Slicing Guarantees Information Flow Noninterference

Daniel Wasserrab

April 19, 2020

Abstract

In this contribution, we show how correctness proofs for intraprocedural slicing [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-uf 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 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:  
\exists 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:  
\exists 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 Def[High]: 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→∗ n and inner-node n
obtains a’ as’ where as = a’¬as’ and (¬High-) as’→∗ n
and kind a’ = (λs. True) √
⟨proof⟩

lemma Exit-path-Low-path:
assumes n as→∗ (¬Exit-) and inner-node n
obtains a’ as’ where as = as′[a’] and (¬High-) n as’→∗ (¬Low-)
and kind a’ = (λs. True) √
⟨proof⟩

lemma not-Low-High: V /∈ L \implies V \in H
⟨proof⟩

lemma not-High-Low: V /∈ H \implies 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 ⇒ 'state ⇒ bool (infixl \(\approx_L\) 50)
where s \(\approx_L\) s’ ≡ ∀ V ∈ L. state-val s V = state-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 ∈ rv S (¬Entry-) and (¬High-) /∈ backward-slice S
shows V ∈ L
⟨proof⟩
lemma lowEquivalence-relevant-nodes-Entry:
assumes \( s \approx_L s' \) and \((-High-) \notin \text{backward-slice } S \)
shows \( \forall V \in \text{rv } S \ (-\text{Entry}) . \ \text{state-val } s \ V = \text{state-val } s' \ V \)
⟨proof⟩

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

2.3 The Correctness Proofs

In the following, we present two correctness proofs that slicing guarantees IFC noninterference. In both theorems, \((-High-) \notin \text{backward-slice } S \), where \((-Low-) \in S \), 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 \((-\text{Entry-}) \rightarrow^* (-\text{Exit-})\) 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 \((-\text{High-}) \notin \text{backward-slice } S \) and \((-\text{Low-}) \in S \) and \((-\text{Entry-}) \rightarrow^* (-\text{Exit-}) \) and \text{preds} (\text{kinds as}) \ s \) and \((-\text{Entry-}) \rightarrow^* (-\text{Exit-}) \) and \text{preds} (\text{kinds as'}) \ s' \) shows \( \text{transfers} (\text{kinds as}) \ s \approx_L \text{transfers} (\text{kinds as'}) \ s' \)
⟨proof⟩

theorem nonInterference-path:
assumes \( s \approx_L s' \) and \((-\text{High-}) \notin \text{backward-slice } S \) and \((-\text{Low-}) \in S \) and \((-\text{Entry-}) \rightarrow^* (-\text{Exit-}) \) and \text{preds} (\text{kinds as}) \ s \) and \((-\text{Entry-}) \rightarrow^* (-\text{Exit-}) \) and \text{preds} (\text{kinds as'}) \ s' \) shows \( \text{transfers} (\text{kinds as}) \ s \approx_L \text{transfers} (\text{kinds as'}) \ s' \)
⟨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 resulting states are again low equivalent:

locale NonInterferenceIntra =
The following theorem needs the explicit edge from \((-\text{High}-)\) to \(n\). An approach using a \(\text{init}\) predicate for initial statements, being reachable from \((-\text{High}-)\) via a \((\lambda s.\ \text{True})\) 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).

**Theorem nonInterference**: 
assumes \(s_1 \approx_L s_2\) and \((-\text{High}-) \notin \text{backward-slice} S\) and \((-\text{Low}-) \in S\) and valid-edge \(a\) and sourcenode \(a = (-\text{High}-)\) and targetnode \(a = n\) and \((c,s_1) \Rightarrow \langle c',s_1' \rangle\) and \((c,s_2) \Rightarrow \langle c',s_2' \rangle\) shows \(s_1' \approx_L s_2'\)

\[\text{proof}\]

**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 \(\text{NonInterferenceIntraGraph}\)
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 × ('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 ⇒}
\texttt{var set} ⇒ \texttt{var set} ⇒ (\texttt{node LDCF\-G-node} × \texttt{var}) set
for \texttt{Def}::\texttt{\{'node ⇒ \texttt{var set}\}} and \texttt{E::\texttt{\{'node and X::\texttt{\{'node and H::\texttt{\{'var set and L::\texttt{\{'var set}}

where \texttt{lift-Def-node}:
\[ V \in \text{Def} \, n \implies (\text{Node} \, n, V) \in \text{lift-Def-set} \, \text{Def} \, E \, X \, H \, L \]

