Bounded-Deducibility Security

Andrei Popescu Peter Lammich

Contents

1 Introduction

This is a formalization of *bounded-deducibility security* (*BD security*), a flexible notion of information-flow security applicable to arbitrary input–output automata. It generalizes Sutherland's classic notion of nondeducibility [1] by factoring in declassification bounds and triggers—whereas nondeducibility states that, in a system, information cannot flow between specified sources and sinks, BD security indicates upper bounds for the flow and triggers under which these upper bounds are no longer guaranteed.

BD security is introduced and discussed in detail in [2], where an application to the verification of a conference management system is also presented. This formalization only contains the abstract notion and its associated unwinding proof method, as discussed in Sections 4 and 5 from [2].

[1] D. Sutherland. A model of information. In 9th National Security Conference, pp. 175–183, 1986.
[2] S. Kanav, P. Lammich and A. Popescu. A Conference Management System with Verified Document Confidentiality. To appear in CAV 2014. Preprint available at http://www21.in.tum.de/~popescua/pdf/confsys.pdf

2 Trivia

lemma *measure-induct2*:
fixes *meas* :: *'a* ⇒ *'b* ⇒ *nat*
assumes ⋀*x1 x2*. (⋀*y1 y2*. *meas y1 y2* < *meas x1 x2* ⟹ *S y1 y2*) ⟹ *S x1 x2*
shows *S x1 x2*
⟨*proof*⟩

Right cons:

abbreviation *Rcons* (**infix** ## *70*) **where** *xs* ## *x* ≡ *xs* @ [*x*]

lemma *two-singl-Rcons*: [*a*,*b*] = [*a*] ## *b* ⟨*proof*⟩

3 IO Automaton

3.1 Preliminaries

Transitions

datatype (*'state*,*'act*,*'out*) *trans* =
Trans (*srcOf*: *'state*) (*actOf*: *'act*) (*outOf*: *'out*) (*tgtOf*: *'state*)

type-synonym (*'state*,*'act*,*'out*) *trace* = (*'state*,*'act*,*'out*) *trans list*

locale *IO-Automaton* =
fixes *istate* :: *'state*
and *step* :: *'state* ⇒ *'act* ⇒ *'out* ∗ *'state*
begin

3.2 Reachability and invariance

inductive *reach* :: *'state* ⇒ *bool* **where**
Istate: *reach istate*
|
Step: *reach s* ⟹ *reach* (*snd* (*step s a*))

lemma *reach-PairI*:
assumes *reach s* **and** *step s a* = (*ou*, *s'*)
shows *reach s'*
⟨*proof*⟩

definition *holdsIstate* :: (*'state* ⇒ *bool*) ⇒ *bool* **where**
holdsIstate $\varphi \equiv \varphi$ *istate*

definition *invar* :: (*'state* ⇒ *bool*) ⇒ *bool* **where**
invar $\varphi \equiv \forall$ *s a. reach s* $\wedge$ φ *s* $\longrightarrow$ φ (*snd* (*step s a*))

lemma *holdsIstate-invar*:
assumes *h*: *holdsIstate* φ **and** *i*: *invar* φ **and** *a*: *reach s*
shows φ *s*
⟨*proof*⟩

3.3 System traces

definition *out* :: *'state* ⇒ *'act* ⇒ *'out* **where** *out s a* ≡ *fst* (*step s a*)
definition *eff* :: *'state* ⇒ *'act* ⇒ *'state* **where** *eff s a* ≡ *snd* (*step s a*)

primrec *validTrans* :: (*'state*,*'act*,*'out*) *trans* ⇒ *bool* **where**
validTrans (*Trans s a ou s'*) = (*step s a* = (*ou*, *s'*))

lemma *validTrans*:
validTrans trn =
(*step* (*srcOf trn*) (*actOf trn*) = (*outOf trn*, *tgtOf trn*))
⟨*proof*⟩

inductive *valid* :: (*'state*,*'act*,*'out*) *trace* ⇒ *bool* **where**
Singl[*simp*,*intro*!]:
validTrans trn
⟹
valid [*trn*]
|
Cons[*intro*]:
⟦*validTrans trn*; *tgtOf trn* = *srcOf* (*hd tr*); *valid tr*⟧
⟹
valid (*trn* # *tr*)

inductive-cases *valid-SinglE*[*elim*!]: *valid* [*trn*]
inductive-cases *valid-ConsE*[*elim*]: *valid* (*trn* # *tr*)

inductive *valid2* :: (*'state*,*'act*,*'out*) *trace* ⇒ *bool* **where**
Singl[*simp*,*intro*!]:
validTrans trn
⟹
valid2 [*trn*]
|

