Abstract

In this contribution, we show how correctness proofs for intra- and interprocedural slicing 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
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 fulfill 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-uf 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 = (λs. True)✓
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 = (λs. True)✓
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 (\text{-Low-}) = L
and HighLowDistinct: H \cap L = \{\}
and HighLowUNIV: H \cup L = UNIV

\begin{proof}

\textbf{lemma Low-neq-Exit:} \textbf{assumes} L \neq \{\} \textbf{shows} (\text{-Low-}) \neq (\text{-Exit-})
\end{proof}

\textbf{lemma Entry-path-High-path:}
\textbf{assumes} (\text{-Entry-}) \rightarrow^* n \textbf{and} inner-node n
\textbf{obtains} a' as' \textbf{where} as = a'\#as' \textbf{and} (\text{-High-}) \rightarrow^* n
\textbf{and} kind a' = (\lambda s. True)\sqrt{}
\end{proof}

\textbf{lemma Exit-path-Low-path:}
\textbf{assumes} n \rightarrow^* (\text{-Exit-}) \textbf{and} inner-node n
\textbf{obtains} a' as' \textbf{where} as = as\#[a'] \textbf{and} n \rightarrow^* (\text{-Low-})
\textbf{and} kind a' = (\lambda s. True)\sqrt{}
\end{proof}

\textbf{lemma not-Low-High:} V \not\in L \implies V \in H
\begin{proof}

\textbf{lemma not-High-Low:} V \not\in H \implies V \in L
\begin{proof}

\subsection{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. \textit{Low equivalence} groups those states in an equivalence class using the relation \(\approx_L\):

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

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

\textbf{lemma relevant-vars-Entry:}
\textbf{assumes} V \in rv S (\text{-Entry-}) \textbf{and} (\text{-High-}) \not\in \text{backward-slice } S
\textbf{shows} V \in L
\end{proof}
**lemma** lowEquivalence-relevant-nodes-Entry:
assumes $s ≈_L s'$ and (-High-) /∈ backward-slice S
shows $∀ V ∈ rv S (-Entry). \text{state-val } s V = \text{state-val } s' V$
(proof)

**lemma** rv-Low-Use-Low:
assumes (-Low-) ∈ S
shows $[n - as→∗ (-Low); n - as'→∗ (-Low); \forall V ∈ rv S n. \text{state-val } s V = \text{state-val } s' V; preds (slice-kinds S as) s; preds (slice-kinds S as') s'] \implies ∀ V ∈ Use (-Low). \text{state-val (transfers (slice-kinds S as) s) } V = \text{state-val (transfers (slice-kinds S as') s') } V$
(proof)

### 2.3 The Correctness Proofs

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

First, a theorem regarding (-Entry-) − as→∗ (-Exit-) paths in the control flow graph (CFG), which agree to a complete program execution:

**lemma** nonInterference-path-to-Low:
assumes $s ≈_L s'$ and (-High-) /∈ backward-slice S and (-Low-) ∈ S and (-Entry-) − as→∗ (-Low-) and preds (kinds as) s
and (-Entry-) − as'→∗ (-Low-) and preds (kinds as') s'
s shows transfers (kinds as) s ≈_L transfers (kinds as') s'
(proof)

**theorem** nonInterference-path:
assumes $s ≈_L s'$ and (-High-) /∈ backward-slice S and (-Low-) ∈ S and (-Entry-) − as→∗ (-Exit-) and preds (kinds as) s
and (-Entry-) − as'→∗ (-Exit-) and preds (kinds as') s'
s shows transfers (kinds as) s ≈_L transfers (kinds as') s'
(proof)

end

The second theorem assumes that we have a operational semantics, whose evaluations are written $⟨c,s⟩ \Rightarrow ⟨c',s'⟩$ 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 resulting states are again low equivalent:

locale NonInterferenceIntra =
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 (λs. True)√ 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 initial 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-) \( \notin \) backward-slice \( S \) and (-Low-) \( \in S \)
and valid-edge \( a \) and sourcenode \( a = (-High-) \) and targetnode \( a = n \)
and \( \text{kind } a = (\lambda s. \text{True})\sqrt \) and \( n \triangleq c \) 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 \( s_1' \approx_L s_2' \)

\( \langle \text{proof} \rangle \)
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 × ('state edge-kind) × 'node LDCFG-node

3.1.2 Lifting valid-edge

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

where lve-edge:
[valid-edge a; src a ≠ E ∨ trg a ≠ X;
 e = (Node (src a),knd a,Node (trg a))]⇒ lift-valid-edge valid-edge src trg knd E X e

| lve-Entry-edge:
e = (NewEntry,(λs. True),Node E)
⇒ lift-valid-edge valid-edge src trg knd E X e

| lve-Exit-edge:
e = (Node X,(λs. True),NewExit)
⇒ lift-valid-edge valid-edge src trg knd E X e

| lve-Entry-Exit-edge:
e = (NewEntry,(λs. False),NewExit)
⇒ lift-valid-edge valid-edge src trg knd E X e

lemma [simp];¬ lift-valid-edge valid-edge src trg knd E X (Node E,et,Node X)
⟨proof⟩

