Slicing Guarantees Information Flow Noninterference

Daniel Wasserrab
August 16, 2018

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 fulfill 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 determine 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
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 [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 set 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 = (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 \( \text{DefHigh} \): \( \text{Def\ (-High-)} = H \)
and \( \text{UseHigh} \): \( \text{Use\ (-High-)} = H \)
and \( \text{UseLow} \): \( \text{Use\ (-Low-)} = L \)
and \( \text{HighLowDistinct} \): \( H \cap L = \{\} \)
and \( \text{HighLowUNIV} \): \( H \cup L = \text{UNIV} \)

begin

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

lemma \( \text{valid-node-High} \) [simp]: \( \text{valid-node\ (-High-)} \)
⟨proof⟩

lemma \( \text{valid-node-Low} \) [simp]: \( \text{valid-node\ (-Low-)} \)
⟨proof⟩

lemma \( \text{get-proc-Low} \):
get-proc \( \text{(-Low-)} = \text{Main} \)
⟨proof⟩

lemma \( \text{get-proc-High} \):
get-proc \( \text{(-High-)} = \text{Main} \)
⟨proof⟩

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

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

lemma \( \text{not-Low-High} \): \( V \notin L \Rightarrow V \in H \)
⟨proof⟩

lemma \( \text{not-High-Low} \): \( V \notin H \Rightarrow V \in L \)
⟨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 \(\approx\):

\[
\text{definition lowEquivalence :: ('var -> 'val) list \to ('var -> 'val) list \to bool}
\]

\[
\text{(infixl \approx L 50)}
\]

\[s \approx_L s' \equiv \forall V \in L. \text{hd} s V = \text{hd} s' V\]

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

**lemma relevant-vars-Entry:**

assumes \(V \in \text{rv} S (\text{CFG-node (-Entry-)})\) and \((-\text{High-}) \notin \lfloor \text{HRB-slice S} \rfloor_{\text{CFG}}\)

shows \(V \in L\)

(proof)

**lemma lowEquivalence-relevant-nodes-Entry:**

assumes \(s \approx_L s'\) and \((-\text{High-}) \notin \lfloor \text{HRB-slice S} \rfloor_{\text{CFG}}\)

shows \(\forall V \in \text{rv} S (\text{CFG-node (-Entry-)}). \text{hd} s V = \text{hd} s' V\)

(proof)

2.3 The Correctness Proofs

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

**lemma slpa-rv-Low-Use-Low:**

assumes \(\text{CFG-node (-Low-)} \in S\)

shows \(\lfloor \text{same-level-path-aux cs as} = []; \text{same-level-path-aux cs as'}\rfloor_{\text{CFG}}\)

\[\forall i < \text{length} cs. \forall V \in \text{rv} S (\text{CFG-node (sourcenode (cs!i)))}. \text{fst} (s!\text{Suc} i) V = \text{fst} (s'!\text{Suc} i) V; \forall i < \text{Suc} (\text{length} cs). \text{snd} (s!i) = \text{snd} (s'?i); \forall V \in \text{rv} S (\text{CFG-node m}). \text{state-val} s V = \text{state-val} s' V; \text{preds (slice-kinds S as)} s; \text{preds (slice-kinds S as')} s'; \text{length} s = \text{Suc} (\text{length} cs); \text{length} s' = \text{Suc} (\text{length} cs)\]

\[\text{transfers (slice-kinds S as) s} V = \text{transfers (slice-kinds S as') s'} V\]