Rcons[*intro*]:
⟦*valid2 tr*; *tgtOf* (*last tr*) = *srcOf trn*; *validTrans trn*⟧
⟹
valid2 (*tr* ## *trn*)

inductive-cases *valid2-SinglE*[*elim*!]: *valid2* [*trn*]
inductive-cases *valid2-RconsE*[*elim*]: *valid2* (*tr* ## *trn*)

lemma *Nil-not-valid*[*simp*]: ¬ *valid* []
⟨*proof*⟩

lemma *Nil-not-valid2*[*simp*]: ¬ *valid2* []
⟨*proof*⟩

lemma *valid-Rcons*:
assumes *valid tr* **and** *tgtOf* (*last tr*) = *srcOf trn* **and** *validTrans trn*
shows *valid* (*tr* ## *trn*)
⟨*proof*⟩

lemma *valid-hd-Rcons*[*simp*]:
assumes *valid tr*
shows *hd* (*tr* ## *tran*) = *hd tr*
⟨*proof*⟩

lemma *valid2-hd-Rcons*[*simp*]:
assumes *valid2 tr*
shows *hd* (*tr* ## *tran*) = *hd tr*
⟨*proof*⟩

lemma *valid2-last-Cons*[*simp*]:
assumes *valid2 tr*
shows *last* (*tran* # *tr*) = *last tr*
⟨*proof*⟩

lemma *valid2-Cons*:
assumes *valid2 tr* **and** *tgtOf trn* = *srcOf* (*hd tr*) **and** *validTrans trn*
shows *valid2* (*trn* # *tr*)
⟨*proof*⟩

lemma *valid-valid2*: *valid* = *valid2*
⟨*proof*⟩

lemmas *valid2-valid* = *valid-valid2*[*symmetric*]

definition *validFrom* :: *'state* ⇒ (*'state*,*'act*,*'out*) *trace* ⇒ *bool* **where**
validFrom s tr ≡ *tr* = [] ∨ (*valid tr* ∧ *srcOf* (*hd tr*) = *s*)

lemma *validFrom-Nil*[*simp*,*intro*!]: *validFrom s* []
⟨*proof*⟩

lemma *validFrom-valid*[*simp*,*intro*]: *valid tr* ∧ *srcOf* (*hd tr*) = *s* ⟹ *validFrom s tr*
⟨*proof*⟩

lemma *validFrom-validTrans*[*intro*]:
assumes *validTrans* (*Trans s a ou s'*) **and** *validFrom s' tr*
shows *validFrom s* (*Trans s a ou s'* # *tr*)
⟨*proof*⟩

3.4 Traces versus reachability

lemma *valid-reach-src-tgt*:
assumes *valid tr* **and** *reach* (*srcOf* (*hd tr*))
shows *reach* (*tgtOf* (*last tr*))
⟨*proof*⟩

lemma *valid-init-reach*:
assumes *valid tr* **and** *srcOf* (*hd tr*) = *istate*
shows *reach* (*tgtOf* (*last tr*))
⟨*proof*⟩

lemma *Trans-fst-sndI*:
valid [*Trans s a* (*fst* (*step s a*)) (*snd* (*step s a*))]
⟨*proof*⟩

lemma *reach-init-valid*:
assumes *reach s*
shows
s = *istate*
∨
(∃ *tr*. *valid tr* ∧ *srcOf* (*hd tr*) = *istate* ∧ *tgtOf* (*last tr*) = *s*)
⟨*proof*⟩

lemma *reach-validFrom*:
assumes *reach s'*
shows ∃ *s tr*. *s* = *istate* ∧ (*s* = *s'* ∨ (*validFrom s tr* ∧ *tgtOf* (*last tr*) = *s'*))
⟨*proof*⟩

4 BD Security

4.1 Definition

declare *Let-def*[*simp*]

no-notation *relcomp* (**infixr** *O* 75)

abbreviation *never* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ *bool* **where** *never U* ≡ *list-all* (λ *a*. ¬ *U a*)

function *filtermap* ::

