

Bounded-Deducibility Security

Andrei Popescu Peter Lammich

Contents

1	Introduction	1
2	Trivia	2
3	IO Automaton	2
3.1	Preliminaries	2
3.2	Reachability and invariance	2
3.3	System traces	3
3.4	Traces versus reachability	5
4	BD Security	5
4.1	Definition	5
4.2	Unwinding proof method	8
5	Compositional Reasoning	12
5.1	Preliminaries	12
5.2	Decomposition into an arbitrary network of components	13
5.3	A customization for linear modular reasoning	14
5.4	Instances	15

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 \Rightarrow 'b \Rightarrow nat
assumes $\bigwedge x1\ x2. (\bigwedge y1\ y2. \text{meas } y1\ y2 < \text{meas } x1\ x2 \Longrightarrow S\ y1\ y2) \Longrightarrow S\ x1\ x2$
shows $S\ x1\ x2$
<proof>

Right cons:

abbreviation *Rcons* (**infix** ## 70) **where** $xs\ \#\# \ x \equiv xs\ @\ [x]$

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 \Rightarrow 'act \Rightarrow 'out * 'state
begin

3.2 Reachability and invariance

inductive *reach* :: 'state \Rightarrow 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'*
<proof>

definition $holdsIstate :: ('state \Rightarrow bool) \Rightarrow bool$ **where**
 $holdsIstate \varphi \equiv \varphi \text{ ivate}$

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

lemma $holdsIstate$ - $invar$:
assumes h : $holdsIstate\ \varphi$ **and** i : $invar\ \varphi$ **and** a : $reach\ s$
shows $\varphi\ s$
 $\langle proof \rangle$

3.3 System traces

definition $out :: 'state \Rightarrow 'act \Rightarrow 'out$ **where** $out\ s\ a \equiv fst\ (step\ s\ a)$
definition $eff :: 'state \Rightarrow 'act \Rightarrow 'state$ **where** $eff\ s\ a \equiv snd\ (step\ s\ a)$

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

lemma $validTrans$:
 $validTrans\ trn =$
 $(step\ (srcOf\ trn)\ (actOf\ trn) = (outOf\ trn, tgtOf\ trn))$
 $\langle proof \rangle$

inductive $valid :: ('state, 'act, 'out) trace \Rightarrow bool$ **where**
 $Singl[simp, intro!]$:
 $validTrans\ trn$
 \implies
 $valid\ [trn]$
 $|$
 $Cons[intro]$:
 $\llbracket validTrans\ trn; tgtOf\ trn = srcOf\ (hd\ tr); valid\ tr \rrbracket$
 \implies
 $valid\ (trn\ \# tr)$

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

inductive $valid2 :: ('state, 'act, 'out) trace \Rightarrow bool$ **where**
 $Singl[simp, intro!]$:
 $validTrans\ trn$
 \implies
 $valid2\ [trn]$
 $|$

Rcons[*intro*]:
 $\llbracket \text{valid2 } tr; \text{tgtOf } (last \ tr) = \text{srcOf } trn; \text{validTrans } trn \rrbracket$
 \implies
 $\text{valid2 } (tr \ \#\# \ trn)$

inductive-cases *valid2-SingleE*[*elim!*]: $\text{valid2 } [trn]$
inductive-cases *valid2-RconsE*[*elim*]: $\text{valid2 } (tr \ \#\# \ trn)$

lemma *Nil-not-valid*[*simp*]: $\neg \text{valid } []$
 $\langle \text{proof} \rangle$

lemma *Nil-not-valid2*[*simp*]: $\neg \text{valid2 } []$
 $\langle \text{proof} \rangle$

lemma *valid-Rcons*:
assumes *valid* *tr* **and** $\text{tgtOf } (last \ tr) = \text{srcOf } trn$ **and** *validTrans* *trn*
shows $\text{valid } (tr \ \#\# \ trn)$
 $\langle \text{proof} \rangle$