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

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

theorem nonInterference-path:
assumes \( [cf] \approx_L [cf'] \) and \( (\text{High}) \notin [\text{HRB-slice} \ S] \text{CFG} \)
and \( \text{CFG-node} \ (\text{Low}) \in S \)
and \( (\text{-Entry}) \rightarrow_{\rho}^* (\text{Exit}) \) and \( \text{preds} \ (\text{slice-kinds} \ S \ as \) \( [(cf,\text{undefined})] \)
and \( (\text{-Entry}) \rightarrow_{\rho'}^* (\text{Exit}) \) and \( \text{preds} \ (\text{slice-kinds} \ S \ as' \) \( [(cf',\text{undefined})] \)
shows \( \text{map fst} \ (\text{transfers} \ (\text{slice-kinds} \ as \) \( [(cf,\text{undefined})] \)) \approx_L \text{map fst} \ (\text{transfers} \ (\text{slice-kinds} \ as' \) \( [(cf',\text{undefined})] \)) \)
(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 source-node target-node kind valid-edge Entry
get-proc get-return-edges procs Main Exit Def Use ParamDefs ParamUses
H L High Low +
SemanticsProperty source-node target-node kind valid-edge Entry get-proc
get-return-edges procs Main Exit Def Use ParamDefs ParamUses sem identifies
for source-node :: \( '\text{edge} \Rightarrow '\text{node} \) and target-node :: \( '\text{edge} \Rightarrow '\text{node} \)
and kind :: \( '\text{edge} \Rightarrow (\text{var, val, ret, pname}) \) edge-kind
and valid-edge :: \( '\text{edge} \Rightarrow \text{bool} \)
The following theorem needs the explicit edge from (-High-) to \( n \). An approach using a \textit{init} predicate for initial statements, being reachable from (-High-) via a \((\lambda s. \text{True})\sqrt{\text{edge}}\), does not work as the same statement could be identified by several nodes, some initial, some not. E.g., in the program \( \text{while (True) Skip;;Skip} \) two nodes identify this initial statement: the initial node and the node within the loop (because of loop unrolling).

\[ \text{theorem nonInterference:} \]
\[ \text{assumes } [c_1] \approx_L [c_2] \text{ and } (-\text{High-}) \notin [\text{HRB-slice } S] \text{CFG} \]
\[ \text{and CFG-node } (-\text{Low-}) \in S \]
\[ \text{and valid-edge } a \text{ and sourcenode } a = (-\text{High-}) \text{ and targetnode } a = n \]
\[ \text{and kind } a = (\lambda s. \text{True})\sqrt{\text{edge}} \text{ and } n \triangleq c \text{ and final } c' \]
\[ \text{and } \langle c, [c_1] \rangle \Rightarrow \langle c', s_1 \rangle \text{ and } \langle c, [c_2] \rangle \Rightarrow \langle c', s_2 \rangle \]
\[ \text{shows } s_1 \approx_L s_2 \]

(\text{proof})
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 × (('var,'val,'ret,'pname) edge-kind) × 'node LDCFG-node

3.1.2 Lifting basic definitions using 'edge and 'node

inductive lift-valid-edge :: ('edge ⇒ bool) ⇒ ('edge ⇒ 'node) ⇒ ('edge ⇒ 'node) ⇒ ('edge ⇒ ('var,'val,'ret,'pname) edge-kind) ⇒ ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge ⇒ bool

for valid-edge::'edge ⇒ bool and src::'edge ⇒ 'node and trg::'edge ⇒ 'node and knd::'edge ⇒ ('var,'val,'ret,'pname) edge-kind and E::'node and X::'node

where lve-edge:
[ valid-edge a; src a \neq E \lor trg a \neq 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,(\s. True)\_\_\_\_,Node E) 
⇒ lift-valid-edge valid-edge src trg knd E X e

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

| lve-Entry-Exit-edge:
  e = (NewEntry,(\s. 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)

fun lift-get-proc :: ('node ⇒ 'pname) ⇒ 'pname ⇒ 'node LDCFG-node ⇒ '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
\begin{align*}
\text{inductive-set} & \; \text{lift-get-return-edges} :: (\text{edge} \Rightarrow \text{edge set}) \Rightarrow (\text{edge} \Rightarrow \text{bool}) \Rightarrow \\
& (\text{edge} \Rightarrow \text{node}) \Rightarrow (\text{edge} \Rightarrow \text{node}) \Rightarrow (\text{edge} \Rightarrow (\text{var, val, ret, pname}) \text{ edge-kind}) \\
& \Rightarrow (\text{edge}, \text{node}, \text{var}, \text{val}, \text{ret}, \text{pname}) \text{ LDCFG-edge} \\
& \Rightarrow (\text{edge}, \text{node}, \text{var}, \text{val}, \text{ret}, \text{pname}) \text{ LDCFG-edge set} \\
\text{for} & \; \text{get-return-edges} :: \text{edge} \Rightarrow \text{edge set and valid-edge} :: \text{edge} \Rightarrow \text{bool} \\
\text{and} & \; \text{src} :: \text{edge} \Rightarrow \text{node} \text{ and trg} :: \text{edge} \Rightarrow \text{node} \\
\text{and} & \; \text{knd} :: \text{edge} \Rightarrow (\text{var}, \text{val}, \text{ret}, \text{pname}) \text{ edge-kind} \\
\text{and} & \; \text{e} :: (\text{edge}, \text{node}, \text{var}, \text{val}, \text{ret}, \text{pname}) \text{ LDCFG-edge} \\
\text{where} & \; \text{lift-get-return-edgesI}: \\
& [ \left[ e = (\text{Node} (\text{src} a), \text{knd} a, \text{Node} (\text{trg} a)); \text{valid-edge} a; a' \in \text{get-return-edges} a; \right. \\
& \left. e' = (\text{Node} (\text{src} a'), \text{knd} a', \text{Node} (\text{trg} a')) \right] ] \\
& \Rightarrow e' \in \text{lift-get-return-edges} \text{ get-return-edges valid-edge src trg knd e} \\
\end{align*}

