

Bounded-Deducibility Security

Andrei Popescu Peter Lammich Thomas Bauereiss

Contents

1	Introduction	1
2	Preliminaries	2
2.1	Transition Systems	4
2.1.1	Traces	4
2.1.2	Reachability	6
2.2	IO automata	8
2.2.1	IO automata as transition systems	8
2.2.2	State invariants	9
2.2.3	Traces of actions	10
3	BD Security	13
3.1	Abstract definition	13
3.2	Instantiation for transition systems	14
3.3	Instantiation for IO automata	18
3.4	Trigger-preserving BD security	19
3.4.1	Definition	19
3.4.2	Incorporating static triggers into the bound	20
3.4.3	Reflexive-transitive closure of declassification bounds	21
4	Unwinding proof method	21
5	Compositional Reasoning	26
5.1	Preliminaries	27
5.2	Decomposition into an arbitrary network of components	27
5.3	A customization for linear modular reasoning	28
5.4	Instances	30
5.5	A graph alternative presentation	30

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

lemma *filtermap-eq-append*:

assumes *filtermap pred func tr = al1 @ al2*

shows $\exists tr1\ tr2. tr = tr1 @ tr2 \wedge filtermap\ pred\ func\ tr1 = al1 \wedge filtermap\ pred\ func\ tr2 = al2$

<proof>

lemma *holds-filtermap-RCons[simp]*:

pred trn $\implies filtermap\ pred\ func\ (tr\ ##\ trn) = filtermap\ pred\ func\ tr\ ##\ func\ trn$

<proof>

lemma *not-holds-filtermap-RCons[simp]*:

$\neg pred\ trn \implies filtermap\ pred\ func\ (tr\ ##\ trn) = filtermap\ pred\ func\ tr$

<proof>

lemma *filtermap-eq-RCons*:

assumes *filtermap pred func tr = al1 ## a*

shows $\exists trn\ tr1\ tr2.$

$tr = tr1 @ [trn] @ tr2 \wedge never\ pred\ tr2 \wedge pred\ trn \wedge func\ trn = a \wedge filtermap\ pred\ func\ tr1 = al1$

<proof>

lemma *filtermap-eq-Cons-RCons*:

assumes *filtermap pred func tr = a # al1 ## b*

shows $\exists tra\ trna\ tr1\ trnb\ trb.$

$tr = tra @ [trna] @ tr1 @ [trnb] @ trb \wedge$

$never\ pred\ tra \wedge$

$pred\ trna \wedge func\ trna = a \wedge$

$filtermap\ pred\ func\ tr1 = al1 \wedge$

$pred\ trnb \wedge func\ trnb = b \wedge$

$never\ pred\ trb$

<proof>

lemma *filter-Nil-never*: $[] = filter\ pred\ xs \implies never\ pred\ xs$

<proof>

lemma *never-Nil-filter*: $never\ pred\ xs \iff [] = filter\ pred\ xs$

<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*:

$filtermap\ pred\ func\ [x, x'] = filtermap\ pred\ func\ [y, y']$

$\implies filtermap\ pred\ func\ (x\ #\ x'\ #\ zs) = filtermap\ pred\ func\ (y\ #\ y'\ #\ zs)$

<proof>

lemma *filtermap-Cons-cong*:

$filtermap\ pred\ func\ xs = filtermap\ pred\ func\ ys$

$\implies \text{filtermap pred func } (x \# xs) = \text{filtermap pred func } (x \# ys)$
 <proof>

lemma *set-filtermap*:
 $\text{set } (\text{filtermap pred func } xs) \subseteq \{\text{func } x \mid x . x \in xs \wedge \text{pred } x\}$
 <proof>

2.1 Transition Systems

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

2.1.1 Traces

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

locale *Transition-System* =
fixes *istate* :: *'state*
and *validTrans* :: *'trans* \Rightarrow *bool*
and *srcOf* :: *'trans* \Rightarrow *'state*
and *tgtOf* :: *'trans* \Rightarrow *'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]*:
 $tr \neq [] \implies \text{srcOfTrFrom } s \text{ } tr = \text{srcOfTr } tr$
 <proof>