\texttt{\textbar} \texttt{lift-Def-High}:
\[ V \in H \implies (\text{Node} \, E, V) \in \text{lift-Def-set} \, \text{Def} \, E \, X \, H \, L \]

abbreviation \texttt{lift-Def :: \texttt{\{'node ⇒ \texttt{var set}\}} ⇒ \texttt{\{'node ⇒ \texttt{\textbar} \texttt{var set ⇒ \texttt{\textbar} \texttt{node LDCF\-G-node ⇒ \texttt{\textbar} \texttt{var set}}

where \texttt{lift-Def \, \text{Def} \, E \, X \, H \, L \, n} \equiv \{ V. \, (n, V) \in \text{lift-Def-set} \, \text{Def} \, E \, X \, H \, L \}

\texttt{\textbar} \texttt{inductive-set lift-Use-set :: \texttt{\{'node ⇒ \texttt{var set}\}} ⇒ \texttt{\{'node ⇒ \texttt{\textbar} \texttt{var set ⇒ \texttt{\textbar} \texttt{node LDCF\-G-node ⇒ \texttt{\textbar} \texttt{var set}}

for \texttt{Use::\texttt{\{'node ⇒ \texttt{var set} and E::\texttt{\{'node and X::\texttt{\{'node and H::\texttt{\{'var set and L::\texttt{\{'var set}}

where \texttt{lift-Use-node}:
\[ V \in \text{Use} \, n \implies (\text{Node} \, n, V) \in \text{lift-Use-set} \, \text{Use} \, E \, X \, H \, L \]

\texttt{\textbar} \texttt{lift-Use-High}:
\[ V \in H \implies (\text{Node} \, E, V) \in \text{lift-Use-set} \, \text{Use} \, E \, X \, H \, L \]

\texttt{\textbar} \texttt{lift-Use-Low}:
\[ V \in L \implies (\text{Node} \, X, V) \in \text{lift-Use-set} \, \text{Use} \, E \, X \, H \, L \]

abbreviation \texttt{lift-Use :: \texttt{\{'node ⇒ \texttt{var set}\}} ⇒ \texttt{\{'node ⇒ \texttt{\textbar} \texttt{var set ⇒ \texttt{\textbar} \texttt{node LDCF\-G-node ⇒ \texttt{\textbar} \texttt{var set}}

where \texttt{lift-Use \, \text{Use} \, E \, X \, H \, L \, n} \equiv \{ V. \, (n, V) \in \text{lift-Use-set} \, \text{Use} \, E \, X \, H \, L \}

3.2 The lifting lemmas
3.2.1 Lifting the basic locales

abbreviation \texttt{src :: \texttt{\{'edge,\texttt{\{'node,\texttt{\{'state} \\texttt{LDCF\-G-edge ⇒ \texttt{\{'node LDCF\-G-node}

where \texttt{src a} \equiv \texttt{fst} \, a \}

abbreviation \texttt{trg :: \texttt{\{'edge,\texttt{\{'node,\texttt{\{'state} \\texttt{LDCF\-G-edge ⇒ \texttt{\{'node LDCF\-G-node}

where \texttt{trg a} \equiv \texttt{snd} \, (\texttt{snd} \, a) \}

definition \texttt{knd :: \texttt{\{'edge,\texttt{\{'node,\texttt{\{'state} \\texttt{LDCF\-G-edge ⇒ \texttt{\{'state edge-kind}

where \texttt{knd a} \equiv \texttt{fst} \, (\texttt{snd} \, a) \}

lemma \texttt{lift-CFG}:
\textbf{assumes $af$:} \textbf{CFGExit-wf} sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
\textbf{shows} CFG src trg
\begin{itemize}
\item[$\langle$proof$\rangle$] \textbf{CFGExit-wf} sourcenode targetnode kind valid-edge Entry Exit (lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit) NewEntry
\end{itemize}

\textbf{lemma lift-CFG-wf:}
\textbf{assumes $af$:} \textbf{CFGExit-wf} sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
\textbf{shows} CFG-wf src trg knd
\begin{itemize}
\item[$\langle$proof$\rangle$] \textbf{CFGExit-wf} sourcenode targetnode kind valid-edge Entry Exit (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val
\end{itemize}