3.1.3 Lifting the Def and Use sets

\begin{align*}
\text{inductive-set} & \; \text{lift-Def-set} :: (\text{node} \Rightarrow \text{var set}) \Rightarrow (\text{node} \Rightarrow \text{node} \Rightarrow \text{var set}) \Rightarrow (\text{node \ LDCFG-node \ \times \ \text{var set}) set} \\
\text{for} & \; \text{Def} :: (\text{node} \Rightarrow \text{var set}) \text{ and } E :: \text{node} \text{ and } X :: \text{node} \\
\text{and} & \; H :: \text{var set and } L :: \text{var set} \\
\text{where} & \; \text{lift-Def-node}: \\
& V \in \text{Def } n \Rightarrow (\text{Node } n, V) \in \text{lift-Def-set } \text{Def } E X H L \\
& | \text{lift-Def-High}: \\
& V \in H \Rightarrow (\text{Node } E, V) \in \text{lift-Def-set } \text{Def } E X H L \\
\text{abbreviation} & \; \text{lift-Def} :: (\text{node} \Rightarrow \text{var set}) \Rightarrow (\text{node} \Rightarrow \text{node} \Rightarrow \text{var set}) \Rightarrow (\text{node \ LDCFG-node \ \Rightarrow \ \text{var set}) set} \\
\text{where} & \; \text{lift-Def } \text{Def } E X H L n \equiv \{ V, (n, V) \in \text{lift-Def-set } \text{Def } E X H L \} \\
\end{align*}

\begin{align*}
\text{inductive-set} & \; \text{lift-Use-set} :: (\text{node} \Rightarrow \text{var set}) \Rightarrow (\text{node} \Rightarrow \text{node} \Rightarrow \text{var set}) \Rightarrow (\text{node \ LDCFG-node \ \times \ \text{var set}) set} \\
\text{for} & \; \text{Use} :: (\text{node} \Rightarrow \text{var set}) \text{ and } E :: \text{node} \text{ and } X :: \text{node} \\
\text{and} & \; H :: \text{var set and } L :: \text{var set} \\
\text{where} & \; \text{lift-Use-node}: \\
& V \in \text{Use } n \Rightarrow (\text{Node } n, V) \in \text{lift-Use-set } \text{Use } E X H L \\
& | \text{lift-Use-High}: \\
& V \in H \Rightarrow (\text{Node } E, V) \in \text{lift-Use-set } \text{Use } E X H L \\
& | \text{lift-Use-Low}: \\
& V \in L \Rightarrow (\text{Node } X, V) \in \text{lift-Use-set } \text{Use } E X H L \\
\end{align*}
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 af: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 source-node target-node kind Entry Exit) NewEntry
(lift-get-proc get-proc Main)
(lift-get-return-edges get-return-edges valid-edge source-node target-node kind) procs Main
⟨proof⟩

lemma lift-CFG-wf:
assumes af: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
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⟩

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 try kind
  (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 Entry Exit H L) (lift-Use Use Entry Exit H L)
  (lift-ParamDefs ParamDefs) (lift-ParamUses ParamUses)
⟨proof⟩
end

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:≡ lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit
  and lift-proc:≡ lift-get-proc get-proc Main
  and lift-return-edges:≡ lift-get-return-edges get-return-edges valid-edge sourcenode targetnode kind
  and lDef:≡ lift-Def Def Entry Exit H L
  and lUse:≡ lift-Use Use Entry Exit H L
  and lParamDefs:≡ lift-ParamDefs ParamDefs
  and 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 ∩ L = {} and H U L = UNIV
  shows NonInterferenceInterGraph src try kind lve NewEntry lget-proc
  lget-return-edges procs Main NewExit lDef lUse lParamDefs lParamUses H L
  (Node Entry) (Node Exit)
⟨proof⟩
end

References