$(('state,'act,'out)\ trans \Rightarrow bool) \Rightarrow (('state,'act,'out)\ trans \Rightarrow 'a) \Rightarrow ('state,'act,'out)\ trace \Rightarrow 'a\ list$
where
filtermap pred func [] = []
|
¬ *pred trn* ⟹ *filtermap pred func* (*trn* # *tr*) = *filtermap pred func tr*
|
pred trn ⟹ *filtermap pred func* (*trn* # *tr*) = *func trn* # *filtermap pred func tr*
⟨*proof*⟩
termination ⟨*proof*⟩

locale *BD-Security* = *IO-Automaton istate step*
for *istate* :: *'state* **and** *step* :: *'state* ⇒ *'act* ⇒ *'out* × *'state*
+
fixes
φ :: $('state,'act,'out)\ trans$ => *bool* **and** *f* :: $('state,'act,'out)\ trans \Rightarrow 'value$
and
γ :: $('state,'act,'out)\ trans$ => *bool* **and** *g* :: $('state,'act,'out)\ trans \Rightarrow 'obs$
and
T :: $('state,'act,'out)\ trans \Rightarrow bool$
and
B :: *'value list* ⇒ *'value list* ⇒ *bool*
begin

definition *V* :: $('state,'act,'out)\ trace \Rightarrow 'value\ list$ **where** *V* ≡ *filtermap* φ *f*

definition *O* :: $('state,'act,'out)\ trace \Rightarrow 'obs\ list$ **where** *O* ≡ *filtermap* γ *g*

lemma *V-simps*[*simp*]:
V [] = [] ¬ φ *trn* ⟹ *V* (*trn* # *tr*) = *V tr* φ *trn* ⟹ *V* (*trn* # *tr*) = *f trn* # *V tr*
⟨*proof*⟩

lemma *O-simps*[*simp*]:
O [] = [] ¬ γ *trn* ⟹ *O* (*trn* # *tr*) = *O tr* γ *trn* ⟹ *O* (*trn* # *tr*) = *g trn* # *O tr*
⟨*proof*⟩

inductive *reachNT*:: *'state* ⇒ *bool* **where**
Istate: *reachNT istate*
|
Step:
⟦*reachNT* (*srcOf trn*); *step* (*srcOf trn*) (*actOf trn*) = (*outOf trn*, *tgtOf trn*); ¬ *T trn*⟧
⟹ *reachNT* (*tgtOf trn*)

lemma *reachNT-PairI*:
assumes *reachNT s* **and** *step s a* = (*ou*, *s'*) **and** ¬ *T* (*Trans s a ou s'*)
shows *reachNT s'*
⟨*proof*⟩

lemma *reachNT-reach*: **assumes** *reachNT s* **shows** *reach s*
⟨*proof*⟩

lemma *reachNT-stateO-aux*:
assumes *reachNT s*
shows $s = istate \vee (\exists\, sh\ a\ ou.\ reach\ sh \wedge step\ sh\ a = (ou,s) \wedge \neg T\ (Trans\ sh\ a\ ou\ s))$
⟨*proof*⟩

lemma *reachNT-state-cases*[*cases set*,*consumes 1*, *case-names init step*]:
assumes *reachNT s*
obtains $s = istate$
| *sh a ou* **where** *reach sh step sh a* $= (ou,s)$ $\neg T\ (Trans\ sh\ a\ ou\ s)$
⟨*proof*⟩

definition *invarNT* **where**
$invarNT\ Inv \equiv \forall\ s\ a\ ou\ s'.\ reachNT\ s \wedge Inv\ s \wedge \neg\ T\ (Trans\ s\ a\ ou\ s') \wedge step\ s\ a = (ou,s') \longrightarrow Inv\ s'$

lemma *invarNT-disj*:
assumes *invarNT Inv1* **and** *invarNT Inv2*
shows *invarNT* $(\lambda\ s.\ Inv1\ s \vee Inv2\ s)$
⟨*proof*⟩

lemma *invarNT-conj*:
assumes *invarNT Inv1* **and** *invarNT Inv2*
shows *invarNT* $(\lambda\ s.\ Inv1\ s \wedge Inv2\ s)$
⟨*proof*⟩

lemma *holdsIstate-invarNT*:
assumes *h*: *holdsIstate Inv* **and** *i*: *invarNT Inv* **and** *a*: *reachNT s*
shows *Inv s*
⟨*proof*⟩

definition *secure* :: *bool* **where**
$secure \equiv$
$\forall\ tr\ vl\ vl1.$
$\quad validFrom\ istate\ tr \wedge never\ T\ tr \wedge B\ vl\ vl1 \wedge V\ tr = vl \longrightarrow$
$\quad (\exists\ tr1.\ validFrom\ istate\ tr1 \wedge O\ tr1 = O\ tr \wedge V\ tr1 = vl1)$

