
NewEntry (*lift-Def* (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*lift-Use* (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L*) *id*
(*PDG.PDG-BS src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*))

(*lift-Def* (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*lift-Use* (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*Postdomination.standard-control-dependence src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*)) *NewExit*))
reds (*labels-LDCFG-nodes prog*)
NewExit H L (*LDCFG-node.Node* (*-Entry-*)) (*LDCFG-node.Node* (*-Exit-*)) *final*
⟨*proof*⟩

4.4 Semantic Non-Interference for Weak Control Dependence

lemmas *WCDNonInterferenceGraph* =
lift-PDG-weak-backward-slice[*OF WWeakControlDependence.PDG-wcd*
WhileStrongPostdomination-aux - HighLowDistinct HighLowUNIV]

lemma *WCDNonInterference*:
NonInterferenceIntra src trg knd
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*))
NewEntry (*lift-Def* (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*lift-Use* (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L*) *id*
(*PDG.PDG-BS src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*))
(*lift-Def* (*Defs prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*lift-Use* (*Uses prog*) (*-Entry-*) (*-Exit-*) *H L*)
(*StrongPostdomination.weak-control-dependence src trg*
(*lift-valid-edge* (*valid-edge prog*) *sourcenode targetnode kind*
(*-Entry-*) (*-Exit-*)) *NewExit*))
reds (*labels-LDCFG-nodes prog*)
NewExit H L (*LDCFG-node.Node* (*-Entry-*)) (*LDCFG-node.Node* (*-Exit-*)) *final*
⟨*proof*⟩

end

end

References

[1] G. Barthe and L. P. Nieto. Secure information flow for a concurrent language with scheduling. *Journal of Computer Security*, 15(6):647–689, 2007.

[2] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference Java bytecode verifier. In *ESOP 2007*, volume 4421 of *LNCS*, pages 125–140. Springer, 2007.

[3] L. Beringer and M. Hofmann. Secure information flow and program logics. In *Archive of Formal Proofs*. http://isa-afp.org/entries/SIFPL.shtml, November 2008. Formal proof development.

[4] C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.

[5] F. Kammüller. Formalizing non-interference for a simple bytecode language in Coq. *Formal Aspects of Computing*, 20(3):259–275, 2008.

[6] A. Sabelfeld and D. Sands. A per model of secure information flow in sequential programs. *Higher Order Symbolic Computation*, 14(1):59–91, 2001.

[7] G. Snelting and D. Wasserrab. A correctness proof for the Volpano/Smith security typing system. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. http://isa-afp.org/entries/VolpanoSmith.shtml, September 2008. Formal proof development.

[8] D. Wasserrab. Towards certified slicing. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*. http://isa-afp.org/entries/Slicing.shtml, September 2008. Formal proof development.

[9] D. Wasserrab. Backing up slicing: Verifying the interprocedural two-phase Horwitz-Reps-Binkley slicer. In *Archive of Formal Proofs*. http://isa-afp.org/entries/HRB-Slicing.shtml, September 2009. Formal proof development.

[10] D. Wasserrab, D. Lohner, and G. Snelting. On PDG-based noninterference and its modular proof. In *Proc. of PLAS '09*, pages 31–44. ACM, June 2009.