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−
let *?m* = λ *x1x2*. *meas* (*fst x1x2*) (*snd x1x2*) **let** *?S* = λ *x1x2*. *S* (*fst x1x2*) (*snd x1x2*)
have *?S* (*x1*,*x2*)
apply(*rule measure-induct*[*of ?m ?S*])
using *assms* **by** (*metis fst-conv snd-conv*)
thus *?thesis* **by** *auto*
qed

Right cons:

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

lemma *two-singl-Rcons*: [*a*,*b*] = [*a*] ## *b* **by** *auto*

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* $\Longrightarrow$ *reach* (*snd* (*step s a*))

lemma *reach-PairI*:
assumes *reach s* **and** *step s a* = (*ou*, *s'*)
shows *reach s'*
by (*metis Step assms assms snd-eqD*)

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*
using *a* **apply** (*induct rule*: *reach.induct*)
using *h i* **unfolding** *holdsIstate-def invar-def* **by** *auto*

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*))
by (*cases trn*) *auto*

inductive *valid* :: ('*state*,'*act*,'*out*) *trace* ⇒ *bool* **where**
Singl[*simp*,*intro*!]:
validTrans trn
$\Longrightarrow$
valid [*trn*]
|
Cons[*intro*]:
⟦*validTrans trn*; *tgtOf trn* = *srcOf* (*hd tr*); *valid tr*⟧
$\Longrightarrow$
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* []
by (*metis valid.simps neq-Nil-conv*)

lemma *Nil-not-valid2*[*simp*]: ¬ *valid2* []
by (*metis valid2.cases append-Nil butlast.simps butlast-snoc not-Cons-self2*)

lemma *valid-Rcons*:
assumes *valid tr* **and** *tgtOf* (*last tr*) = *srcOf trn* **and** *validTrans trn*
shows *valid* (*tr* ## *trn*)
using *assms* **proof**(*induct arbitrary*: *trn*)
case (*Cons trn tr trna*)
thus *?case* **by** (*cases tr*) (*auto intro*: *valid.Cons*)
qed(*auto intro*: *valid.intros*)

lemma *valid-hd-Rcons*[*simp*]:
assumes *valid tr*
shows *hd* (*tr* ## *tran*) = *hd tr*
by (*metis Nil-not-valid assms hd-append*)

lemma *valid2-hd-Rcons*[*simp*]:
assumes *valid2 tr*
shows *hd* (*tr* ## *tran*) = *hd tr*
by (*metis Nil-not-valid2 assms hd-append*)

lemma *valid2-last-Cons*[*simp*]:
assumes *valid2 tr*
shows *last* (*tran* # *tr*) = *last tr*
by (*metis Nil-not-valid2 assms last.simps*)

lemma *valid2-Cons*:
assumes *valid2 tr* **and** *tgtOf trn* = *srcOf* (*hd tr*) **and** *validTrans trn*
shows *valid2* (*trn* # *tr*)
using *assms* **proof**(*induct arbitrary*: *trn*)

case *Singl* **show** *?case*
unfolding *two-singl-Rcons* **using** *Singl*
by (*intro valid2.Rcons*) (*auto intro*: *valid2.Singl*)
next
case *Rcons* **show** *?case*
unfolding *append.append-Cons*[*symmetric*] **using** *Rcons* **by** (*intro valid2.Rcons*) *auto*
qed

lemma *valid-valid2*: *valid* = *valid2*
proof(*rule ext*, *safe*)
fix *tr* **assume** *valid tr* **thus** *valid2 tr*
by (*induct*) (*auto intro*: *valid2.Singl valid2-Cons*)
next
fix *tr* **assume** *valid2 tr* **thus** *valid tr*
by (*induct*) (*auto intro*: *valid.Singl valid-Rcons*)
qed

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* []
unfolding *validFrom-def* **by** *auto*

lemma *validFrom-valid*[*simp*,*intro*]: *valid tr* ∧ *srcOf* (*hd tr*) = *s* ⟹ *validFrom s tr*
unfolding *validFrom-def* **by** *auto*

lemma *validFrom-validTrans*[*intro*]:
assumes *validTrans* (*Trans s a ou s'*) **and** *validFrom s' tr*
shows *validFrom s* (*Trans s a ou s'* # *tr*)
using *assms* **unfolding** *validFrom-def* **by** *auto*