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

lemma *tgtOfTrFrom-tgtOfTr[simp]*:
 $tr \neq [] \implies \text{tgtOfTrFrom } s \text{ } tr = \text{tgtOfTr } tr$
 <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* \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-SingleE*[*elim!*]: $valid\ [trn]$
inductive-cases *valid-ConsE*[*elim*]: $valid\ (trn\ \#\ tr)$

inductive *valid2* :: $'trans\ trace \Rightarrow bool$ **where**
Singl[*simp, intro!*]:
 $validTrans\ trn$
 \implies
 $valid2\ [trn]$
 $|$
Rcons[*intro*]:
 $\llbracket valid2\ tr; tgtOf\ (last\ tr) = srcOf\ trn; validTrans\ trn \rrbracket$
 \implies
 $valid2\ (tr\ \#\#\ trn)$

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

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

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

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

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

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

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) \longleftrightarrow validTrans trn \wedge ((tgtOf trn = srcOf (hd tr) \wedge valid tr) \vee tr = [])

<proof>

lemma *valid-append*:

tr \neq [] \implies tr1 \neq [] \implies

valid (tr @ tr1) \longleftrightarrow valid tr \wedge valid tr1 \wedge tgtOf (last tr) = srcOf (hd tr1)

<proof>

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

definition *validFrom* :: *'state \Rightarrow 'trans trace \Rightarrow bool* **where**

validFrom s tr \equiv tr = [] \vee (valid tr \wedge srcOf (hd tr) = s)

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

<proof>

lemma *validFrom-valid[simp,intro]*: *valid tr \wedge srcOf (hd tr) = s \implies validFrom s tr*

<proof>

lemma *validFrom-append*:

validFrom s (tr @ tr1) \longleftrightarrow (tr = [] \wedge validFrom s tr1) \vee (tr \neq [] \wedge validFrom s tr \wedge validFrom (tgtOf (last tr)) tr1)

<proof>

lemma *validFrom-Cons*:

validFrom s (trn # tr) \longleftrightarrow validTrans trn \wedge srcOf trn = s \wedge validFrom (tgtOf trn) tr

<proof>

2.1.2 Reachability

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

Istate: reach istate

|

Step: reach s \implies validTrans trn \implies srcOf trn = s \implies tgtOf trn = s' \implies 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
 $s = \text{istate} \vee (\exists \text{tr}. \text{valid tr} \wedge \text{srcOf (hd tr)} = \text{istate} \wedge \text{tgtOf (last tr)} = s)$
<proof>

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

inductive *reachFrom* :: *'state* \Rightarrow *'state* \Rightarrow *bool*
for $s :: 'state$
where
RefI[intro]: reachFrom s s
| Step: $\llbracket \text{reachFrom } s \ s'; \text{ validTrans trn}; \text{ srcOf trn} = s'; \text{ tgtOf trn} = s'' \rrbracket \Longrightarrow \text{reachFrom } s \ s''$

lemma *reachFrom-Step1*:
 $\llbracket \text{validTrans trn}; \text{ srcOf trn} = s; \text{ tgtOf trn} = s' \rrbracket \Longrightarrow \text{reachFrom } s \ s'$
<proof>

lemma *reachFrom-Step-Left*:
 $\text{reachFrom } s' \ s'' \Longrightarrow \text{validTrans trn} \Longrightarrow \text{srcOf trn} = s \Longrightarrow \text{tgtOf trn} = s' \Longrightarrow \text{reachFrom } s \ s''$
<proof>

lemma *reachFrom-trans*: $\text{reachFrom } s0 \ s1 \Longrightarrow \text{reachFrom } s1 \ s2 \Longrightarrow \text{reachFrom } s0 \ s2$
<proof>

lemma *reachFrom-reach*: $\text{reachFrom } s \ s' \Longrightarrow \text{reach } s \Longrightarrow \text{reach } s'$
<proof>

lemma *valid-validTrans-set*:
assumes *valid tr and trn \in tr*
shows *validTrans trn*
<proof>

lemma *validFrom-validTrans-set*:
assumes *validFrom s tr* **and** *trn ∈ 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 ⇒ reach s ⇒ tr ≠ [] ⇒ 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 ⇒ 'act ⇒ 'out * 'state*