lemma *valid-hd-Rcons*[*simp*]:
assumes *valid* *tr*
shows $\text{hd } (tr \ \#\# \ tran) = \text{hd } tr$
 $\langle \text{proof} \rangle$

lemma *valid2-hd-Rcons*[*simp*]:
assumes *valid2* *tr*
shows $\text{hd } (tr \ \#\# \ tran) = \text{hd } tr$
 $\langle \text{proof} \rangle$

lemma *valid2-last-Cons*[*simp*]:
assumes *valid2* *tr*
shows $\text{last } (tran \ \# \ tr) = \text{last } tr$
 $\langle \text{proof} \rangle$

lemma *valid2-Cons*:
assumes *valid2* *tr* **and** $\text{tgtOf } trn = \text{srcOf } (hd \ tr)$ **and** *validTrans* *trn*
shows $\text{valid2 } (trn \ \# \ tr)$
 $\langle \text{proof} \rangle$

lemma *valid-valid2*: $\text{valid} = \text{valid2}$
 $\langle \text{proof} \rangle$

lemmas $\text{valid2-valid} = \text{valid-valid2}$ [*symmetric*]

definition *validFrom* :: $'state \Rightarrow ('state, 'act, 'out) \text{trace} \Rightarrow \text{bool}$ **where**
 $\text{validFrom } s \ tr \equiv tr = [] \vee (\text{valid } tr \wedge \text{srcOf } (hd \ tr) = s)$

lemma *validFrom-Nil*[*simp,intro!*]: $\text{validFrom } s \ []$
 $\langle \text{proof} \rangle$

lemma *validFrom-valid*[simp,intro]: $valid\ tr \wedge srcOf\ (hd\ tr) = s \implies validFrom\ s\ tr$
 <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
 $s = istate$
 \vee
 $(\exists\ tr.\ valid\ tr \wedge srcOf\ (hd\ tr) = istate \wedge tgtOf\ (last\ tr) = s)$
 <proof>

lemma *reach-validFrom*:
assumes *reach s'*
shows $\exists\ s\ tr.\ s = istate \wedge (s = s' \vee (validFrom\ s\ tr \wedge tgtOf\ (last\ tr) = s'))$
 <proof>

4 BD Security

4.1 Definition

declare *Let-def*[simp]

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

abbreviation *never* :: $('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow bool$ **where** *never U* $\equiv list-all\ (\lambda\ a.\ \neg\ U\ a)$

function *filtermap* ::

$((\text{'state}, \text{'act}, \text{'out}) \text{ trans} \Rightarrow \text{bool}) \Rightarrow ((\text{'state}, \text{'act}, \text{'out}) \text{ trans} \Rightarrow \text{'a}) \Rightarrow (\text{'state}, \text{'act}, \text{'out}) \text{ trace} \Rightarrow \text{'a list}$
where

$\text{filtermap pred func []} = []$

$\neg \text{pred trn} \Longrightarrow \text{filtermap pred func (trn \# tr)} = \text{filtermap pred func tr}$

$\text{pred trn} \Longrightarrow \text{filtermap pred func (trn \# tr)} = \text{func trn \# filtermap pred func tr}$

$\langle \text{proof} \rangle$

termination $\langle \text{proof} \rangle$

locale *BD-Security = IO-Automaton* *istate step*

for *istate* :: 'state **and** *step* :: 'state \Rightarrow 'act \Rightarrow 'out \times 'state