3.4 Traces versus reachability

lemma *valid-reach-src-tgt*:
assumes *valid tr* **and** *reach* (*srcOf* (*hd tr*))
shows *reach* (*tgtOf* (*last tr*))
using *assms* **by** *induct* (*auto intro*: *reach-PairI simp*: *validTrans*)

lemma *valid-init-reach*:
assumes *valid tr* **and** *srcOf* (*hd tr*) = *istate*
shows *reach* (*tgtOf* (*last tr*))
using *valid-reach-src-tgt assms reach.Istate* **by** *metis*

lemma *Trans-fst-sndI*:
valid [*Trans s a* (*fst* (*step s a*)) (*snd* (*step s a*))]
by (*metis valid.Singl surjective-pairing validTrans.simps*)

lemma *reach-init-valid*:
assumes *reach s*
shows
s = istate
∨
(∃ *tr*. *valid tr* ∧ *srcOf* (*hd tr*) = *istate* ∧ *tgtOf* (*last tr*) = *s*)
using *assms* **proof** *induction*
case (*Step s a*)
thus *?case* **proof**(*elim disjE exE conjE*)
assume *s*: *s = istate*
let *?ou* = *fst* (*step s a*) **let** *?s'* = *snd* (*step s a*)
show *?thesis* **using** *s*
by (*intro disjI2 exI*[*of - [Trans s a ?ou ?s'*]]) *auto*
next
fix *tr* **assume** *v*: *valid tr* **and** *s*: *srcOf* (*hd tr*) = *istate* **and** *t*: *tgtOf* (*last tr*) = *s*
let *?ou* = *fst* (*step s a*) **let** *?s'* = *snd* (*step s a*)
show *?thesis* **using** *v t s*
by (*intro disjI2 exI*[*of - tr ## Trans s a ?ou ?s'*]) (*auto intro*: *valid-Rcons*)
qed
qed *auto*

lemma *reach-validFrom*:
assumes *reach s'*
shows ∃ *s tr*. *s = istate* ∧ (*s = s'* ∨ (*validFrom s tr* ∧ *tgtOf* (*last tr*) = *s'*))
using *reach-init-valid*[*OF assms*] **unfolding** *validFrom-def* **by** *auto*

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* ⇒ *bool*) ⇒ (('*state*,'*act*,'*out*) *trans* ⇒ '*a*) ⇒ ('*state*,'*act*,'*out*) *trace* ⇒ '*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*
by *auto* (*metis list.exhaust*)
termination by *lexicographic-order*

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* ⇒ *'value*
and
γ :: (*'state*,*'act*,*'out*) *trans* => *bool* **and** *g* :: (*'state*,*'act*,*'out*) *trans* ⇒ *'obs*
and
T :: (*'state*,*'act*,*'out*) *trans* ⇒ *bool*
and
B :: *'value list* ⇒ *'value list* ⇒ *bool*
begin

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

definition *O* :: (*'state*,*'act*,*'out*) *trace* ⇒ *'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*
unfolding *V-def* **by** *auto*

lemma *O-simps*[*simp*]:
O [] = [] ¬ γ *trn* ⟹ *O* (*trn* # *tr*) = *O tr* γ *trn* ⟹ *O* (*trn* # *tr*) = *g trn* # *O tr*
unfolding *O-def* **by** *auto*

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'*
by (*metis BD-Security.reachNT.simps assms trans.sel*)

lemma *reachNT-reach*: **assumes** *reachNT s* **shows** *reach s*
using *assms* **by** *induct* (*auto intro*: *reach.intros*, *metis reach.Step snd-conv validTrans*)

lemma *reachNT-stateO-aux*:
assumes *reachNT s*
shows *s* = *istate* ∨ (∃ *sh a ou*. *reach sh* ∧ *step sh a* = (*ou*,*s*) ∧ ¬*T* (*Trans sh a ou s*))
using *assms* **by** *induct* (*clarsimp*, *metis BD-Security.reachNT-reach trans.exhaust-sel*)

lemma *reachNT-state-cases*[*cases set*,*consumes 1*, *case-names init step*]:
assumes *reachNT s*

obtains $s = istate$
| $sh\ a\ ou$ **where** $reach\ sh\ step\ sh\ a = (ou,s)\ \neg T\ (Trans\ sh\ a\ ou\ s)$
by $(metis\ reachNT\text{-}stateO\text{-}aux[OF\ assms])$

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 $invarNT\text{-}disj$:
assumes $invarNT\ Inv1$ **and** $invarNT\ Inv2$
shows $invarNT\ (\lambda\ s.\ Inv1\ s \vee Inv2\ s)$
using $assms$ **unfolding** $invarNT\text{-}def$ **by** $blast$

lemma $invarNT\text{-}conj$:
assumes $invarNT\ Inv1$ **and** $invarNT\ Inv2$
shows $invarNT\ (\lambda\ s.\ Inv1\ s \wedge Inv2\ s)$
using $assms$ **unfolding** $invarNT\text{-}def$ **by** $blast$

lemma $holdsIstate\text{-}invarNT$:
assumes h: $holdsIstate\ Inv$ **and** i: $invarNT\ Inv$ **and** a: $reachNT\ s$
shows $Inv\ s$
using a **using** $h\ i$ **unfolding** $holdsIstate\text{-}def\ invarNT\text{-}def$
by $(induct\ rule{:}\ reachNT.induct)\ (metis\ i\ invarNT\text{-}def\ trans.exhaust\text{-}sel)+$