begin

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)*

fun *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))
 <proof>

sublocale *Transition-System*
where *istate* = *istate* **and** *validTrans* = *validTrans* **and** *srcOf* = *srcOf* **and** *tgtOf* = *tgtOf* <proof>

lemma *reach-step*:
reach s \Longrightarrow *reach* (snd (step s a))
 <proof>

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

lemma *reach-step-induct*[*consumes* 1, *case-names* *Istate Step*]:
assumes *s*: *reach* s
and *istate*: *P* *istate*
and *step*: \bigwedge s a. *reach* s \Longrightarrow *P* s \Longrightarrow *P* (snd (step s a))
shows *P* s
 <proof>

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

lemma *valid-filter-no-state-change*:
valid tr \Longrightarrow (\bigwedge trn. trn $\in\in$ tr \Longrightarrow \neg (PP trn) \Longrightarrow *srcOf* trn = *tgtOf* trn) \Longrightarrow
 \exists trn. trn $\in\in$ tr \wedge PP trn \Longrightarrow *valid* (filter PP tr) \wedge *srcOfTr* tr = *srcOfTr* (filter PP tr)
 \wedge *tgtOfTr* tr = *tgtOfTr* (filter PP 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>

2.2.2 State invariants

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

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 $\varphi\ s$
 $\langle proof \rangle$

2.2.3 Traces of actions

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

fun $sstep :: 'state \Rightarrow 'act\ list \Rightarrow 'out\ list \times 'state$ **where**
 $sstep\ s\ [] = ([], s)$
 $|$
 $sstep\ s\ (a\ \# \ al) = (case\ step\ s\ a\ of\ (ou, s') \Rightarrow (case\ sstep\ s'\ al\ of\ (oul, s'') \Rightarrow (ou\ \# \ oul, s'')))$

lemma $length-traceOf[simp]$:
 $length\ (traceOf\ s\ al) = length\ al$
 $\langle proof \rangle$

lemma $traceOf-Nil[simp]$:
 $traceOf\ s\ al = [] \longleftrightarrow al = []$
 $\langle proof \rangle$

lemma $sstep-outOf-traceOf[simp]$:
 $sstep\ s\ al = (ou, s') \Longrightarrow map\ outOf\ (traceOf\ s\ al) = ou$
 $\langle proof \rangle$

lemma $sstep-tgtOf-traceOf[simp]$:
 $al \neq [] \Longrightarrow sstep\ s\ al = (ou, s') \Longrightarrow tgtOf\ (last\ (traceOf\ s\ al)) = s'$
 $\langle proof \rangle$

lemma $srcOf-traceOf[simp]$:
 $al \neq [] \Longrightarrow srcOf\ (hd\ (traceOf\ s\ al)) = s$
 $\langle proof \rangle$

lemma $actOf-traceOf[simp]$:
 $map\ actOf\ (traceOf\ s\ al) = al$
 $\langle proof \rangle$

lemma *traceOf-append*:

$al \neq [] \implies s1 = \text{tgtOf } (\text{last } (\text{traceOf } s \text{ } al)) \implies$
 $\text{traceOf } s \text{ } (al @ al1) = \text{traceOf } s \text{ } al @ \text{traceOf } s1 \text{ } al1$
(proof)

lemma *sstep-append*:

assumes $\text{sstep } s \text{ } al = (oul, s1)$ **and** $\text{sstep } s1 \text{ } al1 = (oul1, s2)$
shows $\text{sstep } s \text{ } (al @ al1) = (oul @ oul1, s2)$
(proof)

lemma *reach-sstep*:

assumes $\text{reach } s$ **and** $\text{sstep } s \text{ } al = (ou, s1)$
shows $\text{reach } s1$
(proof)

lemma *traceOf-consR[simp]*:

assumes $al \neq []$ **and** $s1 = \text{tgtOf } (\text{last } (\text{traceOf } s \text{ } al))$ **and** $\text{step } s1 \text{ } a = (ou, s2)$
shows $\text{traceOf } s \text{ } (al \## a) = \text{traceOf } s \text{ } al \## \text{Trans } s1 \text{ } a \text{ } ou \text{ } s2$
(proof)

lemma *sstep-consR[simp]*:

assumes $\text{sstep } s \text{ } al = (oul, s1)$ **and** $\text{step } s1 \text{ } a = (ou, s2)$
shows $\text{sstep } s \text{ } (al \## a) = (oul \## ou, s2)$
(proof)

lemma *fst-sstep-consR*:

$\text{fst } (\text{sstep } s \text{ } (al \## a)) = \text{fst } (\text{sstep } s \text{ } al) \## (\text{fst } (\text{step } (snd (\text{sstep } s \text{ } al)) \text{ } a))$
(proof)

lemma *valid-traceOf[simp]*: $al \neq [] \implies \text{valid } (\text{traceOf } s \text{ } al)$

(proof)

lemma *validFrom-traceOf[simp]*: $\text{validFrom } s \text{ } (\text{traceOf } s \text{ } al)$

(proof)

lemma *validFrom-traceOf2*:

assumes $\text{validFrom } s \text{ } tr$

shows $tr = \text{traceOf } s \text{ } (\text{map } \text{actOf } tr)$

(proof)

lemma *set-traceOf-validTrans*:

assumes $trn \in \text{traceOf } s \text{ } al$ **shows** $\text{validTrans } trn$

(proof)

lemma *traceOf-append-sstep*: $\text{traceOf } s \text{ } (al @ al1) = \text{traceOf } s \text{ } al @ \text{traceOf } (snd (\text{sstep } s \text{ } al)) \text{ } al1$

(proof)

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

lemma *snd-sstep-step-constant*:
assumes $\forall a. a \in \in al \longrightarrow \text{snd} (\text{step } s a) = s$
shows $\text{snd} (\text{sstep } s al) = s$
 ⟨proof⟩

definition *const-tr* $tr \equiv \forall trn. trn \in \in tr \longrightarrow \text{srcOf } trn = \text{tgtOf } trn$

lemma *const-tr-same-src-tgt*:
assumes *valid tr const-tr tr*
shows $\text{srcOfTr } tr = \text{tgtOfTr } tr$
 ⟨proof⟩

lemma *traceOf-snoc*:
 $\text{traceOf } s (al \#\# a) =$
 $\text{traceOf } s al \#\#$
 $\text{Trans} (\text{snd} (\text{sstep } s al))$
 $\quad a$
 $\quad (\text{fst} (\text{step} (\text{snd} (\text{sstep } s al)) a))$
 $\quad (\text{snd} (\text{step} (\text{snd} (\text{sstep } s al)) a))$
 ⟨proof⟩

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

abbreviation *transOf* $s a \equiv \text{Trans } s a (\text{fst} (\text{step } s a)) (\text{snd} (\text{step } s a))$

lemma *traceOf-Cons*: $\text{traceOf } s (a \# al) = \text{transOf } s a \# \text{traceOf} (\text{snd} (\text{step } s a)) al$
 ⟨proof⟩

definition *commute* $s a1 a2$
 $\equiv \text{snd} (\text{sstep } s [a1, a2]) = \text{snd} (\text{sstep } s [a2, a1])$

definition *absorb* $:: 'state \Rightarrow 'act \Rightarrow 'act \Rightarrow \text{bool}$ **where**
 $\text{absorb } s a1 a2 \equiv \text{snd} (\text{sstep } s [a1, a2]) = \text{snd} (\text{step } s a2)$

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 $validFrom\ s0\ (tr1\ @\ transOf\ s\ a'\ \# \ tr2)$
 $\langle proof \rangle$

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

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

lemma $sstep\ Cons$:
 $sstep\ s\ (a\ \# \ al) = (fst\ (step\ s\ a)\ \# \ fst\ (sstep\ (snd\ (step\ s\ a))\ al),\ snd\ (sstep\ (snd\ (step\ s\ a))\ al))$
 $\langle proof \rangle$
declare $sstep.simps(2)[simp\ del]$

lemma $length\ fst\ sstep$: $length\ (fst\ (sstep\ s\ al)) = length\ al$
 $\langle proof \rangle$