+

fixes

$\varphi :: (\text{'state}, \text{'act}, \text{'out}) \text{ trans} \Rightarrow \text{bool}$ **and** $f :: (\text{'state}, \text{'act}, \text{'out}) \text{ trans} \Rightarrow \text{'value}$

and

$\gamma :: (\text{'state}, \text{'act}, \text{'out}) \text{ trans} \Rightarrow \text{bool}$ **and** $g :: (\text{'state}, \text{'act}, \text{'out}) \text{ trans} \Rightarrow \text{'obs}$

and

$T :: (\text{'state}, \text{'act}, \text{'out}) \text{ trans} \Rightarrow \text{bool}$

and

$B :: \text{'value list} \Rightarrow \text{'value list} \Rightarrow \text{bool}$

begin

definition $V :: (\text{'state}, \text{'act}, \text{'out}) \text{ trace} \Rightarrow \text{'value list}$ **where** $V \equiv \text{filtermap } \varphi f$

definition $O :: (\text{'state}, \text{'act}, \text{'out}) \text{ trace} \Rightarrow \text{'obs list}$ **where** $O \equiv \text{filtermap } \gamma g$

lemma *V-simps[simp]*:

$V [] = [] \quad \neg \varphi \text{ trn} \Longrightarrow V (\text{trn} \# \text{tr}) = V \text{tr} \quad \varphi \text{ trn} \Longrightarrow V (\text{trn} \# \text{tr}) = f \text{trn} \# V \text{tr}$
 $\langle \text{proof} \rangle$

lemma *O-simps[simp]*:

$O [] = [] \quad \neg \gamma \text{ trn} \Longrightarrow O (\text{trn} \# \text{tr}) = O \text{tr} \quad \gamma \text{ trn} \Longrightarrow O (\text{trn} \# \text{tr}) = g \text{trn} \# O \text{tr}$
 $\langle \text{proof} \rangle$

inductive *reachNT*:: 'state \Rightarrow bool **where**

Istate: *reachNT istate*

|

Step:

$\llbracket \text{reachNT} (\text{srcOf trn}); \text{step} (\text{srcOf trn}) (\text{actOf trn}) = (\text{outOf trn}, \text{tgtOf trn}); \neg T \text{trn} \rrbracket$
 $\Longrightarrow \text{reachNT} (\text{tgtOf trn})$

lemma *reachNT-PairI*:

assumes *reachNT s* **and** *step s a = (ou, s')* **and** $\neg T (\text{Trans } s a \text{ ou } s')$

shows *reachNT s'*

$\langle \text{proof} \rangle$

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

lemma *reachNT-stateO-aux*:

assumes *reachNT s*

shows $s = \text{istate} \vee (\exists sh\ a\ ou. \text{reach}\ sh \wedge \text{step}\ sh\ a = (ou, s) \wedge \neg T (\text{Trans}\ sh\ a\ ou\ s))$

⟨*proof*⟩

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

assumes *reachNT s*

obtains $s = \text{istate}$

| $sh\ a\ ou$ **where** $\text{reach}\ sh\ \text{step}\ sh\ a = (ou, s) \wedge \neg T (\text{Trans}\ sh\ a\ ou\ s)$

⟨*proof*⟩

definition *invarNT where*

$\text{invarNT}\ Inv \equiv \forall s\ a\ ou\ s'. \text{reachNT}\ s \wedge Inv\ s \wedge \neg T (\text{Trans}\ s\ a\ ou\ s') \wedge \text{step}\ s\ a = (ou, s') \longrightarrow Inv\ s'$

lemma *invarNT-disj*:

assumes *invarNT Inv1 and invarNT Inv2*

shows $\text{invarNT} (\lambda s. Inv1\ s \vee Inv2\ s)$

⟨*proof*⟩

lemma *invarNT-conj*:

assumes *invarNT Inv1 and invarNT Inv2*

shows $\text{invarNT} (\lambda s. Inv1\ s \wedge Inv2\ s)$

⟨*proof*⟩

lemma *holdsIstate-invarNT*:

assumes $h: \text{holdsIstate}\ Inv$ **and** $i: \text{invarNT}\ Inv$ **and** $a: \text{reachNT}\ s$

shows $Inv\ s$

⟨*proof*⟩

definition *secure :: bool where*

$\text{secure} \equiv$

$\forall tr\ vl\ vl1.$

$\text{validFrom}\ \text{istate}\ tr \wedge \text{never}\ T\ tr \wedge B\ vl\ vl1 \wedge V\ tr = vl \longrightarrow$