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

lemma $V\text{-}iff\text{-}non\text{-}\varphi[simp]$: $V\ (trn\ \#\ tr) = V\ tr \longleftrightarrow \neg\ \varphi\ trn$
by $(cases\ \varphi\ trn)\ auto$

lemma $V\text{-}imp\text{-}\varphi$: $V\ (trn\ \#\ tr) = v\ \#\ V\ tr \Longrightarrow \varphi\ trn$
by $(cases\ \varphi\ trn)\ auto$

lemma $V\text{-}imp\text{-}Nil$: $V\ (trn\ \#\ tr) = [] \Longrightarrow V\ tr = []$
by $(metis\ V\text{-}simps\ list.distinct\ trans.exhaust)$

lemma $V\text{-}iff\text{-}Nil[simp]$: $V\ (trn\ \#\ tr) = [] \longleftrightarrow \neg\ \varphi\ trn \wedge V\ tr = []$
by $(metis\ V\text{-}iff\text{-}non\text{-}\varphi\ V\text{-}imp\text{-}Nil)$

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*)
unfolding *consume-def* **by** (*auto split*: *if-splits*)

lemma *ex-consume-φ*:
assumes ¬ φ *trn*
obtains *vl'* **where** *consume trn vl vl'*
using *assms* **unfolding** *consume-def* **by** *auto*

lemma *ex-consume-NO*:
assumes *vl* ≠ [] **and** *f trn* = *hd vl*
obtains *vl'* **where** *consume trn vl vl'*
using *assms* **unfolding** *consume-def* **by** (*cases* φ *trn*) *auto*

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*
unfolding *iaction-def* **using** *assms* **by** *auto*

definition *match* **where**
match Δ *s s1 vl1 a ou s' vl'* ≡
∃ *a1 ou1 s1' vl1'*.
let trn = *Trans s a ou s'*; *trn1* = *Trans s1 a1 ou1 s1' in*
validTrans trn1 ∧
consume trn1 vl1 vl1' ∧
γ *trn* = γ *trn1* ∧ (γ *trn* ⟶ *g trn* = *g trn1*) ∧
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'*
unfolding *match-def* **using** *assms* **by** *auto*

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'*
unfolding *ignore-def* **using** *assms* **by** *auto*

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*
using *assms* **unfolding** *reaction-def* **by** *auto*

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*⟧
$\Longrightarrow$ (φ *trn* $\longrightarrow$ *f trn* $\neq$ *v*) $\wedge$ *K* (*tgtOf trn*)
shows *exit s v*
using *K* **unfolding** *exit-def* **proof**(*intro allI conjI impI*)
fix *tr trn* **assume** *K s* **and** *validFrom s* (*tr ##* *trn*) $\wedge$ *never T* (*tr ##* *trn*) $\wedge$ φ *trn*
thus *f trn* $\neq$ *v*

using *I* **unfolding** *validFrom-def* **by** (*induction tr arbitrary*: *s trn*)
(*auto*, *metis neq-Nil-conv rotate1.simps*(*2*) *rotate1-is-Nil-conv valid-ConsE*)
qed

definition *noVal* **where**
noVal K v ≡
∀ *s a ou s′*. *reachNT s* ∧ *K s* ∧ *step s a* = (*ou*,*s′*) ∧ φ (*Trans s a ou s′*) ⟶ *f* (*Trans s a ou s′*) ≠ *v*

lemma *noVal-disj*:
assumes *noVal Inv1 v* **and** *noVal Inv2 v*
shows *noVal* (λ *s*. *Inv1 s* ∨ *Inv2 s*) *v*
using *assms* **unfolding** *noVal-def* **by** *metis*

lemma *noVal-conj*:
assumes *noVal Inv1 v* **and** *noVal Inv2 v*
shows *noVal* (λ *s*. *Inv1 s* ∧ *Inv2 s*) *v*
using *assms* **unfolding** *noVal-def* **by** *blast*

