Bounded-Deducibility Security

Andrei Popescu Peter Lammich Thomas Bauereiss

Contents

1 Introduction

This is a formalization of *Bounded-Deducibility Security* (*BD Security*), a flexible notion of information-flow security applicable to arbitrary transition systems. It generalizes Sutherland's classic notion of nondeducibility [7] 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 was introduced in [4], where an application to the verification of a conference management called CoCon system is also presented. The framework is further discussed in detail in [6] and [5].

Other verification case studies of BD Security are discussed in [1, 3] and [2].
⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof

2 Preliminaries

function *filtermap* ::
(*'trans* ⇒ *bool*) ⇒ (*'trans* ⇒ *'a*) ⇒ *'trans list* ⇒ *'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*⟩

lemma *filtermap-map-filter*: *filtermap pred func xs* = *map func* (*filter pred xs*)
⟨*proof*⟩

lemma *filtermap-append*: *filtermap pred func* (*tr* @ *tr1*) = *filtermap pred func tr* @ *filtermap pred func tr1*
⟨*proof*⟩

lemma *filtermap-Nil-list-ex*: *filtermap pred func tr* = [] ⟷ ¬ *list-ex pred tr*
⟨*proof*⟩

lemma *filtermap-Nil-never*: *filtermap pred func tr* = [] ⟷ *never pred tr*
⟨*proof*⟩

lemma *length-filtermap*: *length* (*filtermap pred func tr*) ≤ *length tr*
⟨*proof*⟩

lemma *filtermap-list-all*[*simp*]: *filtermap pred func tr* = *map func tr* ⟷ *list-all pred tr*
⟨*proof*⟩

lemma *filtermap-eq-Cons*:
assumes *filtermap pred func tr* = *a* # *al1*
shows ∃ *trn tr2 tr1*.
tr = *tr2* @ [*trn*] @ *tr1* ∧ *never pred tr2* ∧ *pred trn* ∧ *func trn* = *a* ∧ *filtermap pred func tr1* = *al1*
⟨*proof*⟩

lemma *filtermap-eq-append*:
assumes *filtermap pred func tr = al1 @ al2*
shows ∃ *tr1 tr2. tr = tr1 @ tr2* ∧ *filtermap pred func tr1 = al1* ∧ *filtermap pred func tr2 = al2*
⟨*proof*⟩

lemma *holds-filtermap-RCons*[*simp*]:
pred trn ⟹ *filtermap pred func* (*tr* ## *trn*) = *filtermap pred func tr* ## *func trn*
⟨*proof*⟩

lemma *not-holds-filtermap-RCons*[*simp*]:
¬ *pred trn* ⟹ *filtermap pred func* (*tr* ## *trn*) = *filtermap pred func tr*
⟨*proof*⟩

lemma *filtermap-eq-RCons*:
assumes *filtermap pred func tr = al1* ## *a*
shows ∃ *trn tr1 tr2*.
tr = tr1 @ [*trn*] @ *tr2* ∧ *never pred tr2* ∧ *pred trn* ∧ *func trn = a* ∧ *filtermap pred func tr1 = al1*
⟨*proof*⟩

lemma *filtermap-eq-Cons-RCons*:
assumes *filtermap pred func tr = a* # *al1* ## *b*
shows ∃ *tra trna tr1 trnb trb*.
tr = tra @ [*trna*] @ *tr1* @ [*trnb*] @ *trb* ∧
never pred tra ∧
pred trna ∧ *func trna = a* ∧
filtermap pred func tr1 = al1 ∧
pred trnb ∧ *func trnb = b* ∧
never pred trb
⟨*proof*⟩

lemma *filter-Nil-never*: [] = *filter pred xs* ⟹ *never pred xs*
⟨*proof*⟩

lemma *never-Nil-filter*: *never pred xs* ⟷ [] = *filter pred xs*
⟨*proof*⟩

lemma *snoc-eq-filterD*:
assumes *xs* ## *x = filter Q ys*
obtains *us vs* **where** *ys = us* @ *x* # *vs* **and** *never Q vs* **and** *Q x* **and** *xs = filter Q us*
⟨*proof*⟩

lemma *filtermap-Cons2-eq*:
filtermap pred func [*x*, *x'*] = *filtermap pred func* [*y*, *y'*]
⟹ *filtermap pred func* (*x* # *x'* # *zs*) = *filtermap pred func* (*y* # *y'* # *zs*)
⟨*proof*⟩

lemma *filtermap-Cons-cong*:
filtermap pred func xs = filtermap pred func ys

$\Longrightarrow$ *filtermap pred func* $(x \# xs)$ = *filtermap pred func* $(x \# ys)$
⟨*proof*⟩

lemma *set-filtermap*:
set (*filtermap pred func xs*) $\subseteq$ {*func x* | *x* . *x* ∈∈ *xs* ∧ *pred x*}
⟨*proof*⟩

2.1 Transition Systems

We define transition systems, their valid traces, and state rechability.

2.1.1 Traces

type-synonym *'trans trace* = *'trans list*

locale *Transition-System* =
fixes *istate* :: *'state*
  and *validTrans* :: *'trans* ⇒ *bool*
  and *srcOf* :: *'trans* ⇒ *'state*
  and *tgtOf* :: *'trans* ⇒ *'state*
begin

fun *srcOfTr* **where** *srcOfTr tr* = *srcOf*(*hd tr*)
fun *tgtOfTr* **where** *tgtOfTr tr* = *tgtOf*(*last tr*)

fun *srcOfTrFrom* **where**
  srcOfTrFrom s [] = *s*
| *srcOfTrFrom s tr* = *srcOfTr tr*

lemma *srcOfTrFrom-srcOfTr*[*simp*]:
  tr ≠ [] $\Longrightarrow$ *srcOfTrFrom s tr* = *srcOfTr tr*
  ⟨*proof*⟩

fun *tgtOfTrFrom* **where**
  tgtOfTrFrom s [] = *s*
| *tgtOfTrFrom s tr* = *tgtOfTr tr*

lemma *tgtOfTrFrom-tgtOfTr*[*simp*]:
  tr ≠ [] $\Longrightarrow$ *tgtOfTrFrom s tr* = *tgtOfTr tr*
  ⟨*proof*⟩