$(\exists tr1. \text{validFrom}\ \text{istate}\ tr1 \wedge O\ tr1 = O\ tr \wedge V\ tr1 = vl1)$

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

⟨*proof*⟩

lemma *V-imp-φ*: $V (trn \# tr) = v \# V tr \implies \varphi\ trn$

⟨*proof*⟩

lemma *V-imp-Nil*: $V (trn \# tr) = [] \implies V tr = []$

⟨*proof*⟩

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

end

4.2 Unwinding proof method

context *BD-Security*

begin

definition *consume* :: $('state, 'act, 'out) trans \Rightarrow 'value list \Rightarrow 'value list \Rightarrow bool$ **where**
consume $trn\ vl\ vl' \equiv$
 if $\varphi\ trn$ then $vl \neq [] \wedge f\ trn = hd\ vl \wedge vl' = tl\ vl$
 else $vl' = vl$

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

lemma *ex-consume-φ*:
assumes $\neg \varphi\ trn$
obtains vl' **where** *consume* $trn\ vl\ vl'$
 ⟨proof⟩

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

definition *iaction* **where**

iaction $\Delta\ s\ vl\ s1\ vl1 \equiv$
 $\exists\ a1\ ou1\ s1'\ vl1'.$
 let $trn1 = Trans\ s1\ a1\ ou1\ s1'$ in
 validTrans $trn1 \wedge$
 $\varphi\ trn1 \wedge consume\ trn1\ vl1\ vl1' \wedge$
 $\neg\ \gamma\ trn1$
 \wedge
 $\Delta\ s\ vl\ s1'\ vl1'$

lemma *iactionI[intro?]*:
assumes $step\ s1\ a1 = (ou1, s1')$ **and** $\varphi\ (Trans\ s1\ a1\ ou1\ s1')$
and *consume* $(Trans\ s1\ a1\ ou1\ s1')\ vl1\ vl1'$
and $\neg\ \gamma\ (Trans\ s1\ a1\ ou1\ s1')$ **and** $\Delta\ s\ vl\ s1'\ vl1'$
shows *iaction* $\Delta\ s\ vl\ s1\ vl1$
 ⟨proof⟩

definition *match* **where**
match $\Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl' \equiv$

$\exists a1\ ou1\ s1'\ vl1'$.
 let $trn = Trans\ s\ a\ ou\ s'$; $trn1 = Trans\ s1\ a1\ ou1\ s1'$ in
 $validTrans\ trn1 \wedge$
 $consume\ trn1\ vl1\ vl1' \wedge$
 $\gamma\ trn = \gamma\ trn1 \wedge (\gamma\ trn \longrightarrow g\ trn = g\ trn1) \wedge$
 $\Delta\ s'\ vl'\ s1'\ vl1'$

lemma *matchI*[*intro?*]:
assumes $validTrans\ (Trans\ s1\ a1\ ou1\ s1')$
and $consume\ (Trans\ s1\ a1\ ou1\ s1')\ vl1\ vl1'$ **and** $\gamma\ (Trans\ s\ a\ ou\ s') = \gamma\ (Trans\ s1\ a1\ ou1\ s1')$
and $\gamma\ (Trans\ s\ a\ ou\ s') \implies g\ (Trans\ s\ a\ ou\ s') = g\ (Trans\ s1\ a1\ ou1\ s1')$
and $\Delta\ s'\ vl'\ s1'\ vl1'$
shows $match\ \Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl'$
<proof>

definition *ignore where*
 $ignore\ \Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl' \equiv$
 $\neg\ \gamma\ (Trans\ s\ a\ ou\ s') \wedge$
 $\Delta\ s'\ vl'\ s1\ vl1$

