Slicing Guarantees Information Flow
Noninterference

Daniel Wasserrab
August 28, 2014

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 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 pros 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 pros :: ('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 DefHigh: Def (-High-) = H
and UseHigh: Use (-High-) = H
and UseLow: Use (-Low-) = L
and HighLowDistinct: H ∩ L = {}
and HighLowUNIV: H ∪ L = UNIV

begin

lemma Low-neq-Exit: assumes L ≠ {} shows (-Low-) ≠ (-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→* 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 n − as'→* (-Low-)
and kind a' = (λs. True)√
⟨proof⟩

lemma not-Low-High: V /∈ L ⇒ V ∈ H
⟨proof⟩

lemma not-High-Low: V /∈ H ⇒ V ∈ 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. \textit{Low equivalence} groups those states in an equivalence class using the relation \(\approx_L\):

\[
\text{definition lowEquivalence :: ('var \to 'val) list \Rightarrow ('var \to 'val) list \Rightarrow bool}
\]
\[
\text{(infixl \(\approx_L\) 50)}
\]
\[
\text{where } s \approx_L s' \equiv \forall V \in L. \text{hd } s \text{ V} = \text{hd } s' \text{ V}
\]

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 \text{rv } S \text{ (CFG-node (-Entry-)) and (-High-) \notin HRB-slice } S \text{ ]}_{CFG}
\textbf{shows} \(V \in L\)
\textbf{⟨proof⟩}

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

2.3 The Correctness Proofs

In the following, we present two correctness proofs that slicing guarantees IFC noninterference. In both theorems, \(CFG\text{-node } (-High-) \notin HRB\text{-slice } S\), where \(CFG\text{-node } (-Low-) \in S\), 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:

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

state-$\text{val} (\text{transfers} (\text{slice-kinds} \ S \ m)) V =$
state-$\text{val} (\text{transfers} (\text{slice-kinds} \ S \ m')) V$

⟨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 $(\text{-Entry-}) \rightarrow^{\star} (-\text{Low-})$ and $\text{preds} (\text{slice-kinds} S \ m) [([cf,\text{undefined}])]$
and $(\text{-Entry-}) \rightarrow^{\star} (-\text{Low-})$ and $\text{preds} (\text{slice-kinds} S \ m') [([cf',\text{undefined}])]$
shows $\text{map} \ fst (\text{transfers} (\text{slice-kinds} S \ m)) [([cf,\text{undefined}])] \approx_{L} 
\text{map} \ fst (\text{transfers} (\text{slice-kinds} S \ m')) [([cf',\text{undefined}])]$

⟨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 $(\text{-Entry-}) \rightarrow^{\star} (-\text{Exit-})$ and $\text{preds} (\text{slice-kinds} S \ m) [([cf,\text{undefined}])]$
and $(\text{-Entry-}) \rightarrow^{\star} (-\text{Exit-})$ and $\text{preds} (\text{slice-kinds} S \ m') [([cf',\text{undefined}])]$
shows $\text{map} \ fst (\text{transfers} (\text{slice-kinds} S \ m)) [([cf,\text{undefined}])] \approx_{L} 
\text{map} \ fst (\text{transfers} (\text{slice-kinds} S \ m')) [([cf',\text{undefined}])]$

⟨proof⟩

**end**

The second theorem assumes that we have an 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** 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-kind
and valid-edge :: 'edge $\Rightarrow$ bool

5
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 $(\lambda s. \text{True})\sqrt{\text{edge}}$, does not work as the same statement could
be identified by several nodes, some initial, some not. E.g., in the program

```
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 $[cf_1] \approx_L [cf_2]$ and (-High-) $\notin [HRB-slice S] \ CFG$
and CFG-node (-Low-) $\in S$
and valid-edge $a$ and sourcenode $a = (-High-) \ and \ targetnode a = n$
and kind $a = (\lambda s. \text{True})\sqrt{\text{edge}}$ and $n \triangleq c$ and final $c'$
and $(c,[cf_1]) \Rightarrow (c',s_1)$ and $(c,[cf_2]) \Rightarrow (c',s_2)$
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

```ml
datatype 'node LDCFG-node = Node 'node |
| NewEntry |
| NewExit
```

```ml
type-synonym ('edge,'node,'var,'val,'ret,'pname) LDCFG-edge =
'dnode LDCFG-node × (('var,'val,'ret,'pname) edge-kind) × 'node LDCFG-node
```

3.1.2 Lifting basic definitions using 'edge and 'node

```ml
inductive lift-valid-edge :: ('edge ⇒ bool) ⇒ ('edge ⇒ 'node) ⇒ ('edge ⇒ 'node) ⇒ ('edge ⇒ ('var,'val,'ret,'pname) edge-kind) 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,(\ls. True) \\/,Node E)
⇒ lift-valid-edge valid-edge src trg knd E X e

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

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

```ml
lemma [simp]:¬ lift-valid-edge valid-edge src trg knd E X (Node E,et,Node X)
(proof)
```

```ml
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
```

7
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,'node,'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 (try a)); valid-edge a; a' ∈ get-return-edges a; e' = (Node (src a'),knd a',Node (try 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:
\[ V ∈ Def n \implies (Node n,V) ∈ lift-Def-set Def E X H L \]
| lift-Def-High:
\[ V ∈ H \implies (Node E,V) ∈ lift-Def-set Def E X H L \]
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:
\[ V ∈ Use n \implies (Node n,V) ∈ lift-Use-set Use E X H L \]
| lift-Use-High:
\[ V ∈ H \implies (Node E,V) ∈ lift-Use-set Use E X H L \]
| lift-Use-Low:
\[ V ∈ L \implies (Node X,V) ∈ lift-Use-set Use E X H L \]
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-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 sourcenode targetnode kind Entry Exit)
(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 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
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 (lift-Def Def Entry Exit H L) (lift-Use Use Entry Exit H L)
(lift-ParamDefs ParamDefs) (lift-ParamUses ParamUses)
⟨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
(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 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 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 Entry Exit H L

and lUse::lUse ≡ lift-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 ∩ L = {} and H ∪ L = UNIV

**Shows** NonInterferenceInterGraph src trg kind lve NewEntry lget-proc
lget-return-edges procs Main NewExit lDef lUse lParamDefs lParamUses H L
(Node Entry) (Node Exit)

**⟨proof⟩

**End**

**References**