Traces allowed by the system (starting in any given state), with two alternative definitions: growing from the left and growing from the right:

inductive *valid* :: *'trans trace* ⇒ *bool* **where**
Singl[*simp*,*intro*!]:
validTrans trn
 $\Longrightarrow$

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* :: ′*trans 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*⟩

lemma *valid-Cons-iff*:
valid (*trn* # *tr*) ⟷ *validTrans trn* ∧ ((*tgtOf trn* = *srcOf* (*hd tr*) ∧ *valid tr*) ∨ *tr* = [])
⟨*proof*⟩

lemma *valid-append*:
tr ≠ [] ⟹ *tr1* ≠ [] ⟹
valid (*tr* @ *tr1*) ⟷ *valid tr* ∧ *valid tr1* ∧ *tgtOf* (*last tr*) = *srcOf* (*hd tr1*)
⟨*proof*⟩

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

definition *validFrom* :: ′*state* ⇒ ′*trans 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-append*:
validFrom s (*tr* @ *tr1*) ⟷ (*tr* = [] ∧ *validFrom s tr1*) ∨ (*tr* ≠ [] ∧ *validFrom s tr* ∧ *validFrom* (*tgtOf* (*last tr*)) *tr1*)
⟨*proof*⟩

lemma *validFrom-Cons*:
validFrom s (*trn* # *tr*) ⟷ *validTrans trn* ∧ *srcOf trn* = *s* ∧ *validFrom* (*tgtOf trn*) *tr*
⟨*proof*⟩

2.1.2 Reachability

inductive *reach* :: ′*state* ⇒ *bool* **where**
Istate: *reach istate*
|
Step: *reach s* ⟹ *validTrans trn* ⟹ *srcOf trn* = *s* ⟹ *tgtOf trn* = *s*′ ⟹ *reach s*′

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 *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*⟩

inductive *reachFrom* :: $'state \Rightarrow 'state \Rightarrow bool$
for *s* :: $'state$
where
Refl[*intro*]: *reachFrom s s*
| *Step*: $[\![reachFrom\ s\ s';\ validTrans\ trn;\ srcOf\ trn = s';\ tgtOf\ trn = s'']\!] \Longrightarrow reachFrom\ s\ s''$

lemma *reachFrom-Step1*:
$[\![validTrans\ trn;\ srcOf\ trn = s;\ tgtOf\ trn = s']\!] \Longrightarrow reachFrom\ s\ s'$
⟨*proof*⟩

lemma *reachFrom-Step-Left*:
$reachFrom\ s'\ s'' \Longrightarrow validTrans\ trn \Longrightarrow srcOf\ trn = s \Longrightarrow tgtOf\ trn = s' \Longrightarrow reachFrom\ s\ s''$
⟨*proof*⟩

lemma *reachFrom-trans*: $reachFrom\ s0\ s1 \Longrightarrow reachFrom\ s1\ s2 \Longrightarrow reachFrom\ s0\ s2$
⟨*proof*⟩

lemma *reachFrom-reach*: $reachFrom\ s\ s' \Longrightarrow reach\ s \Longrightarrow reach\ s'$
⟨*proof*⟩

lemma *valid-validTrans-set*:
assumes *valid tr* **and** $trn \in\in tr$
shows *validTrans trn*
⟨*proof*⟩

lemma *validFrom-validTrans-set*:
assumes *validFrom s tr* **and** *trn* $\in\in$ *tr*
shows *validTrans trn*
⟨*proof*⟩

lemma *valid-validTrans-nth*:
assumes *v*: *valid tr* **and** *i*: $i <$ *length tr*
shows *validTrans* (*tr*!*i*)
⟨*proof*⟩

lemma *valid-validTrans-nth-srcOf-tgtOf*:
assumes *v*: *valid tr* **and** *i*: *Suc i* $<$ *length tr*
shows *srcOf* (*tr*!(*Suc i*)) = *tgtOf* (*tr*!*i*)
⟨*proof*⟩

lemma *validFrom-reach*: *validFrom s tr* $\Longrightarrow$ *reach s* $\Longrightarrow$ *tr* $\neq$ [] $\Longrightarrow$ *reach* (*tgtOf* (*last tr*))
⟨*proof*⟩

end

2.2 IO automata

IO automata are defined. Since they are a particular kind of transition systems, they inherit the notions of traces and reachability from those. Various useful concepts and theorems are provided, including invariants and the multi-step operator.

2.2.1 IO automata as transition systems

In this context, transitions are quadruples consisting of a source state, an action (input), and output and a target state.

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

lemmas *srcOf-simps* = *trans.sel*(*1*)
lemmas *actOf-simps* = *trans.sel*(*2*)
lemmas *outOf-simps* = *trans.sel*(*3*)
lemmas *tgtOf-simps* = *trans.sel*(*4*)

locale *IO-Automaton* =
fixes *istate* :: *'state*
  and *step* :: *'state* $\Rightarrow$ *'act* $\Rightarrow$ *'out* $*$ *'state*
begin

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*)

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

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

sublocale *Transition-System*
where $\mathit{istate} = \mathit{istate}$ **and** $\mathit{validTrans} = \mathit{validTrans}$ **and** $\mathit{srcOf} = \mathit{srcOf}$ **and** $\mathit{tgtOf} = \mathit{tgtOf}$ $\langle \mathit{proof} \rangle$

lemma *reach-step*:
$\mathit{reach}\ s \Longrightarrow \mathit{reach}\ (\mathit{snd}\ (\mathit{step}\ s\ a))$
$\langle \mathit{proof} \rangle$

lemma *reach-PairI*:
assumes $\mathit{reach}\ s$ **and** $\mathit{step}\ s\ a = (ou,\ s')$
shows $\mathit{reach}\ s'$
$\langle \mathit{proof} \rangle$

lemma *reach-step-induct*[*consumes 1*, *case-names Istate Step*]:
assumes s: $\mathit{reach}\ s$
and *istate*: $P\ \mathit{istate}$
and *step*: $\bigwedge s\ a.\ \mathit{reach}\ s \Longrightarrow P\ s \Longrightarrow P\ (\mathit{snd}\ (\mathit{step}\ s\ a))$
shows $P\ s$
$\langle \mathit{proof} \rangle$