lemma *ignoreI*[*intro?*]:
assumes $\neg\ \gamma\ (Trans\ s\ a\ ou\ s')$ **and** $\Delta\ s'\ vl'\ s1\ vl1$
shows $ignore\ \Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl'$
<proof>

definition *reaction where*
 $reaction\ \Delta\ s\ vl\ s1\ vl1 \equiv$
 $\forall\ a\ ou\ s'\ vl'$.
 let $trn = Trans\ s\ a\ ou\ s'$ in
 $validTrans\ trn \wedge \neg\ T\ trn \wedge$
 $consume\ trn\ vl\ vl'$
 \longrightarrow
 $match\ \Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl'$
 \vee
 $ignore\ \Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl'$

lemma *reactionI*[*intro?*]:
assumes
 $\bigwedge a\ ou\ s'\ vl'$.
 $\llbracket step\ s\ a = (ou, s'); \neg\ T\ (Trans\ s\ a\ ou\ s');$
 $consume\ (Trans\ s\ a\ ou\ s')\ vl\ vl' \rrbracket$
 \implies
 $match\ \Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl' \vee ignore\ \Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl'$
shows $reaction\ \Delta\ s\ vl\ s1\ vl1$
<proof>

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

lemma *exit-coind*:
assumes $K: K\ s$
and $I: \bigwedge\ trn. \llbracket K\ (srcOf\ trn);\ validTrans\ trn; \neg\ T\ trn \rrbracket$
 $\implies (\varphi\ trn \longrightarrow f\ trn \neq v) \wedge K\ (tgtOf\ trn)$
shows $exit\ s\ v$
 $\langle proof \rangle$

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

lemma *noVal-disj*:
assumes $noVal\ Inv1\ v$ **and** $noVal\ Inv2\ v$
shows $noVal\ (\lambda\ s.\ Inv1\ s \vee Inv2\ s)\ v$
 $\langle proof \rangle$

lemma *noVal-conj*:
assumes $noVal\ Inv1\ v$ **and** $noVal\ Inv2\ v$
shows $noVal\ (\lambda\ s.\ Inv1\ s \wedge Inv2\ s)\ v$
 $\langle proof \rangle$

definition *no φ* **where**
 $no\varphi\ K \equiv \forall\ s\ a\ ou\ s'.\ reachNT\ s \wedge K\ s \wedge step\ s\ a = (ou, s') \longrightarrow \neg\ \varphi\ (Trans\ s\ a\ ou\ s')$

lemma *no φ -noVal*: $no\varphi\ K \implies noVal\ K\ v$
 $\langle proof \rangle$

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

lemma *exitI2*:
assumes $rs: reachNT\ s$ **and** $K: K\ s$
and $invarNT\ K$ **and** $noVal\ K\ v$
shows $exit\ s\ v$
 $\langle proof \rangle$

definition *noVal2* **where**
 $noVal2\ K\ v \equiv$

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

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* \implies *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*:

$\bigwedge s a ou s'.$

$\llbracket reach s; reachNT s; step s a = (ou, s'); K s v \rrbracket$

$\implies (\varphi (Trans s a ou s') \longrightarrow f (Trans s a ou s') \neq v) \wedge K s' v$

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
 $\bigwedge s\ vl\ s1\ vl1.$
 $\llbracket reachNT\ s; reach\ s1; \Delta\ s\ vl\ s1\ vl1 \rrbracket$
 \implies
 $(vl \neq [] \wedge exit\ s\ (hd\ vl))$
 \vee
 $iaction\ \Delta\ s\ vl\ s1\ vl1$
 \vee
 $((vl \neq [] \vee vl1 = []) \wedge reaction\ \Delta\ s\ vl\ s1\ vl1)$
shows *unwind* Δ
 $\langle proof \rangle$