lemma *V-iff-non-φ*[*simp*]: $V\ (trn\ \#\ tr) = V\ tr \longleftrightarrow \neg\ \varphi\ trn$
⟨*proof*⟩

lemma *V-imp-φ*: $V\ (trn\ \#\ tr) = v\ \#\ V\ tr \Longrightarrow \varphi\ trn$
⟨*proof*⟩

lemma *V-imp-Nil*: $V\ (trn\ \#\ tr) = [] \Longrightarrow V\ tr = []$
⟨*proof*⟩

lemma *V-iff-Nil*[*simp*]: $V\ (trn\ \#\ tr) = [] \longleftrightarrow \neg\ \varphi\ trn \wedge\ V\ tr = []$
⟨*proof*⟩

end

4.2 Unwinding proof method

context *BD-Security*
begin

definition *consume* :: ('*state*,'*act*,'*out*) *trans* ⇒ '*value list* ⇒ '*value list* ⇒ *bool* **where**
consume trn vl vl' ≡
if φ *trn then vl* ≠ [] ∧ *f trn* = *hd vl* ∧ *vl'* = *tl vl*
else vl' = *vl*

lemma *length-consume*[*simp*]:
consume trn vl vl' ⟹ *length vl'* < *Suc* (*length vl*)
⟨*proof*⟩

lemma *ex-consume-*φ:
assumes ¬ φ *trn*
obtains *vl'* **where** *consume trn vl vl'*
⟨*proof*⟩

lemma *ex-consume-NO*:
assumes *vl* ≠ [] **and** *f trn* = *hd vl*
obtains *vl'* **where** *consume trn vl vl'*
⟨*proof*⟩

definition *iaction* **where**
iaction Δ *s vl s1 vl1* ≡
∃ *a1 ou1 s1' vl1'*.
let trn1 = *Trans s1 a1 ou1 s1' in*
validTrans trn1 ∧
φ *trn1* ∧ *consume trn1 vl1 vl1'* ∧
¬ γ *trn1*
∧
Δ *s vl s1' vl1'*

lemma *iactionI*[*intro?*]:
assumes *step s1 a1* = (*ou1*, *s1'*) **and** φ (*Trans s1 a1 ou1 s1'*)
and *consume* (*Trans s1 a1 ou1 s1'*) *vl1 vl1'*
and ¬ γ (*Trans s1 a1 ou1 s1'*) **and** Δ *s vl s1' vl1'*
shows *iaction* Δ *s vl s1 vl1*
⟨*proof*⟩

definition *match* **where**
match Δ *s s1 vl1 a ou s' vl'* ≡

$\exists$ *a1 ou1 s1′ vl1′.*
let trn = Trans s a ou s′; trn1 = Trans s1 a1 ou1 s1′ in
validTrans trn1 $\wedge$
consume trn1 vl1 vl1′ $\wedge$
γ *trn* $=$ γ *trn1* $\wedge$ (γ *trn* $\longrightarrow$ *g trn* $=$ *g trn1*) $\wedge$
Δ *s′ vl′ s1′ vl1′*

lemma *matchI*[*intro?*]:
assumes *validTrans* (*Trans s1 a1 ou1 s1′*)
and *consume* (*Trans s1 a1 ou1 s1′*) *vl1 vl1′* **and** γ (*Trans s a ou s′*) $=$ γ (*Trans s1 a1 ou1 s1′*)
and γ (*Trans s a ou s′*) $\Longrightarrow$ *g* (*Trans s a ou s′*) $=$ *g* (*Trans s1 a1 ou1 s1′*)
and Δ *s′ vl′ s1′ vl1′*
shows *match* Δ *s s1 vl1 a ou s′ vl′*
⟨*proof*⟩

definition *ignore* **where**
ignore Δ *s s1 vl1 a ou s′ vl′* $\equiv$
$\neg$ γ (*Trans s a ou s′*) $\wedge$
Δ *s′ vl′ s1 vl1*

lemma *ignoreI*[*intro?*]:
assumes $\neg$ γ (*Trans s a ou s′*) **and** Δ *s′ vl′ s1 vl1*
shows *ignore* Δ *s s1 vl1 a ou s′ vl′*
⟨*proof*⟩

