Slicing Guarantees Information Flow Noninterference

Daniel Wasserrab

March 12, 2013

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 HRB Slicing guarantees IFC Noninterference

theory *NonInterferenceInter*
  imports *../HRB−Slicing/StaticInter/FundamentalProperty*
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 [9] accordingly in a new locale:

locale *NonInterferenceInterGraph* =
  SDG sourcenode targetnode kind valid-edge Entry
    get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
  for *sourcenode* :: *'edge* ⇒ *'node* **and** *targetnode* :: *'edge* ⇒ *'node*
  and *kind* :: *'edge* ⇒ (*'var*,*'val*,*'ret*,*'pname*) *edge-kind*
  and *valid-edge* :: *'edge* ⇒ *bool*
  and *Entry* :: *'node* ('('-*Entry*'-')) **and** *get-proc* :: *'node* ⇒ *'pname*
  and *get-return-edges* :: *'edge* ⇒ *'edge set*
  and *procs* :: (*'pname* × *'var list* × *'var list*) *list* **and** *Main* :: *'pname*
  and *Exit*::*'node* ('('-*Exit*'-'))
  and *Def* :: *'node* ⇒ *'var set* **and** *Use* :: *'node* ⇒ *'var set*
  and *ParamDefs* :: *'node* ⇒ *'var list* **and** *ParamUses* :: *'node* ⇒ *'var set list* +
  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 *valid-node-High* [*simp*]:*valid-node* (*-High-*)
⟨*proof*⟩

lemma *valid-node-Low* [*simp*]:*valid-node* (*-Low-*)
⟨*proof*⟩

lemma *get-proc-Low*:
get-proc (*-Low-*) = *Main*
⟨*proof*⟩

lemma *get-proc-High*:
get-proc (*-High-*) = *Main*
⟨*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* :: $('var \rightharpoonup 'val)$ *list* $\Rightarrow$ $('var \rightharpoonup 'val)$ *list* $\Rightarrow$ *bool*
(**infixl** $\approx_L$ *50*)
  where $s \approx_L s' \equiv \forall\, V \in L.\ hd\ s\ V = hd\ s'\ V$

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

lemma *relevant-vars-Entry*:
  assumes $V \in rv\ S\ (CFG\text{-}node\ (\text{-}Entry\text{-}))$ **and** $(\text{-}High\text{-}) \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
  shows $V \in L$
⟨*proof*⟩

lemma *lowEquivalence-relevant-nodes-Entry*:
  assumes $s \approx_L s'$ **and** $(\text{-}High\text{-}) \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
  shows $\forall\, V \in rv\ S\ (CFG\text{-}node\ (\text{-}Entry\text{-})).\ hd\ s\ V = hd\ s'\ V$
⟨*proof*⟩

2.3 The Correctness Proofs

In the following, we present two correctness proofs that slicing guarantees IFC noninterference. In both theorems, *CFG-node* (*-High-*) $\notin$ *HRB-slice S*, where *CFG-node* (*-Low-*) $\in$ *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-*) $-as\rightarrow*$ (*-Exit-*) paths in the control flow graph (CFG), which agree to a complete program execution:

lemma *slpa-rv-Low-Use-Low*:
  assumes $CFG\text{-}node\ (\text{-}Low\text{-}) \in S$
  shows $[\![same\text{-}level\text{-}path\text{-}aux\ cs\ as;\ upd\text{-}cs\ cs\ as = [];\ same\text{-}level\text{-}path\text{-}aux\ cs\ as';$
   $\forall\, c \in set\ cs.\ valid\text{-}edge\ c;\ m\ -as\rightarrow*\ (\text{-}Low\text{-});\ m\ -as'\rightarrow*\ (\text{-}Low\text{-});$
  $\forall\, i < length\ cs.\ \forall\, V \in rv\ S\ (CFG\text{-}node\ (sourcenode\ (cs!i))).$
  $fst\ (s!Suc\ i)\ V = fst\ (s'!Suc\ i)\ V;\ \forall\, i < Suc\ (length\ cs).\ snd\ (s!i) = snd\ (s'!i);$
  $\forall\, V \in rv\ S\ (CFG\text{-}node\ m).\ state\text{-}val\ s\ V = state\text{-}val\ s'\ V;$
  $preds\ (slice\text{-}kinds\ S\ as)\ s;\ preds\ (slice\text{-}kinds\ S\ as')\ s';$
  $length\ s = Suc\ (length\ cs);\ length\ s' = Suc\ (length\ cs)]\!]$
  $\Longrightarrow \forall\, V \in Use\ (\text{-}Low\text{-}).\ state\text{-}val\ (transfers(slice\text{-}kinds\ S\ as)\ s)\ V =$
          $state\text{-}val\ (transfers(slice\text{-}kinds\ S\ as')\ s')\ V$
⟨*proof*⟩