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*
unfolding *noφ-def noVal-def* **by** *auto*

lemma *exitI*[*consumes 2*, *induct pred*: *exit*]:
assumes *rs*: *reachNT s* **and** *K*: *K s*
and *I*:
⋀ *s a ou s′*.
⟦*reach s*; *reachNT s*; *step s a* = (*ou*,*s′*); *K s*⟧
⟹ (φ (*Trans s a ou s′*) ⟶ *f* (*Trans s a ou s′*) ≠ *v*) ∧ *K s′*
shows *exit s v*
proof−
let *?K* = λ *s*. *reachNT s* ∧ *K s*
show *?thesis* **using** *assms* **by** (*intro exit-coind*[*of ?K*])
(*metis BD-Security.reachNT-reach IO-Automaton.validTrans reachNT.Step trans.exhaust-sel*)+
qed

lemma *exitI2*:
assumes *rs*: *reachNT s* **and** *K*: *K s*
and *invarNT K* **and** *noVal K v*
shows *exit s v*
proof−
let *?K* = λ *s*. *reachNT s* ∧ *K s*
show *?thesis* **using** *assms* **unfolding** *invarNT-def noVal-def* **apply**(*intro exit-coind*[*of ?K*])
by *metis* (*metis IO-Automaton.validTrans reachNT.Step trans.exhaust-sel*)
qed

definition *noVal2* **where**
noVal2 K v ≡
∀ *s a ou s′*. *reachNT s* ∧ *K s v* ∧ *step s a* = (*ou*,*s′*) ∧ φ (*Trans s a ou s′*) ⟶ *f* (*Trans s a ou s′*) ≠ *v*

lemma *noVal2-disj*:
assumes *noVal2 Inv1 v* **and** *noVal2 Inv2 v*
shows *noVal2* (λ *s v*. *Inv1 s v* ∨ *Inv2 s v*) *v*
using *assms* **unfolding** *noVal2-def* **by** *metis*

lemma *noVal2-conj*:
assumes *noVal2 Inv1 v* **and** *noVal2 Inv2 v*
shows *noVal2* (λ *s v*. *Inv1 s v* ∧ *Inv2 s v*) *v*
using *assms* **unfolding** *noVal2-def* **by** *blast*

lemma *noVal-noVal2*: *noVal K v* ⟹ *noVal2* (λ *s v*. *K s*) *v*
unfolding *noVal-def noVal2-def* **by** *auto*

lemma *exitI-noVal2*[*consumes 2*, *induct pred*: *exit*]:
assumes *rs*: *reachNT s* **and** *K*: *K s v*
and *I*:
⋀ *s a ou s′*.
⟦*reach s*; *reachNT s*; *step s a* = (*ou*,*s′*); *K s v*⟧
⟹ (φ (*Trans s a ou s′*) ⟶ *f* (*Trans s a ou s′*) ≠ *v*) ∧ *K s′ v*
shows *exit s v*
proof−
let *?K* = λ *s*. *reachNT s* ∧ *K s v*
show *?thesis* **using** *assms* **by** (*intro exit-coind*[*of ?K*])
(*metis BD-Security.reachNT-reach IO-Automaton.validTrans reachNT.Step trans.exhaust-sel*)+
qed

lemma *exitI2-noVal2*:
assumes *rs*: *reachNT s* **and** *K*: *K s v*
and *invarNT* (λ *s*. *K s v*) **and** *noVal2 K v*
shows *exit s v*
proof−
let *?K* = λ *s*. *reachNT s* ∧ *K s v*
show *?thesis* **using** *assms* **unfolding** *invarNT-def noVal2-def*
by (*intro exit-coind*[*of ?K*]) (*metis IO-Automaton.validTrans reachNT.Step trans.exhaust-sel*)+
qed

lemma *exit-validFrom*:
assumes *vl*: *vl* ≠ [] **and** *i*: *exit s* (*hd vl*) **and** *v*: *validFrom s tr* **and** *V*: *V tr* = *vl*
and *T*: *never T tr*
shows *False*
using *i v V T* **proof**(*induction tr arbitrary*: *s*)

```
  case Nil thus ?case by (metis V-simps(1) vl)
next
  case (Cons trn tr s)
  show ?case
  proof(cases φ trn)
    case True
    hence f trn = hd vl using Cons by (metis V-simps(3) hd-Cons-tl list.inject vl)
    moreover have validFrom s [trn] using ‹validFrom s (trn # tr)›
    unfolding validFrom-def by auto
    ultimately show ?thesis using Cons True unfolding exit-def
    by (elim allE[of - []]) auto
  next
    case False
    hence V tr = vl using Cons by auto
    moreover have never T tr by (metis Cons.prems list-all-simps)
    moreover from ‹validFrom s (trn # tr)› have validFrom (tgtOf trn) tr and s: s = srcOf trn
    by (metis list.distinct(1) validFrom-def valid-ConsE Cons.prems(2)
            IO-Automaton.validFrom-def list.discI list.sel(1))+
    moreover have exit (tgtOf trn) (hd vl) using ‹exit s (hd vl)›
    unfolding exit-def s by simp
    (metis (no-types) Cons.prems(2) Cons.prems(4) append-Cons list.sel(1)
          list.distinct list-all-simps valid.Cons validFrom-def valid-ConsE)
    ultimately show ?thesis using Cons(1) by auto
  qed
qed

definition unwind where
unwind Δ ≡
 ∀ 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)

lemma unwindI[intro?]:
assumes
⋀ 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
using assms unfolding unwind-def by auto
```