definition *reaction* **where**
reaction Δ *s vl s1 vl1* $\equiv$
$\forall$ *a ou s′ vl′.*
let trn = Trans s a ou s′ in
validTrans trn $\wedge$ $\neg$ *T trn* $\wedge$
consume trn vl vl′
$\longrightarrow$
match Δ *s s1 vl1 a ou s′ vl′*
$\vee$
ignore Δ *s s1 vl1 a ou s′ vl′*

lemma *reactionI*[*intro?*]:
assumes
$\bigwedge$*a ou s′ vl′.*
⟦*step s a* $=$ (*ou*, *s′*); $\neg$ *T* (*Trans s a ou s′*);
consume (*Trans s a ou s′*) *vl vl′*⟧
$\Longrightarrow$
match Δ *s s1 vl1 a ou s′ vl′* $\vee$ *ignore* Δ *s s1 vl1 a ou s′ vl′*
shows *reaction* Δ *s vl s1 vl1*
⟨*proof*⟩

definition *exit* :: *'state* $\Rightarrow$ *'value* $\Rightarrow$ *bool* **where**
exit s v $\equiv$ $\forall$ *tr trn. validFrom s* (*tr ##* *trn*) $\wedge$ *never T* (*tr ##* *trn*) $\wedge$ φ *trn* $\longrightarrow$ *f trn* $\neq$ *v*

lemma *exit-coind*:
assumes K: $K\ s$
and I: $\bigwedge\ trn.\ [\![K\ (srcOf\ trn);\ validTrans\ trn;\ \neg\ T\ trn]\!]$
$\Longrightarrow (\varphi\ trn \longrightarrow f\ trn \neq v) \wedge K\ (tgtOf\ trn)$
shows *exit s v*
⟨*proof*⟩

definition *noVal* **where**
noVal K v ≡
$\forall\ s\ a\ ou\ s'.\ reachNT\ s \wedge K\ s \wedge step\ s\ a = (ou,s') \wedge \varphi\ (Trans\ s\ a\ ou\ s') \longrightarrow f\ (Trans\ s\ a\ ou\ s') \neq v$

lemma *noVal-disj*:
assumes *noVal Inv1 v* **and** *noVal Inv2 v*
shows *noVal* (λ *s*. *Inv1 s* ∨ *Inv2 s*) *v*
⟨*proof*⟩

lemma *noVal-conj*:
assumes *noVal Inv1 v* **and** *noVal Inv2 v*
shows *noVal* (λ *s*. *Inv1 s* ∧ *Inv2 s*) *v*
⟨*proof*⟩

definition *noφ* **where**
noφ K ≡ ∀ *s a ou s'*. *reachNT s* ∧ *K s* ∧ *step s a* = (*ou*,*s'*) ⟶ ¬ φ (*Trans s a ou s'*)

lemma *noφ-noVal*: *noφ K* ⟹ *noVal K v*
⟨*proof*⟩

lemma *exitI*[*consumes 2*, *induct pred*: *exit*]:
assumes *rs*: *reachNT s* **and** *K*: *K s*
and *I*:
$\bigwedge\ s\ a\ ou\ s'.$
$[\![reach\ s;\ reachNT\ s;\ step\ s\ a = (ou,s');\ K\ s]\!]$
$\Longrightarrow (\varphi\ (Trans\ s\ a\ ou\ s') \longrightarrow f\ (Trans\ s\ a\ ou\ s') \neq v) \wedge K\ s'$
shows *exit s v*
⟨*proof*⟩

lemma *exitI2*:
assumes *rs*: *reachNT s* **and** *K*: *K s*
and *invarNT K* **and** *noVal K v*
shows *exit s v*
⟨*proof*⟩

definition *noVal2* **where**
noVal2 K v ≡

$\forall\ s\ a\ ou\ s'.\ reachNT\ s \wedge K\ s\ v \wedge step\ s\ a = (ou,s') \wedge \varphi\ (Trans\ s\ a\ ou\ s') \longrightarrow f\ (Trans\ s\ a\ ou\ s') \neq v$

lemma *noVal2-disj*:
assumes *noVal2 Inv1 v* **and** *noVal2 Inv2 v*
shows *noVal2* $(\lambda\ s\ v.\ Inv1\ s\ v \vee Inv2\ s\ v)\ v$
⟨*proof*⟩

lemma *noVal2-conj*:
assumes *noVal2 Inv1 v* **and** *noVal2 Inv2 v*
shows *noVal2* $(\lambda\ s\ v.\ Inv1\ s\ v \wedge Inv2\ s\ v)\ v$
⟨*proof*⟩