lemma *reachFrom-step-induct*[*consumes 1*, *case-names Refl Step*]:
assumes s: $\mathit{reachFrom}\ s\ s'$
and *refl*: $P\ s$
and *step*: $\bigwedge s'\ a\ ou\ s''.\ \mathit{reachFrom}\ s\ s' \Longrightarrow P\ s' \Longrightarrow \mathit{step}\ s'\ a = (ou,\ s'') \Longrightarrow P\ s''$
shows $P\ s'$
$\langle \mathit{proof} \rangle$

lemma *valid-filter-no-state-change*:
$\mathit{valid}\ \mathit{tr} \Longrightarrow (\bigwedge \mathit{trn}.\ \mathit{trn} \in\in \mathit{tr} \Longrightarrow \neg(\mathit{PP}\ \mathit{trn}) \Longrightarrow \mathit{srcOf}\ \mathit{trn} = \mathit{tgtOf}\ \mathit{trn}) \Longrightarrow$
$\exists\, \mathit{trn}.\ \mathit{trn} \in\in \mathit{tr} \wedge \mathit{PP}\ \mathit{trn} \Longrightarrow \mathit{valid}\ (\mathit{filter}\ \mathit{PP}\ \mathit{tr}) \wedge \mathit{srcOfTr}\ \mathit{tr} = \mathit{srcOfTr}\ (\mathit{filter}\ \mathit{PP}\ \mathit{tr})$
$\wedge\ \mathit{tgtOfTr}\ \mathit{tr} = \mathit{tgtOfTr}\ (\mathit{filter}\ \mathit{PP}\ \mathit{tr})$
$\langle \mathit{proof} \rangle$

lemma *validFrom-validTrans*[*intro*]:
assumes $\mathit{validTrans}\ (\mathit{Trans}\ s\ a\ ou\ s')$ **and** $\mathit{validFrom}\ s'\ \mathit{tr}$
shows $\mathit{validFrom}\ s\ (\mathit{Trans}\ s\ a\ ou\ s'\ \#\ \mathit{tr})$
$\langle \mathit{proof} \rangle$

2.2.2 State invariants

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

definition *invar* :: ('*state* ⇒ *bool*) ⇒ *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* φ **and** *i*: *invar* φ **and** *a*: *reach s*
shows φ *s*
⟨*proof*⟩

2.2.3 Traces of actions

fun *traceOf* :: '*state* ⇒ '*act list* ⇒ ('*state*,'*act*,'*out*) *trans trace* **where**
traceOf s [] = []
|
traceOf s (*a* # *al*) =
(*case step s a of* (*ou*,*s1*) ⇒ (*Trans s a ou s1*) # *traceOf s1 al*)

fun *sstep* :: '*state* ⇒ '*act list* ⇒ '*out list* × '*state* **where**
sstep s [] = ([], *s*)
|
sstep s (*a* # *al*) = (*case step s a of* (*ou*,*s*′) ⇒ (*case sstep s*′ *al of* (*oul*, *s*′′) ⇒ (*ou* # *oul*, *s*′′)))

lemma *length-traceOf*[*simp*]:
length (*traceOf s al*) = *length al*
⟨*proof*⟩

lemma *traceOf-Nil*[*simp*]:
traceOf s al = [] ⟷ *al* = []
⟨*proof*⟩

lemma *sstep-outOf-traceOf*[*simp*]:
sstep s al = (*ou*,*s*′) ⟹ *map outOf* (*traceOf s al*) = *ou*
⟨*proof*⟩

lemma *sstep-tgtOf-traceOf*[*simp*]:
al ≠ [] ⟹ *sstep s al* = (*ou*,*s*′) ⟹ *tgtOf* (*last* (*traceOf s al*)) = *s*′
⟨*proof*⟩

lemma *srcOf-traceOf*[*simp*]:
al ≠ [] ⟹ *srcOf* (*hd* (*traceOf s al*)) = *s*
⟨*proof*⟩

lemma *actOf-traceOf*[*simp*]:
map actOf (*traceOf s al*) = *al*
⟨*proof*⟩

lemma *traceOf-append*:
$al \neq [] \Longrightarrow s1 = tgtOf\ (last\ (traceOf\ s\ al)) \Longrightarrow$
$traceOf\ s\ (al\ @\ al1) = traceOf\ s\ al\ @\ traceOf\ s1\ al1$
⟨*proof*⟩

lemma *sstep-append*:
assumes $sstep\ s\ al = (oul,s1)$ **and** $sstep\ s1\ al1 = (oul1,s2)$
shows $sstep\ s\ (al\ @\ al1) = (oul\ @\ oul1,\ s2)$
⟨*proof*⟩

lemma *reach-sstep*:
assumes $reach\ s$ **and** $sstep\ s\ al = (ou,s1)$
shows $reach\ s1$
⟨*proof*⟩

lemma *traceOf-consR*[*simp*]:
assumes $al \neq []$ **and** $s1 = tgtOf\ (last\ (traceOf\ s\ al))$ **and** $step\ s1\ a = (ou,s2)$
shows $traceOf\ s\ (al\ \#\#\ a) = traceOf\ s\ al\ \#\#\ Trans\ s1\ a\ ou\ s2$
⟨*proof*⟩

lemma *sstep-consR*[*simp*]:
assumes $sstep\ s\ al = (oul,s1)$ **and** $step\ s1\ a = (ou,s2)$
shows $sstep\ s\ (al\ \#\#\ a) = (oul\ \#\#\ ou,\ s2)$
⟨*proof*⟩

lemma *fst-sstep-consR*:
$fst\ (sstep\ s\ (al\ \#\#\ a)) = fst\ (sstep\ s\ al)\ \#\#\ (fst\ (step\ (snd\ (sstep\ s\ al))\ a))$
⟨*proof*⟩

lemma *valid-traceOf*[*simp*]: $al \neq [] \Longrightarrow valid\ (traceOf\ s\ al)$
⟨*proof*⟩

lemma *validFrom-traceOf*[*simp*]: $validFrom\ s\ (traceOf\ s\ al)$
⟨*proof*⟩

lemma *validFrom-traceOf2*:
assumes $validFrom\ s\ tr$
shows $tr = traceOf\ s\ (map\ actOf\ tr)$
⟨*proof*⟩