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−
let *?S* = λ *tr vl1*.
∀ *s vl s1*. *reachNT s* ∧ *reach s1* ∧ Δ *s vl s1 vl1* ∧ *validFrom s tr* ∧ *never T tr* ∧ *V tr* = *vl* ⟶
(∃ *tr1*. *validFrom s1 tr1* ∧ *O tr1* = *O tr* ∧ *V tr1* = *vl1*)
let *?f* = λ *tr vl1*. *length tr* + *length vl1*
have *?S tr vl1*
proof(*induct rule*: *measure-induct2*[*of ?f ?S*])
case (*1 tr vl1*)
show *?case*
proof(*intro allI impI*, *elim conjE*)
fix *s vl s1* **assume** *rs*: *reachNT s* **and** *rs1*: *reach s1* **and** Δ: Δ *s vl s1 vl1*
and *v*: *validFrom s tr* **and** *NT*: *never T tr* **and** *V*: *V tr* = *vl*
hence (*vl* ≠ [] ∧ *exit s* (*hd vl*)) ∨
iaction Δ *s vl s1 vl1* ∨
(*reaction* Δ *s vl s1 vl1* ∧ ¬ *iaction* Δ *s vl s1 vl1*)
(**is** *?exit* ∨ *?iact* ∨ *?react* ∧ -)
using *unwind* **unfolding** *unwind-def* **by** *metis*
thus ∃ *tr1*. *validFrom s1 tr1* ∧ *O tr1* = *O tr* ∧ *V tr1* = *vl1*
proof *safe*
assume *vl* ≠ [] **and** *exit s* (*hd vl*)
hence *False* **using** *v V exit-validFrom NT* **by** *auto*
thus *?thesis* **by** *auto*
next
assume *?iact*
thus *?thesis* **unfolding** *iaction-def Let-def* **proof** *safe*
fix *a1* :: *'act* **and** *ou1* :: *'out* **and** *s1'* :: *'state* **and** *vl1'*
let *?trn1* = *Trans s1 a1 ou1 s1'*
assume *vtrans1*: *validTrans ?trn1* **and** φ*1*: φ *?trn1*
and *c*: *consume ?trn1 vl1 vl1'* **and** γ: ¬ γ *?trn1* **and** Δ: Δ *s vl s1' vl1'*
from φ*1 c* **obtain** *v1* **where** *vl1*: *vl1* = *v1* # *vl1'* **and** *f1*: *f ?trn1* = *v1*
unfolding *consume-def* **by** (*cases vl1*) *auto*
have *rs1'*: *reach s1'* **using** *rs1 vtrans1* **by** (*auto intro*: *reach-PairI*)
obtain *tr1* **where** *v1*: *validFrom s1' tr1* **and** *O*: *O tr1* = *O tr* **and** *V*: *V tr1* = *vl1'*
using *1*[*of tr vl1'*] *rs rs1'* Δ *v NT V* **unfolding** *vl1* **by** *auto*
show *?thesis*
using *vtrans1 v1 O* γ *V* φ*1 f1* **unfolding** *vl1* **by** (*intro exI*[*of - ?trn1* # *tr1*]) *auto*
qed
next
assume *react*: *?react* **and** *iact*: ¬ *?iact*
show *?thesis*
proof(*cases tr*)
case *Nil* **note** *tr* = *Nil*
hence *vl*: *vl* = [] **using** *V* **by** *simp*
show *?thesis* **proof**(*cases vl1*)

```
    case Nil note vl1 = Nil
    show ?thesis using 1[of tr vl1] Δ V NT V unfolding tr vl1 by auto
  next
    case Cons
    hence False using vl unwind rs rs1 Δ iact unfolding unwind-def by auto
    thus ?thesis by auto
  qed
next
  case (Cons trn tr') note tr = Cons
  show ?thesis
  proof(cases trn)
    case (Trans ss a ou s') note trn = Trans let ?trn = Trans s a ou s'
    have ss: ss = s using trn v unfolding tr validFrom-def by auto
    have Ta: ¬ T ?trn and s: s = srcOf trn and vtrans: validTrans ?trn
    and v': validFrom s' tr' and NT': never T tr'
    using v NT V unfolding tr validFrom-def trn by auto
    have rs': reachNT s' using rs vtrans Ta by (auto intro: reachNT-PairI)
    {assume φ ?trn hence vl ≠ [] ∧ f ?trn = hd vl using V unfolding tr trn ss by auto
    }
    then obtain vl' where c: consume ?trn vl vl'
    using ex-consume-φ ex-consume-NO by metis
    have V': V tr' = vl' using V c unfolding tr trn ss consume-def
    by (cases φ ?trn) (simp-all, metis list.sel(2−3))
    have match Δ s s1 vl1 a ou s' vl' ∨ ignore Δ s s1 vl1 a ou s' vl' (is ?match ∨ ?ignore)
    using react unfolding reaction-def using vtrans Ta c by auto
    thus ?thesis proof safe
      assume ?match
      thus ?thesis unfolding match-def Let-def proof (elim exE conjE)
        fix a1 :: 'act and ou1 :: 'out and s1' :: 'state and vl1'
        let ?trn1 = Trans s1 a1 ou1 s1'
        assume Δ: Δ s' vl' s1' vl1' and vtrans1: validTrans ?trn1
        and c1: consume ?trn1 vl1 vl1' and γ: γ ?trn = γ ?trn1
        and g: γ ?trn ⟶ g ?trn = g ?trn1
        have rs1': reach s1' using rs rs1 vtrans vtrans1 by (auto intro: reach-PairI)
        obtain tr1 where v1: validFrom s1' tr1 and O: O tr1 = O tr' and V: V tr1 = vl1'
        using 1[of tr' vl1'] rs' rs1' Δ v' NT' V' c1 unfolding tr by auto
        have V (?trn1 # tr1) = vl1
        using c1 V unfolding consume-def by (cases φ ?trn1) auto
        thus ?thesis
        apply(intro exI[of - Trans s1 a1 ou1 s1' # tr1])
        using vtrans1 v1 O γ g V unfolding tr trn ss by auto
      qed
    next
      assume ?ignore
      thus ?thesis unfolding ignore-def Let-def proof (elim exE conjE)
        assume γ: ¬ γ ?trn and Δ: Δ s' vl' s1 vl1
        obtain tr1 where v1: validFrom s1 tr1 and O: O tr1 = O tr' and V: V tr1 = vl1
        using 1[of tr' vl1] rs' rs1 Δ v' NT' V' c unfolding tr by auto
        show ?thesis
```