3.1.3 Lifting Def and Use sets

inductive-set lift-Def-set :: ('node ⇒ 'var set) ⇒ 'node ⇒ 'node ⇒
'var set ⇒ 'var set ⇒ ('node LDCFG-node × 'var) set

for Def::('node ⇒ 'var set) and E::'node and X::'node
and H::'var set and L::'var set

where lift-Def-node:
V ∈ Def n ⇒ (Node n, V) ∈ lift-Def-set Def E X H L

| lift-Def-High:
V ∈ H ⇒ (Node E, V) ∈ lift-Def-set Def E X H L

abbreviation lift-Def :: ('node ⇒ 'var set) ⇒ 'node ⇒ 'node ⇒
'var set ⇒ 'var set ⇒ ('node LDCFG-node ⇒ 'var set)

where lift-Def Def E X H L n ≡ \{ V. (n, V) ∈ lift-Def-set Def E X H L\}

inductive-set lift-Use-set :: ('node ⇒ 'var set) ⇒ 'node ⇒ 'node ⇒
'var set ⇒ 'var set ⇒ ('node LDCFG-node × 'var) set

for Use::'node ⇒ 'var set and E::'node and X::'node
and H::'var set and L::'var set

where

lift-Use-node:
V ∈ Use n ⇒ (Node n, V) ∈ lift-Use-set Use E X H L

| lift-Use-High:
V ∈ H ⇒ (Node E, V) ∈ lift-Use-set Use E X H L

| lift-Use-Low:
V ∈ L ⇒ (Node X, V) ∈ lift-Use-set Use E X H L

abbreviation lift-Use :: ('node ⇒ 'var set) ⇒ 'node ⇒ 'node ⇒
'var set ⇒ 'var set ⇒ ('node LDCFG-node ⇒ 'var set)

where lift-Use Use E X H L n ≡ \{ V. (n, V) ∈ lift-Use-set Use E X H L\}

3.2 The lifting lemmas

3.2.1 Lifting the basic locales

abbreviation src :: ('edge,'node,'state) LDCFG-edge ⇒ 'node LDCFG-node

where src a ≡ fst a

abbreviation trg :: ('edge,'node,'state) LDCFG-edge ⇒ 'node LDCFG-node

where trg a ≡ snd(snd a)

definition knd :: ('edge,'node,'state) LDCFG-edge ⇒ 'state edge-kind

where knd a ≡ fst(snd a)

lemma lift-CFG:
assumes $af:CFGExit-wf$ 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 $af:CFGExit-wf$ sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
shows $CFG-wf$ src trg lnd
(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 $af:CFGExit-wf$ sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
shows $CFGExit$ src trg lnd
(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry NewExit (proof)

lemma lift-CFGExit-wf:
assumes $af:CFGExit-wf$ sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
shows $CFGExit-wf$ src trg lnd
(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)

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 $lve:lv$ $\equiv$ lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit and $lDef:lDef \equiv$ lift-Def Def Entry Exit H L and $lUse:lUse \equiv$ lift-Use Use Entry Exit H L
assumes $af:CFGExit-wf$ sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
and $H \cap L = \{}$ and $H \cup L = \text{UNIV}$
serves $NonInterferenceIntraGraph src trg lnd 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

- \( \text{wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use} \)
- \( \text{state-val Exit} \)
- \( \text{pd:Postdomination sourcenode targetnode kind valid-edge Entry Exit} \)
- \( \text{inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx} \)

Shows

Postdomination \( \text{src trg kn}d \)

\( \text{(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry NewExit} \)

⟨proof⟩

**Lemma lift-PDG-scd:**

Assumes

- \( \text{PDG:PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit} \)
- \( \text{(Postdomination.standard-control-dependence sourcenode targetnode valid-edge Exit)} \)
- \( \text{pd:Postdomination sourcenode targetnode kind valid-edge Entry Exit} \)
- \( \text{inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx} \)

Shows

PDG \( \text{src trg kn}d \)

\( \text{(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry NewExit} \)

\( \text{(lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val NewExit} \)

\( \text{(Postdomination.standard-control-dependence src trg} \)

\( \text{(lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewExit} \)

⟨proof⟩

**Lemma lift-PDG-standard-backward-slice:**

Fixes

valid-edge \( \text{and sourcenode and targetnode and kind and Entry and Exit} \)

Def \( \text{and Use and H and L} \)

defines

\( \text{lve:lle ≡ lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit} \)

\( \text{lDef:lDef ≡ lift-Def Def Entry Exit H L} \)

\( \text{lUse:lUse ≡ lift-Use Use Entry Exit H L} \)

Assumes

- \( \text{PDG:PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit} \)
- \( \text{(Postdomination.standard-control-dependence sourcenode targetnode valid-edge Exit)} \)
- \( \text{inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx} \)
- \( \text{H ∩ L = \{} \text{and} H ∪ L = \text{UNIV} \)

Shows

NonInterferenceIntraGraph src trg knd lve NewEntry lDef lUse state-val

\( \text{(PDG.PDG-BS src trg lve lDef lUse} \)