lemma *set-traceOf-validTrans*:
assumes $trn \in\in traceOf\ s\ al$ **shows** $validTrans\ trn$
⟨*proof*⟩

lemma *traceOf-append-sstep*: $traceOf\ s\ (al\ @\ al1) = traceOf\ s\ al\ @\ traceOf\ (snd\ (sstep\ s\ al))\ al1$
⟨*proof*⟩

lemma *snd-sstep-append*: *snd* (*sstep s* (*al* @ *al1*)) = *snd* (*sstep* (*snd* (*sstep s al*)) *al1*)
⟨*proof*⟩

lemma *snd-sstep-step-constant*:
assumes ∀ *a*. *a* ∈∈ *al* ⟶ *snd* (*step s a*) = *s*
shows *snd* (*sstep s al*) = *s*
⟨*proof*⟩

definition *const-tr tr* ≡ ∀ *trn*. *trn* ∈∈ *tr* ⟶ *srcOf trn* = *tgtOf trn*

lemma *const-tr-same-src-tgt*:
  assumes *valid tr const-tr tr*
  shows *srcOfTr tr* = *tgtOfTr tr*
⟨*proof*⟩

lemma *traceOf-snoc*:
traceOf s (*al* ## *a*) =
  traceOf s al ##
  Trans (*snd* (*sstep s al*))
        a
        (*fst* (*step* (*snd* (*sstep s al*)) *a*))
        (*snd* (*step* (*snd* (*sstep s al*)) *a*))
⟨*proof*⟩

lemma *traceOf-append-unfold*:
traceOf s (*al1* @ *al2*) =
 traceOf s al1 @ *traceOf* (*if al1* = [] *then s else tgtOf* (*last* (*traceOf s al1*))) *al2*
⟨*proof*⟩

abbreviation *transOf s a* ≡ *Trans s a* (*fst* (*step s a*)) (*snd* (*step s a*))

lemma *traceOf-Cons*: *traceOf s* (*a* # *al*) = *transOf s a* # *traceOf* (*snd* (*step s a*)) *al*
⟨*proof*⟩

definition *commute s a1 a2*
 ≡ *snd* (*sstep s* [*a1*, *a2*]) = *snd* (*sstep s* [*a2*, *a1*])

definition *absorb* :: ′*state* ⇒ ′*act* ⇒ ′*act* ⇒ *bool* **where**
absorb s a1 a2 ≡ *snd* (*sstep s* [*a1*, *a2*]) = *snd* (*step s a2*)

lemma *validFrom-commute*:
  assumes *validFrom s0* (*tr1* @ *transOf s a* # *transOf* (*snd* (*step s a*)) *a*′ # *tr2*)
    and *commute s a a*′
  shows *validFrom s0* (*tr1* @ *transOf s a*′ # *transOf* (*snd* (*step s a*′)) *a* # *tr2*)
⟨*proof*⟩

lemma *validFrom-absorb*:

assumes *validFrom s0* (*tr1* @ *transOf s a* # *transOf* (*snd* (*step s a*)) *a'* # *tr2*)
and *absorb s a a'*
shows *validFrom s0* (*tr1* @ *transOf s a'* # *tr2*)
⟨*proof*⟩

lemma *validTrans-Trans-srcOf-actOf-tgtOf*:
validTrans trn ⟹ *Trans* (*srcOf trn*) (*actOf trn*) (*outOf trn*) (*tgtOf trn*) = *trn*
⟨*proof*⟩

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

lemma *sstep-Cons*:
sstep s (*a* # *al*) = (*fst* (*step s a*) # *fst* (*sstep* (*snd* (*step s a*)) *al*), *snd* (*sstep* (*snd* (*step s a*)) *al*))
⟨*proof*⟩
declare *sstep.simps*(*2*)[*simp del*]

lemma *length-fst-sstep*: *length* (*fst* (*sstep s al*)) = *length al*
⟨*proof*⟩

3 BD Security

3.1 Abstract definition

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

locale *Abstract-BD-Security* =
fixes
validSystemTrace :: *'traces* ⇒ *bool*
and — secret values:
V :: *'traces* ⇒ *'values*
and — observations:
O :: *'traces* ⇒ *'observations*
and — declassification bound:
B :: *'values* ⇒ *'values* ⇒ *bool*
and — declassification trigger:
TT :: *'traces* ⇒ *bool*
begin

A system is considered to be secure if, for all traces that satisfy a given condition (later instantiated to be the absence of transitions satisfying a declassification trigger condition, releasing the secret information), the secret value can be replaced by another secret value within the declassification bound, without changing the observation. Hence, an observer cannot distinguish secrets related by the declassification bound, unless and until release of the secret information is allowed by the declassification trigger.

definition *secure* :: *bool* **where**
secure ≡

$\forall$ *tr vl vl1*.
validSystemTrace tr $\wedge$ *TT tr* $\wedge$ *B vl vl1* $\wedge$ *V tr* = *vl* $\longrightarrow$
($\exists$ *tr1*. *validSystemTrace tr1* $\wedge$ *O tr1* = *O tr* $\wedge$ *V tr1* = *vl1*)

lemma *secureE*:
assumes *secure* **and** *validSystemTrace tr* **and** *TT tr* **and** *B* (*V tr*) *vl1*
obtains *tr1* **where** *validSystemTrace tr1 O tr1* = *O tr V tr1* = *vl1*
⟨*proof*⟩

end

3.2 Instantiation for transition systems

declare *Let-def*[*simp*]