lemma *rv-Low-Use-Low*:
  assumes $m\ -as\rightarrow_{\surd}*\ (\text{-}Low\text{-})$ **and** $m\ -as'\rightarrow_{\surd}*\ (\text{-}Low\text{-})$ **and** $get\text{-}proc\ m = Main$
  and $\forall\ V \in rv\ S\ (CFG\text{-}node\ m).\ cf\ V = cf'\ V$
  and $preds\ (slice\text{-}kinds\ S\ as)\ [(cf,undefined)]$
  and $preds\ (slice\text{-}kinds\ S\ as')\ [(cf',undefined)]$
  and $CFG\text{-}node\ (\text{-}Low\text{-}) \in S$
  shows $\forall\ V \in Use\ (\text{-}Low\text{-}).$
    $state\text{-}val\ (transfers(slice\text{-}kinds\ S\ as)\ [(cf,undefined)])\ V =$
    $state\text{-}val\ (transfers(slice\text{-}kinds\ S\ as')\ [(cf',undefined)])\ V$
⟨*proof*⟩

lemma *nonInterference-path-to-Low*:
  assumes $[cf] \approx_L [cf']$ **and** $(\text{-}High\text{-}) \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
  and $CFG\text{-}node\ (\text{-}Low\text{-}) \in S$
  and $(\text{-}Entry\text{-})\ -as\rightarrow_{\surd}*\ (\text{-}Low\text{-})$ **and** $preds\ (kinds\ as)\ [(cf,undefined)]$
  and $(\text{-}Entry\text{-})\ -as'\rightarrow_{\surd}*\ (\text{-}Low\text{-})$ **and** $preds\ (kinds\ as')\ [(cf',undefined)]$
  shows $map\ fst\ (transfers\ (kinds\ as)\ [(cf,undefined)]) \approx_L$
        $map\ fst\ (transfers\ (kinds\ as')\ [(cf',undefined)])$
⟨*proof*⟩

theorem *nonInterference-path*:
  assumes $[cf] \approx_L [cf']$ **and** $(\text{-}High\text{-}) \notin \lfloor HRB\text{-}slice\ S \rfloor_{CFG}$
  and $CFG\text{-}node\ (\text{-}Low\text{-}) \in S$
  and $(\text{-}Entry\text{-})\ -as\rightarrow_{\surd}*\ (\text{-}Exit\text{-})$ **and** $preds\ (kinds\ as)\ [(cf,undefined)]$
  and $(\text{-}Entry\text{-})\ -as'\rightarrow_{\surd}*\ (\text{-}Exit\text{-})$ **and** $preds\ (kinds\ as')\ [(cf',undefined)]$
  shows $map\ fst\ (transfers\ (kinds\ as)\ [(cf,undefined)]) \approx_L$
  $map\ fst\ (transfers\ (kinds\ as')\ [(cf',undefined)])$
⟨*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 *NonInterferenceInter* =
  NonInterferenceInterGraph sourcenode targetnode kind valid-edge Entry
    get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
    H L High Low +
  SemanticsProperty sourcenode targetnode kind valid-edge Entry get-proc
    get-return-edges procs Main Exit Def Use ParamDefs ParamUses sem identifies
  for $sourcenode :: 'edge \Rightarrow 'node$ **and** $targetnode :: 'edge \Rightarrow 'node$
  and $kind :: 'edge \Rightarrow ('var,'val,'ret,'pname)\ edge\text{-}kind$
  and $valid\text{-}edge :: 'edge \Rightarrow bool$

and *Entry* :: *'node* ('('-*Entry*'-')) **and** *get-proc* :: *'node* ⇒ *'pname*
and *get-return-edges* :: *'edge* ⇒ *'edge set*
and *procs* :: (*'pname* × *'var list* × *'var list*) *list* **and** *Main* :: *'pname*
and *Exit*::*'node* ('('-*Exit*'-'))
and *Def* :: *'node* ⇒ *'var set* **and** *Use* :: *'node* ⇒ *'var set*
and *ParamDefs* :: *'node* ⇒ *'var list* **and** *ParamUses* :: *'node* ⇒ *'var set list*
and *sem* :: *'com* ⇒ (*'var* ⇀ *'val*) *list* ⇒ *'com* ⇒ (*'var* ⇀ *'val*) *list* ⇒ *bool*
(((*1*⟨-,/-⟩) ⇒/ (*1*⟨-,/-⟩)) [*0*,*0*,*0*,*0*] *81*)
and *identifies* :: *'node* ⇒ *'com* ⇒ *bool* (- $\triangleq$ - [*51*,*0*] *80*)
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* $\triangleq$ *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 $[cf_1] \approx_L [cf_2]$ **and** (-*High*-) ∉ ⌊*HRB-slice S*⌋$_{CFG}$
and *CFG-node* (-*Low*-) ∈ *S*
and *valid-edge a* **and** *sourcenode a* = (-*High*-) **and** *targetnode a* = *n*
and *kind a* = $(\lambda s.\ True)_{\surd}$ **and** *n* $\triangleq$ *c* **and** *final c'*
and $\langle c,[cf_1]\rangle \Rightarrow \langle c',s_1\rangle$ **and** $\langle c,[cf_2]\rangle \Rightarrow \langle c',s_2\rangle$
shows $s_1 \approx_L s_2$
⟨*proof*⟩