\( \text{(Postdomination.standard-control-dependence src trg lve NewExit))} \)

\( \text{NewExit H L (Node Entry) (Node Exit)} \)

⟨proof⟩

3.2.4 Lifting PDG-BS with weak-control-dependence

**Lemma lift-StrongPostdomination:**

Assumes

- \( \text{wf:CFGExit-wf sourcenode targetnode kind valid-edge Entry Def Use} \)
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 kn (lif-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 kn (lif-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 (lif-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 ≡ lif-valid-edge valid-edge sourcenode targetnode kind Entry Exit and lDef:lDef ≡ lif-Def Def Entry Exit H L and lUse:lUse ≡ lif-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 ∩ L = {} and H ∪ L = UNIV
shows NonInterferenceIntraGraph src trg kn 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 ∩ L = {}
  and HighLowUNIV: H ∪ 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

11
lemma final-edge:
labels-nodes prog n Skip \implies prog \vdash n \uparrow id \rightarrow \text{-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
   \text{-Entry-} \text{-Exit-})
  NewEntry (lift-Def (Defs prog) \text{-Entry-} \text{-Exit-} H L)
  (lift-Use (Uses prog) \text{-Entry-} \text{-Exit-} H L) id
  (CFG-wf, wod-backward-slice src trg
   (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
    \text{-Entry-} \text{-Exit-}))
  (lift-Def (Defs prog) \text{-Entry-} \text{-Exit-} H L)
  (lift-Use (Uses prog) \text{-Entry-} \text{-Exit-} H L)
  reds (labels-LDCFG-nodes prog
   NewExit H L (LDCFG-node.\text{Node} \text{-Entry-}) (LDCFG-node.\text{Node} \text{-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) \text{-Entry-} \text{-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
   \text{-Entry-} \text{-Exit-})
  NewEntry (lift-Def (Defs prog) \text{-Entry-} \text{-Exit-} H L)
  (lift-Use (Uses prog) \text{-Entry-} \text{-Exit-} H L) id
  (PDG.PDG-BS src trg
   (lift-valid-edge (valid-edge prog) sourcenode targetnode kind
    \text{-Entry-} \text{-Exit-}))
4.4 Semantic Non-Interference for Weak Control Dependence

**lemmas**  \( WCDNonInterferenceGraph = \)
\[
\text{lift-Def (Defs prog)} \ (\text{-Entry-}) \ (\text{-Exit-}) \ H \ L
\]
\[
\text{lift-Use (Uses prog)} \ (\text{-Entry-}) \ (\text{-Exit-}) \ H \ L
\]
\[
\text{(Postdomination \_standard-control-dependence \ src \ trg}
\]
\[
\text{(lift-valid-edge \ (valid-edge prog) \ sourcenode \ targetnode \ kind}
\]
\[
\text{(\text{-Entry-}) \ (\text{-Exit-}) \ NewExit})
\]
\[
\text{reds \ (labels-LDCFG-nodes \ prog) \ NewExit \ H \ L \ (LDCFG-node.Node \ (\text{-Entry-})) \ (LDCFG-node.Node \ (\text{-Exit-})) \ final}
\]
\[
\langle \text{proof} \rangle
\]

**lemma**  \( WCDNonInterference: \)
\[
\text{NonInterferenceIntra \ src \ trg \ kind}
\]
\[
\text{(lift-valid-edge \ (valid-edge prog) \ sourcenode \ targetnode \ kind}
\]
\[
\text{(\text{-Entry-}) \ (\text{-Exit-}))}
\]
\[
\text{NewEntry \ (lift-Def \ (Defs prog)} \ (\text{-Entry-}) \ (\text{-Exit-}) \ H \ L
\]
\[
\text{(lift-Use \ (Uses prog)} \ (\text{-Entry-}) \ (\text{-Exit-}) \ H \ L \ id}
\]
\[
\text{(PDG.PDG-BS \ src \ trg}
\]
\[
\text{(lift-valid-edge \ (valid-edge prog) \ sourcenode \ targetnode \ kind}
\]
\[
\text{(\text{-Entry-}) \ (\text{-Exit-}))}
\]
\[
\text{(lift-Def \ (Defs prog)} \ (\text{-Entry-}) \ (\text{-Exit-}) \ H \ L
\]
\[
\text{(lift-Use \ (Uses prog)} \ (\text{-Entry-}) \ (\text{-Exit-}) \ H \ L
\]
\[
\text{(StrongPostdomination \_weak-control-dependence \ src \ trg}
\]
\[
\text{(lift-valid-edge \ (valid-edge prog) \ sourcenode \ targetnode \ kind}
\]
\[
\text{(\text{-Entry-}) \ (\text{-Exit-}) \ NewExit})
\]
\[
\text{reds \ (labels-LDCFG-nodes \ prog) \ NewExit \ H \ L \ (LDCFG-node.Node \ (\text{-Entry-})) \ (LDCFG-node.Node \ (\text{-Exit-})) \ final}
\]
\[
\langle \text{proof} \rangle
\]

end

end

References