lemma *unwind-trace*:
assumes *unwind*: *unwind* Δ **and** *reachNT* s **and** *reach* $s1$ **and** $\Delta\ s\ vl\ s1\ vl1$
and *validFrom* $s\ tr$ **and** *never* $T\ tr$ **and** $V\ tr = vl$
shows $\exists tr1. validFrom\ s1\ tr1 \wedge O\ tr1 = O\ tr \wedge V\ tr1 = vl1$
 $\langle proof \rangle$

theorem *unwind-secure*:
assumes *init*: $\bigwedge vl\ vl1. B\ vl\ vl1 \implies \Delta\ istate\ vl\ istate\ vl1$
and *unwind*: *unwind* Δ
shows *secure*
 $\langle proof \rangle$

5 Compositional Reasoning

context *BD-Security* **begin**

5.1 Preliminaries

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

lemma *disjAll-simps*[*simp*]:
 $disjAll\ \{\} \equiv \lambda - - -. False$
 $disjAll\ (insert\ \Delta\ \Delta s) \equiv \lambda s\ vl\ s1\ vl1. \Delta\ s\ vl\ s1\ vl1 \vee disjAll\ \Delta s\ s\ vl\ s1\ vl1$
 $\langle proof \rangle$

lemma *iaction-mono*:
assumes $1: iaction\ \Delta\ s\ vl\ s1\ vl1$ **and** $2: \bigwedge s\ vl\ s1\ vl1. \Delta\ s\ vl\ s1\ vl1 \implies \Delta'\ s\ vl\ s1\ vl1$
shows $iaction\ \Delta'\ s\ vl\ s1\ vl1$
 $\langle proof \rangle$

lemma *match-mono*:
assumes $1: match\ \Delta\ s\ s1\ vl1\ a\ ou\ s'\ vl'$ **and** $2: \bigwedge s\ vl\ s1\ vl1. \Delta\ s\ vl\ s1\ vl1 \implies \Delta'\ s\ vl\ s1\ vl1$
shows $match\ \Delta'\ s\ s1\ vl1\ a\ ou\ s'\ vl'$

<proof>

lemma *ignore-mono*:

assumes 1: *ignore* Δ s $s1$ $vl1$ *a ou* s' vl' **and** 2: $\bigwedge s$ vl $s1$ $vl1$. Δ s vl $s1$ $vl1 \implies \Delta' s$ 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: $\bigwedge s$ vl $s1$ $vl1$. Δ s vl $s1$ $vl1 \implies \Delta' s$ 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 $\Delta s \equiv$

$\forall s$ vl $s1$ $vl1$.

reachNT $s \wedge$ *reach* $s1 \wedge \Delta$ s vl $s1$ $vl1$

\longrightarrow

$vl \neq [] \wedge$ *exit* s (*hd* vl)

\vee

iaction (*disjAll* Δs) s vl $s1$ $vl1$

\vee

$(vl \neq [] \vee vl1 = []) \wedge$ *reaction* (*disjAll* Δs) s vl $s1$ $vl1$

lemma *unwind-toI[intro?]*:

assumes

$\bigwedge s$ vl $s1$ $vl1$.

\llbracket *reachNT* s ; *reach* $s1$; Δ s vl $s1$ $vl1$ \rrbracket

\implies

$vl \neq [] \wedge$ *exit* s (*hd* vl)

\vee

iaction (*disjAll* Δs) s vl $s1$ $vl1$

\vee

$(vl \neq [] \vee vl1 = []) \wedge$ *reaction* (*disjAll* Δs) s vl $s1$ $vl1$

shows *unwind-to* Δ Δs

<proof>

lemma *unwind-dec*:

assumes *ne*: $\bigwedge \Delta$. $\Delta \in \Delta s \implies$ *next* $\Delta \subseteq \Delta s \wedge$ *unwind-to* Δ (*next* Δ)

shows *unwind* (*disjAll* Δs) (*is unwind ?* Δ)