3 BD Security

3.1 Abstract definition

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

locale $Abstract\ BD\ Security =$
fixes
 $validSystemTrace :: 'traces \Rightarrow bool$
and — secret values:
 $V :: 'traces \Rightarrow 'values$
and — observations:
 $O :: 'traces \Rightarrow 'observations$
and — declassification bound:
 $B :: 'values \Rightarrow 'values \Rightarrow bool$
and — declassification trigger:
 $TT :: 'traces \Rightarrow bool$
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**
 $secure \equiv$

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

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* \Rightarrow *bool*

and *srcOf* :: *'trans* \Rightarrow *'state* **and** *tgtOf* :: *'trans* \Rightarrow *'state*

+

fixes

φ :: *'trans* \Rightarrow *bool* **and** f :: *'trans* \Rightarrow *'value*

and

γ :: *'trans* \Rightarrow *bool* **and** g :: *'trans* \Rightarrow *'obs*

and

T :: *'trans* \Rightarrow *bool*

and

B :: *'value list* \Rightarrow *'value list* \Rightarrow *bool*

begin

definition V :: *'trans list* \Rightarrow *'value list* **where** $V \equiv filtermap\ \varphi\ f$

definition O :: *'trans trace* \Rightarrow *'obs list* **where** $O \equiv filtermap\ \gamma\ g$

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*: $O\ tr = map\ g\ (filter\ \gamma\ tr)$ *<proof>*

lemma *V-map-filter*: $V\ tr = map\ f\ (filter\ \varphi\ tr)$ *<proof>*

lemma *V-simps[simp]*:

$V\ [] = [] \quad \neg\ \varphi\ trn \Longrightarrow V\ (trn\ \#\ tr) = V\ tr\ \varphi\ trn \Longrightarrow V\ (trn\ \#\ tr) = f\ trn\ \#\ V\ tr$

<proof>

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

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

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

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

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

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

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

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

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

lemma *V-eq-Cons*:

assumes $V \ tr = v \ \# \ vl1$

shows $\exists \ trn \ tr2 \ tr1. \ tr = tr2 \ @ \ [trn] \ @ \ tr1 \wedge never \ \varphi \ tr2 \wedge \varphi \ trn \wedge f \ trn = v \wedge V \ tr1 = vl1$

⟨proof⟩

lemma *V-eq-append*:

assumes $V \ tr = vl1 \ @ \ vl2$

shows $\exists \ tr1 \ tr2. \ tr = tr1 \ @ \ tr2 \wedge V \ tr1 = vl1 \wedge V \ tr2 = vl2$

<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:* $O\ tr = [] \iff \neg\ list\text{-}ex\ \gamma\ tr$

<proof>

lemma *O-Nil-never:* $O\ tr = [] \iff never\ \gamma\ tr$

<proof>

lemma *Nil-O-never:* $[] = O\ tr \iff never\ \gamma\ tr$

<proof>

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

<proof>

lemma *O-list-all:* $O\ tr = map\ g\ tr \iff list\text{-}all\ \gamma\ tr$

<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:* $O\ tr \neq [] \implies tr \neq []$

<proof>

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

<proof>

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

<proof>

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

<proof>

lemma *list-ex-length-O:*

assumes *list-ex* $\gamma\ tr$ **shows** $length\ (O\ tr) > 0$

<proof>

lemma *list-ex-iff-length-O:*

list-ex $\gamma\ tr \iff length\ (O\ tr) > 0$

<proof>

lemma *length1-O-list-ex-iff:*

$length\ (O\ tr) > 1 \implies list-ex\ \gamma\ tr$

<proof>

lemma *list-all-O-map:* $list-all\ \gamma\ tr \implies O\ tr = map\ g\ tr$

<proof>

lemma *never-O-Nil:* $never\ \gamma\ tr \implies O\ tr = []$

<proof>

lemma *list-all-V-map:* $list-all\ \varphi\ tr \implies V\ tr = map\ f\ tr$

<proof>

lemma *never-V-Nil:* $never\ \varphi\ tr \implies V\ tr = []$

<proof>

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

Istate: *reachNT* *istate*

|

Step:

$\llbracket \text{reachNT } (\text{srcOf } \text{trn}); \text{validTrans } \text{trn}; \neg T \text{trn} \rrbracket$

$\Longrightarrow \text{reachNT } (\text{tgtOf } \text{trn})$

lemma *reachNT-reach*: **assumes** *reachNT* *s* **shows** *reach* *s*

<proof>

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

<proof>

lemma *V-imp- φ* : $V (\text{trn} \# \text{tr}) = v \# V \text{tr} \Longrightarrow \varphi \text{trn}$

<proof>

lemma *V-imp-Nil*: $V (\text{trn} \# \text{tr}) = [] \Longrightarrow V \text{tr} = []$

<proof>

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

<proof>

end

3.3 Instantiation for IO automata

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

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

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

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

+

fixes

φ :: ('state,'act,'out) trans \Rightarrow bool **and** f :: ('state,'act,'out) trans \Rightarrow 'value

and

γ :: ('state,'act,'out) trans \Rightarrow bool **and** g :: ('state,'act,'out) trans \Rightarrow 'obs

and

T :: ('state,'act,'out) trans \Rightarrow bool

and

B :: 'value list \Rightarrow 'value list \Rightarrow 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 $P\ ystate$
and $\bigwedge s\ a\ ou\ s'.\ reachNT\ s \implies step\ s\ a = (ou, s') \implies \neg T\ (Trans\ s\ a\ ou\ s') \implies P\ s \implies P\ s'$
shows $P\ s$
 $\langle proof \rangle$

lemma $reachNT\text{-}PairI$:

assumes $reachNT\ s$ **and** $step\ s\ a = (ou, s')$ **and** $\neg T\ (Trans\ s\ a\ ou\ s')$
shows $reachNT\ s'$
 $\langle proof \rangle$

lemma $reachNT\text{-}state\text{-}cases$ [*cases set, consumes 1, case-names init step*]:

assumes $reachNT\ s$
obtains $s = ystate$
 $| sh\ a\ ou$ **where** $reach\ sh\ step\ sh\ a = (ou, s) \neg T\ (Trans\ sh\ a\ ou\ s)$
 $\langle proof \rangle$

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 $invarNT\ (\lambda s.\ Inv1\ s \vee Inv2\ s)$
 $\langle proof \rangle$

lemma $invarNT\text{-}conj$:

assumes $invarNT\ Inv1$ **and** $invarNT\ Inv2$
shows $invarNT\ (\lambda s.\ Inv1\ s \wedge Inv2\ s)$
 $\langle proof \rangle$

lemma $holdsIstate\text{-}invarNT$:

assumes h : $holdsIstate\ Inv$ **and** i : $invarNT\ Inv$ **and** a : $reachNT\ s$
shows $Inv\ s$
 $\langle proof \rangle$

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 $Abstract\text{-}BD\text{-}Security$

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 $\varphi' \text{ trn} \equiv \varphi \text{ trn} \vee T \text{ trn}$

abbreviation $f' \text{ trn} \equiv (\text{if } T \text{ trn then None else Some } (f \text{ trn}))$

abbreviation $T' \text{ trn} \equiv \text{False}$

abbreviation $B' \text{ vl}' \text{ vl1}' \equiv B \text{ (these vl')} \text{ (these vl1')} \wedge \text{never Option.is-none vl}' \wedge \text{never Option.is-none vl1}'$

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]*: $Prime.V\ tr = vl \implies never\ Option.is-none\ vl \longleftrightarrow never\ T\ tr$
 ⟨proof⟩

lemma *V'-V*: $never\ T\ tr \longleftrightarrow Prime.V\ tr = map\ Some\ (Orig.V\ tr)$
 ⟨proof⟩

lemma *V-Some-never-T*: $Prime.V\ tr = map\ Some\ vl \implies never\ T\ tr$
 ⟨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*: $Prime.secureTT \longleftrightarrow Prime.secure$
 ⟨proof⟩

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

lemma *secureTT-iff-secure'*: $Orig.secureTT \longleftrightarrow Prime.secure$
 ⟨proof⟩