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

locale *BD-Security-TS* = *Transition-System istate validTrans srcOf tgtOf*
for *istate* :: *'state* **and** *validTrans* :: *'trans* ⇒ *bool*
and *srcOf* :: *'trans* ⇒ *'state* **and** *tgtOf* :: *'trans* ⇒ *'state*
+
fixes
φ :: *'trans* => *bool* **and** *f* :: *'trans* ⇒ *'value*
and
γ :: *'trans* => *bool* **and** *g* :: *'trans* ⇒ *'obs*
and
T :: *'trans* ⇒ *bool*
and
B :: *'value list* ⇒ *'value list* ⇒ *bool*
begin

definition *V* :: *'trans list* ⇒ *'value list* **where** *V* ≡ *filtermap* φ *f*

definition *O* :: *'trans trace* ⇒ *'obs list* **where** *O* ≡ *filtermap* γ *g*

sublocale *Abstract-BD-Security*
where *validSystemTrace* = *validFrom istate* **and** *V* = *V* **and** *O* = *O* **and** *B* = *B* **and** *TT* = *never T*
⟨*proof*⟩

lemma *O-map-filter*: *O tr* = *map g* (*filter* γ *tr*) ⟨*proof*⟩
lemma *V-map-filter*: *V tr* = *map f* (*filter* φ *tr*) ⟨*proof*⟩

lemma *V-simps*[*simp*]:
V [] = [] $\neg$ φ *trn* $\Longrightarrow$ *V* (*trn* # *tr*) = *V tr* φ *trn* $\Longrightarrow$ *V* (*trn* # *tr*) = *f trn* # *V tr*
⟨*proof*⟩

lemma *V-Cons-unfold*: $V\ (trn\ \#\ tr) = (if\ \varphi\ trn\ then\ f\ trn\ \#\ V\ tr\ else\ V\ tr)$
⟨*proof*⟩

lemma *O-simps*[*simp*]:
$O\ [] = []\ \ \neg\ \gamma\ trn \Longrightarrow O\ (trn\ \#\ tr) = O\ tr\ \ \gamma\ trn \Longrightarrow O\ (trn\ \#\ tr) = g\ trn\ \#\ O\ tr$
⟨*proof*⟩

lemma *O-Cons-unfold*: $O\ (trn\ \#\ tr) = (if\ \gamma\ trn\ then\ g\ trn\ \#\ O\ tr\ else\ O\ tr)$
⟨*proof*⟩

lemma *V-append*: $V\ (tr\ @\ tr1) = V\ tr\ @\ V\ tr1$
⟨*proof*⟩

lemma *V-snoc*:
$\neg\ \varphi\ trn \Longrightarrow V\ (tr\ \#\#\ trn) = V\ tr\ \ \varphi\ trn \Longrightarrow V\ (tr\ \#\#\ trn) = V\ tr\ \#\#\ f\ trn$
⟨*proof*⟩

lemma *O-snoc*:
$\neg\ \gamma\ trn \Longrightarrow O\ (tr\ \#\#\ trn) = O\ tr\ \ \gamma\ trn \Longrightarrow O\ (tr\ \#\#\ trn) = O\ tr\ \#\#\ g\ trn$
⟨*proof*⟩

lemma *V-Nil-list-ex*: $V\ tr = [] \longleftrightarrow \neg\ list\text{-}ex\ \varphi\ tr$
⟨*proof*⟩

lemma *V-Nil-never*: $V\ tr = [] \longleftrightarrow never\ \varphi\ tr$
⟨*proof*⟩

lemma *Nil-V-never*: $[] = V\ tr \longleftrightarrow never\ \varphi\ tr$
⟨*proof*⟩

lemma *list-ex-iff-length-V*:
$list\text{-}ex\ \varphi\ tr \longleftrightarrow length\ (V\ tr) > 0$
⟨*proof*⟩

lemma *length-V*: $length\ (V\ tr) \leq length\ tr$
⟨*proof*⟩

lemma *V-list-all*: $V\ tr = map\ f\ tr \longleftrightarrow list\text{-}all\ \varphi\ tr$
⟨*proof*⟩

lemma *V-eq-Cons*:
assumes $V\ tr = v\ \#\ vl1$
shows $\exists\ trn\ tr2\ tr1.\ tr = tr2\ @\ [trn]\ @\ tr1 \land never\ \varphi\ tr2 \land \varphi\ trn \land f\ trn = v \land V\ tr1 = vl1$
⟨*proof*⟩

lemma *V-eq-append*:
assumes $V\ tr = vl1\ @\ vl2$
shows $\exists\ tr1\ tr2.\ tr = tr1\ @\ tr2 \land V\ tr1 = vl1 \land V\ tr2 = vl2$

⟨*proof*⟩

lemma *V-eq-RCons*:
assumes $V\ tr = vl1\ \#\#\ v$
shows $\exists\ trn\ tr1\ tr2.\ tr = tr1\ @\ [trn]\ @\ tr2 \wedge \varphi\ trn \wedge f\ trn = v \wedge V\ tr1 = vl1 \wedge never\ \varphi\ tr2$
⟨*proof*⟩

lemma *V-eq-Cons-RCons*:
assumes $V\ tr = v\ \#\ vl1\ \#\#\ w$
shows $\exists\ trv\ trnv\ tr1\ trnw\ trw.$
$tr = trv\ @\ [trnv]\ @\ tr1\ @\ [trnw]\ @\ trw\ \wedge$
$never\ \varphi\ trv \wedge \varphi\ trnv \wedge f\ trnv = v \wedge V\ tr1 = vl1 \wedge \varphi\ trnw \wedge f\ trnw = w \wedge never\ \varphi\ trw$
⟨*proof*⟩

lemma *O-append*: $O\ (tr\ @\ tr1) = O\ tr\ @\ O\ tr1$
⟨*proof*⟩

lemma *O-Nil-list-ex*: $O\ tr = [] \longleftrightarrow \neg\ list\text{-}ex\ \gamma\ tr$
⟨*proof*⟩

lemma *O-Nil-never*: $O\ tr = [] \longleftrightarrow never\ \gamma\ tr$
⟨*proof*⟩

lemma *Nil-O-never*: $[] = O\ tr \longleftrightarrow never\ \gamma\ tr$
⟨*proof*⟩

lemma *length-O*: $length\ (O\ tr) \leq length\ tr$
⟨*proof*⟩

lemma *O-list-all*: $O\ tr = map\ g\ tr \longleftrightarrow list\text{-}all\ \gamma\ tr$
⟨*proof*⟩

lemma *O-eq-Cons*:
assumes $O\ tr = obs\ \#\ obsl1$
shows $\exists\ trn\ tr2\ tr1.\ tr = tr2\ @\ [trn]\ @\ tr1 \wedge never\ \gamma\ tr2 \wedge \gamma\ trn \wedge g\ trn = obs \wedge O\ tr1 = obsl1$
⟨*proof*⟩