\textbf{lemma lift-CFGExit:}
\textbf{assumes $af$:} \textbf{CFGExit-wf} sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
\textbf{shows} CFGExit src trg knd
\begin{itemize}
\item[$\langle$proof$\rangle$] \textbf{CFGExit-wf} sourcenode targetnode kind valid-edge Entry Exit NewEntry NewExit
\end{itemize}

\textbf{lemma lift-CFGExit-wf:}
\textbf{assumes $af$:} \textbf{CFGExit-wf} sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
\textbf{shows} CFGExit-wf src trg knd
\begin{itemize}
\item[$\langle$proof$\rangle$] \textbf{CFGExit-wf} sourcenode targetnode kind valid-edge Entry Exit (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L) state-val NewExit
\end{itemize}

\textbf{3.2.2 Lifting wod-backward-slice}

\textbf{lemma lift-wod-backward-slice:}
\textbf{fixes} valid-edge and sourcenode and targetnode and kind and Entry and Exit and Def and Use and H and L
\textbf{defines} lve:lve $\equiv$ 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
\textbf{assumes $af$:} \textbf{CFGExit-wf} sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
\textbf{and} $H \cap L = \{\}$ \textbf{and} $H \cup L = \text{UNIV}$
\textbf{shows} NonInterferenceIntraGraph src trg knd lve NewEntry lDef lUse state-val
\begin{itemize}
\item[$\langle$proof$\rangle$] (CFG-wf.wod-backward-slice src trg lve lDef lUse)
\end{itemize}
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`
- `pd:Postdomination` `sourcenode targetnode kind valid-edge Entry Exit`
- `inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx`

**Shows**:
- `(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`)
- `pd:Postdomination` `sourcenode targetnode kind valid-edge Entry Exit`
- `inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx`

**Shows**:
- `(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 `sourcenode targetnode valid-edge Exit`)
- `(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`

**Defines**:
- `lve:lift-valid-edge valid-edge sourcenode targetnode kind Entry Exit`
- `lDef:lift-Def Def Entry Exit H L`
- `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`)
- `pd:Postdomination` `sourcenode targetnode kind valid-edge Entry Exit`
- `inner:CFGExit.inner-node sourcenode targetnode valid-edge Entry Exit nx`
- `H ∩ L = {}` and `H ∪ L = UNIV`

**Shows**:
- `(NonInterferenceIntraGraph src try kind lve NewEntry lDef lUse state-val)` (PDG.PDG-BS src try lve lDef lUse)
- `(Postdomination.standard-control-dependence src try 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`
lemma lift-PDG-wcd:
assumes PDG:PDG sourcenode targetnode kind valid-edge Entry Def Use state-val Exit
(StrongPostdomination.weak-control-dependence source node target node valid-edge Exit)
and spd:StrongPostdomination source node target node kind valid-edge Entry Exit
and inner:CFGExit.inner-node source node target node valid-edge Entry Exit nz
shows StrongPostdomination src trg knd
(lift-valid-edge valid-edge source node target node kind Entry Exit) NewEntry 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 source node target node 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 source node target node kind valid-edge Entry Def Use state-val Exit
(StrongPostdomination.weak-control-dependence source node target node valid-edge Exit)
and spd:StrongPostdomination source node target node kind valid-edge Entry Exit
and inner:CFGExit.inner-node source node target node valid-edge Entry Exit nz
and H ∩ L = {} and H ∪ 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)


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
lemma final-edge:
  labels-nodes prog n Skip \implies prog \vdash n \uparrow id \to (\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.\text{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-})))
(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) source-node target-node 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-ax - HighLowDistinct HighLowUNIV]

lemma WCDNonInterference:
NonInterferenceIntra src trg knd
(lift-valid-edge (valid-edge prog) source-node target-node 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) source-node target-node 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) source-node target-node kind
(-Entry-) (-Exit-) NewExit))
reds (labels-LDCFG-nodes prog)
NewExit H L (LDCFG-node.Node (-Entry-)) (LDCFG-node.Node (-Exit-)) final
⟨proof⟩

end

References


interference Java bytecode verifier. In ESOP 2007, volume 4421 of