apply(*intro exI*[*of - tr1*])
using *v1 O V* γ **unfolding** *tr trn ss* **by** *auto*
qed
qed
qed
qed
qed
qed
qed
thus *?thesis* **using** *assms* **by** *auto*
qed

theorem *unwind-secure*:
assumes *init*: $\bigwedge$ *vl vl1. B vl vl1* $\Longrightarrow$ Δ *istate vl istate vl1*
and *unwind*: *unwind* Δ
shows *secure*
using *assms unwind-trace* **unfolding** *secure-def* **by** (*blast intro*: *reach.Istate reachNT.Istate*)

5 Compositional Reasoning

context *BD-Security* **begin**

5.1 Preliminaries

definition *disjAll* Δs *s vl s1 vl1* $\equiv$ ($\exists \Delta \in \Delta s$. Δ *s vl s1 vl1*)

lemma *disjAll-simps*[*simp*]:
disjAll {} $\equiv$ λ- - - -. *False*
disjAll (*insert* Δ Δs) $\equiv$ λs *vl s1 vl1*. Δ *s vl s1 vl1* $\vee$ *disjAll* Δs *s vl s1 vl1*
unfolding *disjAll-def*[*abs-def*] **by** *auto*

lemma *iaction-mono*:
assumes *1*: *iaction* Δ *s vl s1 vl1* **and** *2*: $\bigwedge$ *s vl s1 vl1*. Δ *s vl s1 vl1* $\Longrightarrow$ Δ' *s vl s1 vl1*
shows *iaction* Δ' *s vl s1 vl1*
proof−
obtain *a1 ou1 s1' vl1'*
where *step s1 a1* = (*ou1*, *s1'*) **and** φ (*Trans s1 a1 ou1 s1'*)
and *consume* (*Trans s1 a1 ou1 s1'*) *vl1 vl1'* **and** $\neg$ γ (*Trans s1 a1 ou1 s1'*)
and Δ *s vl s1' vl1'* **using** *1* **unfolding** *iaction-def* **by** *auto*
thus *?thesis* **unfolding** *iaction-def* **using** *2* **apply** −
by (*rule exI*[*of - a1*], *rule exI*[*of - ou1*], *rule exI*[*of - s1'*], *rule exI*[*of - vl1'*]) *auto*
qed

lemma *match-mono*:

assumes *1*: *match* 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 *match* Δ′ *s s1 vl1 a ou s′ vl′*
proof−
obtain *a1 ou1 s1′ vl1′*
where Δ *s′ vl′ s1′ vl1′*
and *step 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′*) ⟶ *g* (*Trans s a ou s′*) = *g* (*Trans s1 a1 ou1 s1′*))
using *1* **unfolding** *match-def* **by** *auto*
thus *?thesis* **unfolding** *match-def* **using** *2* **apply** −
by (*rule exI*[*of - a1*], *rule exI*[*of - ou1*], *rule exI*[*of - s1′*], *rule exI*[*of - vl1′*]) *auto*
qed