lemma *O-eq-append*:
assumes $O\ tr = obsl1\ @\ obsl2$
shows $\exists\ tr1\ tr2.\ tr = tr1\ @\ tr2 \wedge O\ tr1 = obsl1 \wedge O\ tr2 = obsl2$
⟨*proof*⟩

lemma *O-eq-RCons*:
assumes $O\ tr = oul1\ \#\#\ ou$
shows $\exists\ trn\ tr1\ tr2.\ tr = tr1\ @\ [trn]\ @\ tr2 \wedge \gamma\ trn \wedge g\ trn = ou \wedge O\ tr1 = oul1 \wedge never\ \gamma\ tr2$
⟨*proof*⟩

lemma *O-eq-Cons-RCons*:
assumes $O\ tr0 = ou\ \#\ oul1\ \#\#\ ouu$

shows $\exists$ *tr trn tr1 trnn trr*.
  $tr0 = tr$ @ $[trn]$ @ $tr1$ @ $[trnn]$ @ $trr \wedge$
  $never\ \gamma\ tr \wedge \gamma\ trn \wedge g\ trn = ou \wedge O\ tr1 = oul1 \wedge \gamma\ trnn \wedge g\ trnn = ouu \wedge never\ \gamma\ trr$
⟨*proof*⟩

lemma *O-eq-Cons-RCons-append*:
assumes $O\ tr0 = ou$ # $oul1$ ## ouu @ $oull$
shows $\exists$ *tr trn tr1 trnn trr*.
  $tr0 = tr$ @ $[trn]$ @ $tr1$ @ $[trnn]$ @ $trr \wedge$
  $never\ \gamma\ tr \wedge \gamma\ trn \wedge g\ trn = ou \wedge O\ tr1 = oul1 \wedge \gamma\ trnn \wedge g\ trnn = ouu \wedge O\ trr = oull$
⟨*proof*⟩

lemma *O-Nil-tr-Nil*: $O\ tr \neq [] \Longrightarrow tr \neq []$
⟨*proof*⟩

lemma *V-Cons-eq-append*: $V\ (trn$ # $tr) = V\ [trn]$ @ $V\ tr$
⟨*proof*⟩

lemma *set-V*: $set\ (V\ tr) \subseteq \{f\ trn \mid trn\ .\ trn \in\in tr \wedge \varphi\ trn\}$
⟨*proof*⟩

lemma *set-O*: $set\ (O\ tr) \subseteq \{g\ trn \mid trn\ .\ trn \in\in tr \wedge \gamma\ trn\}$
⟨*proof*⟩

lemma *list-ex-length-O*:
assumes $list\text{-}ex\ \gamma\ tr$ **shows** $length\ (O\ tr) > 0$
⟨*proof*⟩

lemma *list-ex-iff-length-O*:
$list\text{-}ex\ \gamma\ tr \longleftrightarrow length\ (O\ tr) > 0$
⟨*proof*⟩

lemma *length1-O-list-ex-iff*:
$length\ (O\ tr) > 1 \Longrightarrow list\text{-}ex\ \gamma\ tr$
⟨*proof*⟩

lemma *list-all-O-map*: $list\text{-}all\ \gamma\ tr \Longrightarrow O\ tr = map\ g\ tr$
⟨*proof*⟩

lemma *never-O-Nil*: $never\ \gamma\ tr \Longrightarrow O\ tr = []$
⟨*proof*⟩

lemma *list-all-V-map*: $list\text{-}all\ \varphi\ tr \Longrightarrow V\ tr = map\ f\ tr$
⟨*proof*⟩

lemma *never-V-Nil*: $never\ \varphi\ tr \Longrightarrow V\ tr = []$
⟨*proof*⟩

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

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

lemma *V-iff-non-φ*[*simp*]: *V* (*trn* # *tr*) = *V tr* ⟷ ¬ *φ trn*
⟨*proof*⟩

lemma *V-imp-φ*: *V* (*trn* # *tr*) = *v* # *V tr* ⟹ *φ trn*
⟨*proof*⟩

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

lemma *V-iff-Nil*[*simp*]: *V* (*trn* # *tr*) = [] ⟷ ¬ *φ trn* ∧ *V tr* = []
⟨*proof*⟩

end

3.3 Instantiation for IO automata

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

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

locale *BD-Security-IO* = *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

sublocale *BD-Security-TS* **where** *validTrans* = *validTrans* **and** *srcOf* = *srcOf* **and** *tgtOf* = *tgtOf* ⟨*proof*⟩

lemma *reachNT-step-induct*[*consumes 1*, *case-names Istate Step*]:

assumes *reachNT s*
and *P istate*
and $\bigwedge$*s a ou s′*. *reachNT s* $\Longrightarrow$ *step s a* = (*ou*, *s′*) $\Longrightarrow$ ¬*T* (*Trans s a ou s′*) $\Longrightarrow$ *P s* $\Longrightarrow$ *P s′*
shows *P s*
⟨*proof*⟩

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-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*) ¬*T* (*Trans sh a ou s*)
⟨*proof*⟩

definition *invarNT* **where**
invarNT Inv ≡ ∀ *s a ou s′*. *reachNT s* ∧ *Inv s* ∧ ¬ *T* (*Trans s a ou s′*) ∧ *step s a* = (*ou*,*s′*) ⟶ *Inv s′*

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

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

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

end

3.4 Trigger-preserving BD security

Section 3.3 of [3] gives a recipe for incorporating declassification triggers into the bound, and discusses the question whether this is always possible without loss of generality, giving a partially positive answer: the transformed security property is equivalent to a slightly strengthened version of the original one.