end

end

3 Framework Graph Lifting for Noninterference

theory *LiftingInter*
imports *NonInterferenceInter*
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,'var,'val,'ret,'pname) LDCFG-edge =*
  'node LDCFG-node $\times$ *(('var,'val,'ret,'pname) edge-kind)* $\times$ *'node LDCFG-node*

3.1.2 Lifting basic definitions using *'edge* and *'node*

inductive *lift-valid-edge* :: *('edge* $\Rightarrow$ *bool)* $\Rightarrow$ *('edge* $\Rightarrow$ *'node)* $\Rightarrow$ *('edge* $\Rightarrow$ *'node)* $\Rightarrow$
  ('edge $\Rightarrow$ *('var,'val,'ret,'pname) edge-kind)* $\Rightarrow$ *'node* $\Rightarrow$ *'node* $\Rightarrow$
  ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge $\Rightarrow$
  bool
for *valid-edge::'edge* $\Rightarrow$ *bool* **and** *src::'edge* $\Rightarrow$ *'node* **and** *trg::'edge* $\Rightarrow$ *'node*
  and *knd::'edge* $\Rightarrow$ *('var,'val,'ret,'pname) 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,*$(\lambda s.\ True)_{\surd}$*,Node E)*
  $\Longrightarrow$ *lift-valid-edge valid-edge src trg knd E X e*

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

  | *lve-Entry-Exit-edge*:
  e = *(NewEntry,*$(\lambda s.\ False)_{\surd}$*,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*$\rangle$

fun *lift-get-proc* :: *('node* $\Rightarrow$ *'pname)* $\Rightarrow$ *'pname* $\Rightarrow$ *'node LDCFG-node* $\Rightarrow$ *'pname*

where *lift-get-proc get-proc Main* (*Node n*) = *get-proc n*
| *lift-get-proc get-proc Main NewEntry* = *Main*
| *lift-get-proc get-proc Main NewExit* = *Main*

inductive-set *lift-get-return-edges* :: ('*edge* ⇒ '*edge set*) ⇒ ('*edge* ⇒ *bool*) ⇒
('*edge* ⇒ '*node*) ⇒ ('*edge* ⇒ '*node*) ⇒ ('*edge* ⇒ ('*var*,'*val*,'*ret*,'*pname*) *edge-kind*)

⇒ ('*edge*,'*node*,'*var*,'*val*,'*ret*,'*pname*) *LDCFG-edge*
⇒ ('*edge*,'*node*,'*var*,'*val*,'*ret*,'*pname*) *LDCFG-edge set*
for *get-return-edges* :: '*edge* ⇒ '*edge set* **and** *valid-edge* :: '*edge* ⇒ *bool*
and *src*::'*edge* ⇒ '*node* **and** *trg*::'*edge* ⇒ '*node*
and *knd*::'*edge* ⇒ ('*var*,'*val*,'*ret*,'*pname*) *edge-kind*
and *e*::('*edge*,'*node*,'*var*,'*val*,'*ret*,'*pname*) *LDCFG-edge*
where *lift-get-return-edgesI*:
⟦*e* = (*Node* (*src a*),*knd a*,*Node* (*trg a*)); *valid-edge a*; *a'* ∈ *get-return-edges a*;
e' = (*Node* (*src a'*),*knd a'*,*Node* (*trg a'*))⟧
⟹ *e'* ∈ *lift-get-return-edges get-return-edges valid-edge src trg knd e*

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

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

| *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-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*}

fun *lift-ParamUses* :: ('*node* ⇒ '*var set list*) ⇒ '*node LDCFG-node* ⇒ '*var set list*
where *lift-ParamUses ParamUses* (*Node n*) = *ParamUses n*
| *lift-ParamUses ParamUses NewEntry* = []
| *lift-ParamUses ParamUses NewExit* = []

fun *lift-ParamDefs* :: ('*node* ⇒ '*var list*) ⇒ '*node LDCFG-node* ⇒ '*var list*
where *lift-ParamDefs ParamDefs* (*Node n*) = *ParamDefs n*
| *lift-ParamDefs ParamDefs NewEntry* = []
| *lift-ParamDefs ParamDefs NewExit* = []