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

lemma *secure'-secure*: $Prime.secure \implies Orig.secure$
 ⟨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'*: $Orig.secureTT \longleftrightarrow Prime.secureTT$
 ⟨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 \Rightarrow 'value list \Rightarrow 'value list \Rightarrow bool **where**
consume trn vl vl' \equiv
if φ trn then vl \neq [] \wedge f trn = hd vl \wedge vl' = tl vl
else vl' = vl

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

lemma *iactionI-ms*[intro?]:
assumes s: sstep s1 al1 = (ou1, 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') \longleftrightarrow step s1 a1 = (ou1, s1')
(proof)

lemma *iactionI*[intro?]:
assumes step s1 a1 = (ou1, s1') **and** φ (Trans s1 a1 ou1 s1')

and $\text{consume } (\text{Trans } s1 \ a1 \ ou1 \ s1') \ vl1 \ vl1'$
and $\neg \gamma (\text{Trans } s1 \ a1 \ ou1 \ s1')$ **and** $\Delta \ s \ vl \ s1' \ vl1'$
shows $\text{iaction } \Delta \ s \ vl \ s1 \ vl1$
 $\langle \text{proof} \rangle$

definition *match where*

$\text{match } \Delta \ s \ s1 \ vl1 \ a \ ou \ s' \ vl' \equiv$

$\exists \ al1 \ vl1'.$

$\text{let } \text{trn} = \text{Trans } s \ a \ ou \ s'; \ \text{tr1} = \text{traceOf } s1 \ al1; \ s1' = \text{tgtOf } (\text{last } \text{tr1}) \ \text{in}$

$al1 \neq [] \wedge \text{consumeList } \text{tr1} \ vl1 \ vl1' \wedge$

$O \ \text{tr1} = O \ [\text{trn}] \wedge$

$\Delta \ s' \ vl' \ s1' \ vl1'$

lemma *matchI-ms[intro?]:*

assumes $s: \text{sstep } s1 \ al1 = (ou1, s1')$

and $l: al1 \neq []$

and $\text{consumeList } (\text{traceOf } s1 \ al1) \ vl1 \ vl1'$

and $O \ (\text{traceOf } s1 \ al1) = O \ [\text{Trans } s \ a \ ou \ s']$

and $\Delta \ s' \ vl' \ s1' \ vl1'$

shows $\text{match } \Delta \ s \ s1 \ vl1 \ a \ ou \ s' \ vl'$

$\langle \text{proof} \rangle$

lemma *matchI[intro?]:*

assumes $\text{validTrans } (\text{Trans } s1 \ a1 \ ou1 \ s1')$

and $\text{consume } (\text{Trans } s1 \ a1 \ ou1 \ s1') \ vl1 \ vl1'$ **and** $\gamma (\text{Trans } s \ a \ ou \ s') = \gamma (\text{Trans } s1 \ a1 \ ou1 \ s1')$

and $\gamma (\text{Trans } s \ a \ ou \ s') \implies g (\text{Trans } s \ a \ ou \ s') = g (\text{Trans } s1 \ a1 \ ou1 \ s1')$

and $\Delta \ s' \ vl' \ s1' \ vl1'$

shows $\text{match } \Delta \ s \ s1 \ vl1 \ a \ ou \ s' \ vl'$

$\langle \text{proof} \rangle$

definition *ignore where*

$\text{ignore } \Delta \ s \ s1 \ vl1 \ a \ ou \ s' \ vl' \equiv$

$\neg \gamma (\text{Trans } s \ a \ ou \ s') \wedge$

$\Delta \ s' \ vl' \ s1 \ vl1$

lemma *ignoreI[intro?]:*

assumes $\neg \gamma (\text{Trans } s \ a \ ou \ s')$ **and** $\Delta \ s' \ vl' \ s1 \ vl1$

shows $\text{ignore } \Delta \ s \ s1 \ vl1 \ a \ ou \ s' \ vl'$

$\langle \text{proof} \rangle$

definition *reaction where*

$\text{reaction } \Delta \ s \ vl \ s1 \ vl1 \equiv$

$\forall \ a \ ou \ s' \ vl'.$

$\text{let } \text{trn} = \text{Trans } s \ a \ ou \ s' \ \text{in}$

$\text{validTrans } \text{trn} \wedge \neg \text{T } \text{trn} \wedge$

$\text{consume } \text{trn} \ vl \ vl'$

\longrightarrow

$\text{match } \Delta \ s \ s1 \ vl1 \ a \ ou \ s' \ vl'$

\vee
ignore $\Delta s s1 vl1 a ou s' vl'$

lemma *reactionI*[*intro?*]:

assumes

$\bigwedge a ou s' vl'$.