3.4.1 Definition

context *Abstract-BD-Security*

begin

The strengthened variant of BD Security is called *trigger-preserving* in [3], because the difference to regular BD Security is that the (non-firing of the) declassification trigger in the original trace is preserved in alternative traces.

definition *secureTT* :: *bool* **where**
secureTT ≡
∀ *tr vl vl1*.
  validSystemTrace tr ∧ *TT tr* ∧ *B vl vl1* ∧ *V tr* = *vl* ⟶
  (∃ *tr1*. *validSystemTrace tr1* ∧ *TT tr1* ∧ *O tr1* = *O tr* ∧ *V tr1* = *vl1*)

This indeed strengthens the original notion of BD Security.

lemma *secureTT-secure*: *secureTT* ⟹ *secure*
⟨*proof*⟩

lemma *secureTT-E*:
assumes *secureTT*
and *validSystemTrace tr* **and** *TT tr* **and** *B vl vl1* **and** *V tr* = *vl*
obtains *tr1* **where** *validSystemTrace tr1* **and** *TT tr1* **and** *O tr1* = *O tr* **and** *V tr1* = *vl1*
⟨*proof*⟩

lemma *secure-E*:
assumes *secure*
and *validSystemTrace tr* **and** *TT tr* **and** *B vl vl1* **and** *V tr* = *vl*
obtains *tr1* **where** *validSystemTrace tr1* **and** *O tr1* = *O tr* **and** *V tr1* = *vl1*
⟨*proof*⟩

end

3.4.2 Incorporating static triggers into the bound

By making transitions that fire the trigger emit a dedicated secret value (here *None*), the (non-firing of the) trigger can be incorporated into the bound.

locale *BD-Security-TS-Triggerless* = *Orig*: *BD-Security-TS*
begin

abbreviation φ' *trn* ≡ φ *trn* ∨ *T trn*

abbreviation f' *trn* ≡ (*if T trn then None else Some* (*f trn*))

abbreviation T' *trn* ≡ *False*

abbreviation B' vl' $vl1'$ ≡ *B* (*these* vl') (*these* $vl1'$) ∧ *never Option.is-none* vl' ∧ *never Option.is-none* $vl1'$

sublocale *Prime?*: *BD-Security-TS* **where** $\varphi = \varphi'$ **and** $f = f'$ **and** $T = T'$ **and** $B = B'$ ⟨*proof*⟩

lemma *map-Some-these*: *never Option.is-none xs* ⟹ *map Some* (*these xs*) = *xs*
⟨*proof*⟩

lemma *V′-never-none-T*[*simp*]: *Prime.V tr* = *vl* ⟹ *never Option.is-none vl* ⟷ *never T tr*
⟨*proof*⟩

lemma *V′-V*: *never T tr* ⟷ *Prime.V tr* = *map Some* (*Orig.V tr*)
⟨*proof*⟩

lemma *V-Some-never-T*: *Prime.V tr* = *map Some vl* ⟹ *never T tr*
⟨*proof*⟩

In the modified setup, the notions of trigger-preserving and original BD Security coincide due to the trigger being vacuously false.

lemma *secureTT-iff-secure*: *Prime.secureTT* ⟷ *Prime.secure*
⟨*proof*⟩

The modified property is equivalent to trigger-preserving BD Security in the original setup [3, Proposition 2].

lemma *secureTT-iff-secure′*: *Orig.secureTT* ⟷ *Prime.secure*
⟨*proof*⟩

The modified property also strengthens the regular notion of BD Security in the original setup [3, Proposition 1].

lemma *secure′-secure*: *Prime.secure* ⟹ *Orig.secure*
⟨*proof*⟩

end

3.4.3 Reflexive-transitive closure of declassification bounds

Another property of trigger-preserving BD Security is that security w.r.t. an arbitrary bound B is equivalent to security w.r.t. its reflexive-transitive closure B^{**} [3, Proposition 3].

locale *Abstract-BD-Security-Transitive-Closure* = *Orig*: *Abstract-BD-Security*
begin

sublocale *Prime?*: *Abstract-BD-Security* **where** $B = B^{**}$ ⟨*proof*⟩

lemma *secureTT-iff-secureTT′*: *Orig.secureTT* ⟷ *Prime.secureTT*
⟨*proof*⟩

end

4 Unwinding proof method

This section formalizes the unwinding proof method for BD Security discussed in [4, Section 5.1]

context *BD-Security-IO*

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*

definition *consumeList* :: ('*state*,'*act*,'*out*) *trans trace* ⇒ '*value list* ⇒ '*value list* ⇒ *bool* **where**
consumeList tr vl vl' ≡ *vl* = (*V tr*) @ *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* ≡
∃ *al1 vl1'*.
let tr1 = *traceOf s1 al1*; *s1'* = *tgtOf* (*last tr1*) *in*
list-ex φ *tr1* ∧ *consumeList tr1 vl1 vl1'* ∧
never γ *tr1*
∧
Δ *s vl s1' vl1'*

lemma *iactionI-ms*[*intro?*]:
assumes *s*: *sstep s1 al1* = (*oul1*, *s1'*)
and *l*: *list-ex* φ (*traceOf s1 al1*)
and *consumeList* (*traceOf s1 al1*) *vl1 vl1'*
and *never* γ (*traceOf s1 al1*) **and** Δ *s vl s1' vl1'*
shows *iaction* Δ *s vl s1 vl1*
⟨*proof*⟩

lemma *sstep-eq-singleiff*[*simp*]: *sstep s1* [*a1*] = ([*ou1*], *s1'*) ⟷ *step s1 a1* = (*ou1*, *s1'*)
⟨*proof*⟩

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'* ≡
∃ *al1 vl1'*.
let trn = *Trans s a ou s'*; *tr1* = *traceOf s1 al1*; *s1'* = *tgtOf* (*last tr1*) *in*
al1 ≠ [] ∧ *consumeList tr1 vl1 vl1'* ∧
O tr1 = *O* [*trn*] ∧
Δ *s' vl' s1' vl1'*

lemma *matchI-ms*[*intro?*]:
assumes *s*: *sstep s1 al1* = (*oul1*, *s1'*)
and *l*: *al1* ≠ []
and *consumeList* (*traceOf s1 al1*) *vl1 vl1'*
and *O* (*traceOf s1 al1*) = *O* [*Trans s a ou s'*]
and Δ *s' vl' s1' vl1'*
shows *match* Δ *s s1 vl1 a ou s' vl'*
⟨*proof*⟩