lemma *noVal-noVal2*: $noVal\ K\ v \Longrightarrow noVal2\ (\lambda\ s\ v.\ K\ s)\ v$
⟨*proof*⟩

lemma *exitI-noVal2*[*consumes 2*, *induct pred*: *exit*]:
assumes *rs*: *reachNT s* **and** *K*: *K s v*
and *I*:
$\bigwedge\ s\ a\ ou\ s'.$
$[\![reach\ s;\ reachNT\ s;\ step\ s\ a = (ou,s');\ K\ s\ v]\!]$
$\Longrightarrow (\varphi\ (Trans\ s\ a\ ou\ s') \longrightarrow f\ (Trans\ s\ a\ ou\ s') \neq v) \wedge K\ s'\ v$
shows *exit s v*
⟨*proof*⟩

lemma *exitI2-noVal2*:
assumes *rs*: *reachNT s* **and** *K*: *K s v*
and *invarNT* $(\lambda\ s.\ K\ s\ v)$ **and** *noVal2 K v*
shows *exit s v*
⟨*proof*⟩

lemma *exit-validFrom*:
assumes *vl*: $vl \neq []$ **and** *i*: *exit s* (*hd vl*) **and** *v*: *validFrom s tr* **and** *V*: *V tr* = *vl*
and *T*: *never T tr*
shows *False*
⟨*proof*⟩

definition *unwind* **where**
unwind $\Delta \equiv$
$\forall\ s\ vl\ s1\ vl1.$
reachNT s $\wedge$ *reach s1* $\wedge\ \Delta\ s\ vl\ s1\ vl1$
$\longrightarrow$
$(vl \neq [] \wedge exit\ s\ (hd\ vl))$
$\vee$
iaction $\Delta\ s\ vl\ s1\ vl1$
$\vee$
$((vl \neq [] \vee vl1 = []) \wedge reaction\ \Delta\ s\ vl\ s1\ vl1)$

lemma *unwindI*[*intro?*]:
assumes
$\bigwedge$ *s vl s1 vl1*.
  ⟦*reachNT s*; *reach s1*; Δ *s vl s1 vl1*⟧
  ⟹
  (*vl* ≠ [] ∧ *exit s* (*hd vl*))
  ∨
  iaction Δ *s vl s1 vl1*
  ∨
  ((*vl* ≠ [] ∨ *vl1* = []) ∧ *reaction* Δ *s vl s1 vl1*)
shows *unwind* Δ
⟨*proof*⟩

lemma *unwind-trace*:
assumes *unwind*: *unwind* Δ **and** *reachNT s* **and** *reach s1* **and** Δ *s vl s1 vl1*
and *validFrom s tr* **and** *never T tr* **and** *V tr* = *vl*
shows ∃ *tr1*. *validFrom s1 tr1* ∧ *O tr1* = *O tr* ∧ *V tr1* = *vl1*
⟨*proof*⟩

theorem *unwind-secure*:
assumes *init*: $\bigwedge$ *vl vl1*. *B vl vl1* ⟹ Δ *istate vl istate vl1*
and *unwind*: *unwind* Δ
shows *secure*
⟨*proof*⟩

5 Compositional Reasoning

context *BD-Security* **begin**

5.1 Preliminaries

definition *disjAll* Δ*s s vl s1 vl1* ≡ (∃ Δ ∈ Δ*s*. Δ *s vl s1 vl1*)

lemma *disjAll-simps*[*simp*]:
  disjAll {} ≡ λ- - - -. *False*
  disjAll (*insert* Δ Δ*s*) ≡ λ*s vl s1 vl1*. Δ *s vl s1 vl1* ∨ *disjAll* Δ*s s vl s1 vl1*
  ⟨*proof*⟩

lemma *iaction-mono*:
assumes *1*: *iaction* Δ *s vl s1 vl1* **and** *2*: $\bigwedge$ *s vl s1 vl1*. Δ *s vl s1 vl1* ⟹ Δ′ *s vl s1 vl1*
shows *iaction* Δ′ *s vl s1 vl1*
⟨*proof*⟩

lemma *match-mono*:
assumes *1*: *match* Δ *s s1 vl1 a ou s′ vl′* **and** *2*: $\bigwedge$ *s vl s1 vl1*. Δ *s vl s1 vl1* ⟹ Δ′ *s vl s1 vl1*
shows *match* Δ′ *s s1 vl1 a ou s′ vl′*