<proof>

lemma *init-dec*:

assumes $\Delta 0$: $\Delta 0 \in \Delta s$

and *i*: $\bigwedge vl$ $vl1$. *B* vl $vl1 \implies \Delta 0$ *istate* vl *istate* $vl1$

shows $\forall vl$ $vl1$. *B* vl $vl1 \longrightarrow$ *disjAll* Δs *istate* vl *istate* $vl1$

<proof>

theorem *unwind-dec-secure*:
assumes $\Delta 0$: $\Delta 0 \in \Delta s$
and i : $\bigwedge vl\ vl1. B\ vl\ vl1 \implies \Delta 0\ ystate\ vl\ ystate\ vl1$
and ne : $\bigwedge \Delta. \Delta \in \Delta s \implies next\ \Delta \subseteq \Delta s \wedge unwind\text{-to}\ \Delta\ (next\ \Delta)$
shows *secure*
 $\langle proof \rangle$

5.3 A customization for linear modular reasoning

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

lemma *unwind-contI*[*intro?*]:
assumes
 $\bigwedge s\ vl\ s1\ vl1.$
 $\llbracket reachNT\ s; reach\ s1; \Delta\ s\ vl\ s1\ vl1 \rrbracket$
 \implies
 $iaction\ (disjAll\ \Delta s)\ s\ vl\ s1\ vl1$
 \vee
 $((vl \neq [] \vee vl1 = []) \wedge reaction\ (disjAll\ \Delta s)\ s\ vl\ s1\ vl1)$
shows *unwind-cont* $\Delta\ \Delta s$
 $\langle proof \rangle$

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

lemma *unwind-exitI*[*intro?*]:
assumes
 $\bigwedge s\ vl\ s1\ vl1.$
 $\llbracket reachNT\ s; reach\ s1; \Delta e\ s\ vl\ s1\ vl1 \rrbracket$
 \implies
 $vl \neq [] \wedge exit\ s\ (hd\ vl)$
shows *unwind-exit* Δe
 $\langle proof \rangle$

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 $\exists \Delta 2. (\Delta, \Delta 2) \in allConsec \Delta s$

<proof>

lemma *allConsec-set*:

assumes $(\Delta 1, \Delta 2) \in allConsec \Delta s$

shows $\Delta 1 \in set \Delta s \wedge \Delta 2 \in set \Delta s$

<proof>

theorem *unwind-decomp-secure*:

assumes $n: \Delta s \neq []$

and $i: \bigwedge vl \ vl1. B \ vl \ vl1 \implies hd \ \Delta s \ istate \ vl \ istate \ vl1$

and $c: \bigwedge \Delta 1 \ \Delta 2. (\Delta 1, \Delta 2) \in allConsec \Delta s \implies unwind-cont \ \Delta 1 \ \{\Delta 1, \Delta 2, \Delta e\}$

and $l: unwind-cont (last \ \Delta s) \ \{last \ \Delta s, \Delta e\}$

and $e: unwind-exit \ \Delta e$

shows *secure*

<proof>

5.4 Instances

corollary *unwind-decomp3-secure*:

assumes

$i: \bigwedge vl \ vl1. B \ vl \ vl1 \implies \Delta 1 \ istate \ vl \ istate \ vl1$

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 $e: unwind-exit \ \Delta e$

shows *secure*

<proof>

corollary *unwind-decomp4-secure*:

assumes

$i: \bigwedge vl \ vl1. B \ vl \ vl1 \implies \Delta 1 \ istate \ vl \ istate \ vl1$

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 $e: unwind-exit \ \Delta e$

shows *secure*

<proof>

corollary *unwind-decomp5-secure*:

assumes

$i: \bigwedge vl \ vl1. B \ vl \ vl1 \implies \Delta 1 \ istate \ vl \ istate \ vl1$

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>