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′*
using *assms* **unfolding** *ignore-def* **by** *auto*

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
fix *a ou s′ vl′*
assume *step s a* = (*ou*, *s′*) **and** ¬ *T* (*Trans s a ou s′*) **and** *consume* (*Trans s a ou s′*) *vl vl′*
hence *match* Δ *s s1 vl1 a ou s′ vl′* ∨ *ignore* Δ *s s1 vl1 a ou s′ vl′* (**is** *?m* ∨ *?i*)
using *1* **unfolding** *reaction-def* **by** *auto*
thus *match* Δ′ *s s1 vl1 a ou s′ vl′* ∨ *ignore* Δ′ *s s1 vl1 a ou s′ vl′* (**is** *?m′* ∨ *?i′*)
proof
assume *?m* **from** *match-mono*[*OF this 2*] **show** *?thesis* **by** *simp*
next
assume *?i* **from** *ignore-mono*[*OF this 2*] **show** *?thesis* **by** *simp*
qed
qed

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
$\bigwedge$ *s vl s1 vl1.*
$[\![$*reachNT s*; *reach s1*; Δ *s vl s1 vl1*$]\!]$
$\Longrightarrow$
vl $\neq$ [] $\wedge$ *exit s* (*hd vl*)
$\vee$
iaction (*disjAll* Δ*s*) *s vl s1 vl1*
$\vee$
(*vl* $\neq$ [] $\vee$ *vl1* = []) $\wedge$ *reaction* (*disjAll* Δ*s*) *s vl s1 vl1*
shows *unwind-to* Δ Δ*s*
using *assms* **unfolding** *unwind-to-def* **by** *auto*

lemma *unwind-dec*:
assumes *ne*: $\bigwedge$ Δ. $\Delta \in \Delta s \Longrightarrow$ *next* $\Delta \subseteq \Delta s \wedge$ *unwind-to* Δ (*next* Δ)
shows *unwind* (*disjAll* Δ*s*) (**is** *unwind ?*Δ)
proof
fix *s s1* :: *'state* **and** *vl vl1* :: *'value list*
assume *r*: *reachNT s reach s1* **and** Δ: *?*Δ *s vl s1 vl1*
then obtain Δ **where** Δ: $\Delta \in \Delta s$ **and** *2*: Δ *s vl s1 vl1* **unfolding** *disjAll-def* **by** *auto*
let *?*$\Delta s'$ = *next* Δ **let** *?*Δ' = *disjAll ?*$\Delta s'$
have (*vl* $\neq$ [] $\wedge$ *exit s* (*hd vl*)) $\vee$
iaction ?Δ' *s vl s1 vl1* $\vee$
((*vl* $\neq$ [] $\vee$ *vl1* = []) $\wedge$ *reaction ?*Δ' *s vl s1 vl1*)
using *2* Δ *ne r* **unfolding** *unwind-to-def* **by** *auto*
moreover have $\bigwedge$ *s vl s1 vl1. ?*Δ' *s vl s1 vl1* $\Longrightarrow$ *?*Δ *s vl s1 vl1*
using *ne*[*OF* Δ] **unfolding** *disjAll-def* **by** *auto*
ultimately show
(*vl* $\neq$ [] $\wedge$ *exit s* (*hd vl*)) $\vee$
iaction ?Δ *s vl s1 vl1* $\vee$
((*vl* $\neq$ [] $\vee$ *vl1* = []) $\wedge$ *reaction ?*Δ *s vl s1 vl1*)
using *iaction-mono*[*of ?*Δ' - - - - *?*Δ] *reaction-mono*[*of ?*Δ' - - - - *?*Δ] **by** *blast*
qed

lemma *init-dec*:
assumes Δ*0*: Δ*0* $\in \Delta s$
and *i*: $\bigwedge$ *vl vl1. B vl vl1* $\Longrightarrow$ Δ*0 istate vl istate vl1*
shows $\forall$ *vl vl1. B vl vl1* $\longrightarrow$ *disjAll* Δ*s istate vl istate vl1*
using *assms* **unfolding** *disjAll-def* **by** *auto*

theorem *unwind-dec-secure*:
assumes Δ*0*: Δ*0* $\in \Delta s$
and *i*: $\bigwedge$ *vl vl1. B vl vl1* $\Longrightarrow$ Δ*0 istate vl istate vl1*
and *ne*: $\bigwedge$ Δ. $\Delta \in \Delta s \Longrightarrow$ *next* $\Delta \subseteq \Delta s \wedge$ *unwind-to* Δ (*next* Δ)
shows *secure*
using *init-dec*[*OF* Δ*0 i*] *unwind-dec*[*OF ne*] *unwind-secure* **by** *metis*

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
using *assms* **unfolding** *unwind-cont-def* **by** *auto*