⟨*proof*⟩

lemma *ignore-mono*:
assumes *1*: *ignore* Δ *s s1 vl1 a ou s′ vl′* **and** *2*: ⋀ *s vl s1 vl1*. Δ *s vl s1 vl1* ⟹ Δ′ *s vl s1 vl1*
shows *ignore* Δ′ *s s1 vl1 a ou s′ vl′*
⟨*proof*⟩

lemma *reaction-mono*:
assumes *1*: *reaction* Δ *s vl s1 vl1* **and** *2*: ⋀ *s vl s1 vl1*. Δ *s vl s1 vl1* ⟹ Δ′ *s vl s1 vl1*
shows *reaction* Δ′ *s vl s1 vl1*
⟨*proof*⟩

5.2 Decomposition into an arbitrary network of components

definition *unwind-to* **where**
unwind-to Δ Δ*s* ≡
∀ *s vl s1 vl1*.
reachNT s ∧ *reach s1* ∧ Δ *s vl s1 vl1*
⟶
vl ≠ [] ∧ *exit s* (*hd vl*)
∨
iaction (*disjAll* Δ*s*) *s vl s1 vl1*
∨
(*vl* ≠ [] ∨ *vl1* = []) ∧ *reaction* (*disjAll* Δ*s*) *s vl s1 vl1*

lemma *unwind-toI*[*intro?*]:
assumes
⋀ *s vl s1 vl1*.
⟦*reachNT s*; *reach s1*; Δ *s vl s1 vl1*⟧
⟹
vl ≠ [] ∧ *exit s* (*hd vl*)
∨
iaction (*disjAll* Δ*s*) *s vl s1 vl1*
∨
(*vl* ≠ [] ∨ *vl1* = []) ∧ *reaction* (*disjAll* Δ*s*) *s vl s1 vl1*
shows *unwind-to* Δ Δ*s*
⟨*proof*⟩

lemma *unwind-dec*:
assumes *ne*: ⋀ Δ. Δ ∈ Δ*s* ⟹ *next* Δ ⊆ Δ*s* ∧ *unwind-to* Δ (*next* Δ)
shows *unwind* (*disjAll* Δ*s*) (**is** *unwind ?*Δ)
⟨*proof*⟩

lemma *init-dec*:
assumes Δ*0*: Δ*0* ∈ Δ*s*
and *i*: ⋀ *vl vl1*. *B vl vl1* ⟹ Δ*0 istate vl istate vl1*
shows ∀ *vl vl1*. *B vl vl1* ⟶ *disjAll* Δ*s istate vl istate vl1*
⟨*proof*⟩

theorem *unwind-dec-secure*:
assumes $\Delta 0$: $\Delta 0 \in \Delta s$
and *i*: $\bigwedge$ *vl vl1*. *B vl vl1* $\Longrightarrow$ $\Delta 0$ *istate vl istate vl1*
and *ne*: $\bigwedge \Delta$. $\Delta \in \Delta s \Longrightarrow$ *next* $\Delta \subseteq \Delta s \wedge$ *unwind-to* Δ (*next* Δ)
shows *secure*
⟨*proof*⟩

5.3 A customization for linear modular reasoning

definition *unwind-cont* **where**
unwind-cont Δ $\Delta s \equiv$
$\forall$ *s vl s1 vl1*.
reachNT s $\wedge$ *reach s1* $\wedge$ Δ *s vl s1 vl1*
$\longrightarrow$
iaction (*disjAll* Δs) *s vl s1 vl1*
$\vee$
((*vl* $\neq$ [] $\vee$ *vl1* = []) $\wedge$ *reaction* (*disjAll* Δs) *s vl s1 vl1*)

lemma *unwind-contI*[*intro?*]:
assumes
$\bigwedge$ *s vl s1 vl1*.
⟦*reachNT s*; *reach s1*; Δ *s vl s1 vl1*⟧
$\Longrightarrow$
iaction (*disjAll* Δs) *s vl s1 vl1*
$\vee$
((*vl* $\neq$ [] $\vee$ *vl1* = []) $\wedge$ *reaction* (*disjAll* Δs) *s vl s1 vl1*)
shows *unwind-cont* Δ Δs
⟨*proof*⟩

definition *unwind-exit* **where**
unwind-exit $\Delta e \equiv$
$\forall$ *s vl s1 vl1*.
reachNT s $\wedge$ *reach s1* $\wedge$ Δe *s vl s1 vl1*
$\longrightarrow$
vl $\neq$ [] $\wedge$ *exit s* (*hd vl*)