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'*) ⟹ *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'* ≡
¬ γ (*Trans s a ou s'*) ∧
Δ *s' vl' s1 vl1*

lemma *ignoreI*[*intro?*]:
assumes ¬ 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* ≡
∀ *a ou s' vl'*.
let trn = *Trans s a ou s' in*
validTrans trn ∧ ¬ *T trn* ∧
consume trn vl vl'
⟶
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′*); ¬ *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* ⇒ *′value* ⇒ *bool* **where**
exit s v ≡ ∀ *tr trn*. *validFrom s* (*tr* ## *trn*) ∧ *never T* (*tr* ## *trn*) ∧ φ *trn* ⟶ *f trn* ≠ *v*

lemma *exit-coind*:
assumes *K*: *K s*
and *I*: $\bigwedge$ *trn*. ⟦*K* (*srcOf trn*); *validTrans trn*; ¬ *T trn*⟧
$\Longrightarrow$ (φ *trn* ⟶ *f trn* ≠ *v*) ∧ *K* (*tgtOf trn*)
shows *exit s v*
⟨*proof*⟩

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*
⟨*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* $\Longrightarrow$ *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* $\wedge$ Δ *s vl s1 vl1*
  $\longrightarrow$
  (*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*)

lemma *unwindI*[*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 Δ *s vl s1 vl1*
  $\vee$
  ((*vl* $\neq$ [] $\vee$ *vl1* = []) $\wedge$ *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 $\exists$ *tr1*. *validFrom s1 tr1* $\wedge$ *O tr1* = *O tr* $\wedge$ *V tr1* = *vl1*
⟨*proof*⟩

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

end

5 Compositional Reasoning

This section formalizes the compositional unwinding method discussed in [4, Section 5.2]

context *BD-Security-IO* **begin**

5.1 Preliminaries

definition *disjAll* Δs *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* Δ Δs) $\equiv \lambda s$ *vl s1 vl1*. Δ *s vl s1 vl1* $\vee$ *disjAll* Δs *s vl s1 vl1*
⟨*proof*⟩

lemma *disjAll-mono*:
assumes *disjAll* Δs *s vl s1 vl1*
and $\Delta s \subseteq \Delta s'$
shows *disjAll* $\Delta s'$ *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* $\Longrightarrow \Delta'$ *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* $\Longrightarrow \Delta'$ *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*: $\bigwedge$ *s vl s1 vl1*. Δ *s vl s1 vl1* $\Longrightarrow \Delta'$ *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* $\Longrightarrow \Delta'$ *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 Δs $\equiv$
$\forall$ *s vl s1 vl1*.
reachNT s $\wedge$ *reach s1* $\wedge$ Δ *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*.
  $[\![$*reachNT s*; *reach s1*; Δ *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*
shows *unwind-to* Δ Δs
⟨*proof*⟩

lemma *unwind-dec*:
assumes *ne*: $\bigwedge \Delta.\ \Delta \in \Delta s \Longrightarrow$ *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* $\Longrightarrow$ $\Delta 0$ *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* $\Longrightarrow$ $\Delta 0$ *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
$\langle proof \rangle$

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
$\langle proof \rangle$

lemma *unwind-cont-mono*:
assumes Δs: *unwind-cont* Δ Δs
and $\Delta s'$: $\Delta s \subseteq \Delta s'$
shows *unwind-cont* Δ $\Delta s'$
$\langle proof \rangle$

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 $\exists$ $\Delta 2$. (Δ,$\Delta 2$) $\in$ *allConsec* Δs
$\langle proof \rangle$

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

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* Δs $\Longrightarrow$ *unwind-cont* $\Delta 1$ {$\Delta 1$, $\Delta 2$, Δe}
and *l*: *unwind-cont* (*last* Δs) {*last* Δs, Δe}
and *e*: *unwind-exit* Δe
shows *secure*

⟨*proof*⟩

5.4 Instances

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

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

corollary *unwind-decomp5-secure*:
assumes
i: $\bigwedge$ *vl vl1*. *B vl vl1* $\Longrightarrow$ Δ*1 istate vl istate vl1*
and *c1*: *unwind-cont* Δ*1* {Δ*1*, Δ*2*, Δ*e*}
and *c2*: *unwind-cont* Δ*2* {Δ*2*, Δ*3*, Δ*e*}
and *c3*: *unwind-cont* Δ*3* {Δ*3*, Δ*4*, Δ*e*}
and *c4*: *unwind-cont* Δ*4* {Δ*4*, Δ*5*, Δ*e*}
and *l*: *unwind-cont* Δ*5* {Δ*5*, Δ*e*}
and *e*: *unwind-exit* Δ*e*
shows *secure*
⟨*proof*⟩

5.5 A graph alternative presentation

theorem *unwind-decomp-secure-graph*:
assumes *n*: $\forall$ Δ $\in$ *Domain Gr*. $\exists$ Δ*s*. Δ*s* $\subseteq$ *Domain Gr* $\wedge$ (Δ,Δ*s*) $\in$ *Gr*
and *i*: Δ*0* $\in$ *Domain Gr* $\bigwedge$ *vl vl1*. *B vl vl1* $\Longrightarrow$ Δ*0 istate vl istate vl1*
and *c*: $\bigwedge$ Δ. *unwind-exit* Δ $\vee$ ($\forall$ Δ*s*. (Δ,Δ*s*) $\in$ *Gr* $\longrightarrow$ *unwind-cont* Δ Δ*s*)
shows *secure*
⟨*proof*⟩

References

[1] T. Bauereiß, A. Pesenti Gritti, A. Popescu, and F. Raimondi. Cosmed: A confidentiality-verified social media platform. In J. C. Blanchette and S. Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2016.

[2] T. Bauereiß, A. Pesenti Gritti, A. Popescu, and F. Raimondi. Cosmedis: A distributed social media platform with formally verified confidentiality guarantees. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 729–748. IEEE Computer Society, 2017.

[3] T. Bauereiß, A. Pesenti Gritti, A. Popescu, and F. Raimondi. Cosmed: A confidentiality-verified social media platform. *J. Autom. Reason.*, 61(1-4):113–139, 2018.

[4] S. Kanav, P. Lammich, and A. Popescu. A conference management system with verified document confidentiality. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 2014.

[5] A. Popescu, T. Bauereiss, and P. Lammich. Bounded-Deducibility security (invited paper). In L. Cohen and C. Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 3:1–3:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[6] A. Popescu, P. Lammich, and P. Hou. Cocon: A conference management system with formally verified document confidentiality. *J. Autom. Reason.*, 65(2):321–356, 2021.

[7] D. Sutherland. A model of information. In *9th National Security Conference*, pages 175–183, 1986.