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
using *assms* **unfolding** *unwind-exit-def* **by** *auto*

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 $\exists$ $\Delta 2$. (Δ,$\Delta 2$) $\in$ *allConsec* Δs
using *assms* **proof** (*induction* $\Delta s'$ *arbitrary*: Δs)
case *Nil* **thus** *?case* **by** *auto*
next

```
  case (Cons Δ3 Δs' Δs)
  show ?case proof(cases Δ = Δ3)
    case True
    show ?thesis proof(cases Δs')
      case Nil
      show ?thesis unfolding ‹Δs = (Δ3 # Δs') ## Δ1› Nil True by (rule exI[of - Δ1]) simp
    next
      case (Cons Δ4 Δs'')
      show ?thesis unfolding ‹Δs = (Δ3 # Δs') ## Δ1› Cons True by (rule exI[of - Δ4]) simp
    qed
  next
    case False hence Δ ∈ set Δs' using Cons by auto
    then obtain Δ2 where (Δ, Δ2) ∈ allConsec (Δs' ## Δ1) using Cons by auto
    thus ?thesis unfolding ‹Δs = (Δ3 # Δs') ## Δ1› by (intro exI[of - Δ2]) (cases Δs', auto)
  qed
qed

lemma allConsec-set:
assumes (Δ1,Δ2) ∈ allConsec Δs
shows Δ1 ∈ set Δs ∧ Δ2 ∈ set Δs
using assms by (induct Δs rule: allConsec.induct) auto


theorem unwind-decomp-secure:
assumes n: Δs ≠ []
and i: ⋀ vl vl1. B vl vl1 ⟹ hd Δs istate vl istate vl1
and c: ⋀ Δ1 Δ2. (Δ1,Δ2) ∈ allConsec Δs ⟹ unwind-cont Δ1 {Δ1, Δ2, Δe}
and l: unwind-cont (last Δs) {last Δs, Δe}
and e: unwind-exit Δe
shows secure
proof–
  let ?Δ0 = hd Δs  let ?Δs = insert Δe (set Δs)
  define next where next Δ1 =
    (if Δ1 = Δe then {}
     else if Δ1 = last Δs then {Δ1,Δe}
     else {Δ1,SOME Δ2. (Δ1,Δ2) ∈ allConsec Δs,Δe}) for Δ1
  show ?thesis
  proof(rule unwind-dec-secure)
    show ?Δ0 ∈ ?Δs using n by auto
  next
    fix vl vl1 assume B vl vl1
    thus ?Δ0 istate vl istate vl1 by fact
  next
    fix Δ
    assume 1: Δ ∈ ?Δs show next Δ ⊆ ?Δs ∧ unwind-to Δ (next Δ)
    proof–
      {assume Δ = Δe
       hence ?thesis using e unfolding next-def unwind-exit-def unwind-to-def by auto
      }
```

moreover
{assume $\Delta = last\ \Delta s$ **and** $\Delta \neq \Delta e$
hence *?thesis* **using** *n l* **unfolding** *next-def unwind-cont-def unwind-to-def* **by** *simp*
}
moreover
{assume *1*: $\Delta \in set\ \Delta s$ **and** *2*: $\Delta \neq last\ \Delta s\ \Delta \neq \Delta e$
then obtain $\Delta' \Delta s'$ **where** Δs: $\Delta s = \Delta s' \#\# \Delta'$ **and** Δ: $\Delta \in set\ \Delta s'$
by (*metis* (*no-types*) *append-Cons append-assoc in-set-conv-decomp last-snoc rev-exhaust*)
have $\exists\ \Delta 2.\ (\Delta, \Delta 2) \in allConsec\ \Delta s$ **using** *set-allConsec*[*OF* $\Delta\ \Delta s$] **.**
hence $(\Delta, SOME\ \Delta 2.\ (\Delta, \Delta 2) \in allConsec\ \Delta s) \in allConsec\ \Delta s$ **by** (*metis* (*lifting*) *someI-ex*)
hence *?thesis* **using** *1 2 c* **unfolding** *next-def unwind-cont-def unwind-to-def*
by *simp* (*metis* (*no-types*) *allConsec-set*)
}
ultimately show *?thesis* **using** *1* **by** *blast*
qed
qed
qed

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*
apply(*rule unwind-decomp-secure*[*of* $[\Delta 1, \Delta 2, \Delta 3]\ \Delta e$])
using *assms* **by** *auto*

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*
apply(*rule unwind-decomp-secure*[*of* $[\Delta 1, \Delta 2, \Delta 3, \Delta 4]\ \Delta e$])
using *assms* **by** *auto*

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*
apply(*rule unwind-decomp-secure*[*of* [$\Delta 1$, $\Delta 2$, $\Delta 3$, $\Delta 4$, $\Delta 5$] Δe])
using *assms* **by** *auto*