lemma *unwind-exitI*[*intro?*]:
assumes
$\bigwedge$ *s vl s1 vl1*.
⟦*reachNT s*; *reach s1*; Δe *s vl s1 vl1*⟧
$\Longrightarrow$
vl $\neq$ [] $\wedge$ *exit s* (*hd vl*)
shows *unwind-exit* Δe
⟨*proof*⟩

fun *allConsec* :: *'a list* $\Rightarrow$ (*'a* $*$ *'a*) *set* **where**
allConsec [] = {}
| *allConsec* [*a*] = {}

| *allConsec* (*a* # *b* # *as*) = *insert* (*a*,*b*) (*allConsec* (*b*#*as*))

lemma *set-allConsec*:
assumes $\Delta \in$ *set* $\Delta s'$ **and** $\Delta s = \Delta s' \,\#\#\, \Delta 1$
shows $\exists\ \Delta 2.\ (\Delta,\Delta 2) \in$ *allConsec* Δs
⟨*proof*⟩

lemma *allConsec-set*:
assumes $(\Delta 1,\Delta 2) \in$ *allConsec* Δs
shows $\Delta 1 \in$ *set* $\Delta s \wedge \Delta 2 \in$ *set* Δs
⟨*proof*⟩

theorem *unwind-decomp-secure*:
assumes *n*: $\Delta s \neq []$
and *i*: $\bigwedge$ *vl vl1*. *B vl vl1* $\Longrightarrow$ *hd* Δs *istate vl istate vl1*
and *c*: $\bigwedge\ \Delta 1\ \Delta 2.\ (\Delta 1,\Delta 2) \in$ *allConsec* $\Delta s \Longrightarrow$ *unwind-cont* $\Delta 1$ $\{\Delta 1, \Delta 2, \Delta e\}$
and *l*: *unwind-cont* (*last* Δs) $\{$*last* $\Delta s, \Delta e\}$
and *e*: *unwind-exit* Δe
shows *secure*
⟨*proof*⟩

5.4 Instances

corollary *unwind-decomp3-secure*:
assumes
i: $\bigwedge$ *vl vl1*. *B vl vl1* $\Longrightarrow$ $\Delta 1$ *istate vl istate vl1*
and *c1*: *unwind-cont* $\Delta 1$ $\{\Delta 1, \Delta 2, \Delta e\}$
and *c2*: *unwind-cont* $\Delta 2$ $\{\Delta 2, \Delta 3, \Delta e\}$
and *l*: *unwind-cont* $\Delta 3$ $\{\Delta 3, \Delta e\}$
and *e*: *unwind-exit* Δe
shows *secure*
⟨*proof*⟩

corollary *unwind-decomp4-secure*:
assumes
i: $\bigwedge$ *vl vl1*. *B vl vl1* $\Longrightarrow$ $\Delta 1$ *istate vl istate vl1*
and *c1*: *unwind-cont* $\Delta 1$ $\{\Delta 1, \Delta 2, \Delta e\}$
and *c2*: *unwind-cont* $\Delta 2$ $\{\Delta 2, \Delta 3, \Delta e\}$
and *c3*: *unwind-cont* $\Delta 3$ $\{\Delta 3, \Delta 4, \Delta e\}$
and *l*: *unwind-cont* $\Delta 4$ $\{\Delta 4, \Delta e\}$
and *e*: *unwind-exit* Δe
shows *secure*
⟨*proof*⟩

corollary *unwind-decomp5-secure*:
assumes
i: $\bigwedge$ *vl vl1*. *B vl vl1* $\Longrightarrow$ $\Delta 1$ *istate vl istate vl1*

and *c1*: *unwind-cont* $\Delta 1$ $\{\Delta 1, \Delta 2, \Delta e\}$
and *c2*: *unwind-cont* $\Delta 2$ $\{\Delta 2, \Delta 3, \Delta e\}$
and *c3*: *unwind-cont* $\Delta 3$ $\{\Delta 3, \Delta 4, \Delta e\}$
and *c4*: *unwind-cont* $\Delta 4$ $\{\Delta 4, \Delta 5, \Delta e\}$
and *l*: *unwind-cont* $\Delta 5$ $\{\Delta 5, \Delta e\}$
and *e*: *unwind-exit* Δe
shows *secure*
⟨*proof*⟩