3.2 The lifting lemmas

3.2.1 Lifting the CFG locales

abbreviation *src* :: ('*edge*,'*node*,'*var*,'*val*,'*ret*,'*pname*) *LDCFG-edge* ⇒ '*node LDCFG-node*
where *src a* ≡ *fst a*

abbreviation *trg* :: ('*edge*,'*node*,'*var*,'*val*,'*ret*,'*pname*) *LDCFG-edge* ⇒ '*node LDCFG-node*
where *trg a* ≡ *snd*(*snd a*)

abbreviation *knd* :: ('*edge*,'*node*,'*var*,'*val*,'*ret*,'*pname*) *LDCFG-edge* ⇒
('*var*,'*val*,'*ret*,'*pname*) *edge-kind*
where *knd a* ≡ *fst*(*snd a*)

lemma *lift-CFG*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc*
get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and *pd*:*Postdomination sourcenode targetnode kind valid-edge Entry get-proc*
get-return-edges procs Main Exit
shows *CFG src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
(*lift-get-proc get-proc Main*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind*)
procs Main
⟨*proof*⟩

lemma *lift-CFG-wf*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc*

get-return-edges procs Main Exit Def Use ParamDefs ParamUses
and *pd*:*Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit*
shows *CFG-wf src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
(*lift-get-proc get-proc Main*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind*)
procs Main (*lift-Def Def Entry Exit H L*) (*lift-Use Use Entry Exit H L*)
(*lift-ParamDefs ParamDefs*) (*lift-ParamUses ParamUses*)
⟨*proof*⟩

lemma *lift-CFGExit*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses*
and *pd*:*Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit*
shows *CFGExit src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
(*lift-get-proc get-proc Main*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind*)
procs Main NewExit
⟨*proof*⟩

lemma *lift-CFGExit-wf*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses*
and *pd*:*Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit*
shows *CFGExit-wf src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
(*lift-get-proc get-proc Main*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind*)
procs Main NewExit (*lift-Def Def Entry Exit H L*) (*lift-Use Use Entry Exit H L*)
(*lift-ParamDefs ParamDefs*) (*lift-ParamUses ParamUses*)
⟨*proof*⟩

3.2.2 Lifting the SDG

lemma *lift-Postdomination*:
assumes *wf*:*CFGExit-wf sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses*
and *pd*:*Postdomination sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main 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*
(*lift-get-proc get-proc Main*)

(*lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind*)
procs Main NewExit
⟨*proof*⟩

lemma *lift-SDG*:
assumes *SDG*:*SDG sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses*
and *inner*:*CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx*
shows *SDG src trg knd*
(*lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*) *NewEntry*
(*lift-get-proc get-proc Main*)
(*lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind*)
procs Main NewExit (*lift-Def Def Entry Exit H L*) (*lift-Use Use Entry Exit H L*)
(*lift-ParamDefs ParamDefs*) (*lift-ParamUses ParamUses*)
⟨*proof*⟩

3.2.3 Low-deterministic security via the lifted graph

lemma *Lift-NonInterferenceGraph*:
fixes *valid-edge* **and** *sourcenode* **and** *targetnode* **and** *kind* **and** *Entry* **and** *Exit*
and *get-proc* **and** *get-return-edges* **and** *procs* **and** *Main*
and *Def* **and** *Use* **and** *ParamDefs* **and** *ParamUses* **and** *H* **and** *L*
defines *lve*:*lve* ≡ *lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit*
and *lget-proc*:*lget-proc* ≡ *lift-get-proc get-proc Main*
and *lget-return-edges*:*lget-return-edges* ≡
lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
and *lDef*:*lDef* ≡ *lift-Def Def Entry Exit H L*
and *lUse*:*lUse* ≡ *lift-Use Use Entry Exit H L*
and *lParamDefs*:*lParamDefs* ≡ *lift-ParamDefs ParamDefs*
and *lParamUses*:*lParamUses* ≡ *lift-ParamUses ParamUses*
assumes *SDG*:*SDG sourcenode targetnode kind valid-edge Entry get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses*
and *inner*:*CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx*
and $H \cap L = \{\}$ **and** $H \cup L = UNIV$
shows *NonInterferenceInterGraph src trg knd lve NewEntry lget-proc lget-return-edges procs Main NewExit lDef lUse lParamDefs lParamUses H L*
(*Node Entry*) (*Node Exit*)
⟨*proof*⟩

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://afp.sf.net/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://afp.sf.net/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://afp.sf.net/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://afp.sf.net/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.