$\llbracket \text{step } s a = (ou, s'); \neg T (Trans s a ou s');$
 $\text{consume } (Trans s a ou s') vl vl' \rrbracket$

\implies

$\text{match } \Delta s s1 vl1 a ou s' vl' \vee \text{ignore } \Delta s s1 vl1 a ou s' vl'$

shows *reaction* $\Delta s vl s1 vl1$

<proof>

definition *exit* :: 'state \Rightarrow 'value \Rightarrow bool **where**

$\text{exit } s v \equiv \forall tr trn. \text{validFrom } s (tr \#\# trn) \wedge \text{never } T (tr \#\# trn) \wedge \varphi trn \longrightarrow f trn \neq v$

lemma *exit-coind*:

assumes *K*: *K s*

and *I*: $\bigwedge trn. \llbracket K (srcOf trn); \text{validTrans } trn; \neg T trn \rrbracket$

$\implies (\varphi trn \longrightarrow f trn \neq v) \wedge K (tgtOf trn)$

shows *exit s v*

<proof>

definition *noVal* **where**

$\text{noVal } K v \equiv$

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

lemma *noVal-disj*:

assumes *noVal Inv1 v* **and** *noVal Inv2 v*

shows *noVal* $(\lambda s. \text{Inv1 } s \vee \text{Inv2 } s) v$

<proof>

lemma *noVal-conj*:

assumes *noVal Inv1 v* **and** *noVal Inv2 v*

shows *noVal* $(\lambda s. \text{Inv1 } s \wedge \text{Inv2 } s) v$

<proof>

definition *no φ* **where**

$\text{no}\varphi K \equiv \forall s a ou s'. \text{reachNT } s \wedge K s \wedge \text{step } s a = (ou, s') \longrightarrow \neg \varphi (Trans s a ou s')$

lemma *no φ -noVal*: $\text{no}\varphi K \implies \text{noVal } K v$

<proof>

lemma *exitI*[*consumes 2, induct pred: exit*]:

assumes *rs*: *reachNT s* **and** *K*: *K s*

and *I*:

$\bigwedge s a ou s'$.

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

$$\implies (\varphi (\text{Trans } s \ a \ ou \ s') \longrightarrow f (\text{Trans } s \ a \ ou \ s') \neq v) \wedge K \ s'$$
shows $\text{exit } s \ v$
 $\langle \text{proof} \rangle$

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

definition *noVal2* **where**
 $\text{noVal2 } K \ v \equiv$
 $\forall s \ a \ ou \ s'. \text{reachNT } s \wedge K \ s \ v \wedge \text{step } s \ a = (ou, s') \wedge \varphi (\text{Trans } s \ a \ ou \ s') \longrightarrow f (\text{Trans } s \ a \ ou \ s') \neq v$

lemma *noVal2-disj*:
assumes $\text{noVal2 } \text{Inv1 } v$ **and** $\text{noVal2 } \text{Inv2 } v$
shows $\text{noVal2 } (\lambda s \ v. \text{Inv1 } s \ v \vee \text{Inv2 } s \ v) \ v$
 $\langle \text{proof} \rangle$

lemma *noVal2-conj*:
assumes $\text{noVal2 } \text{Inv1 } v$ **and** $\text{noVal2 } \text{Inv2 } v$
shows $\text{noVal2 } (\lambda s \ v. \text{Inv1 } s \ v \wedge \text{Inv2 } s \ v) \ v$
 $\langle \text{proof} \rangle$

lemma *noVal-noVal2*: $\text{noVal } K \ v \implies \text{noVal2 } (\lambda s \ v. K \ s) \ v$
 $\langle \text{proof} \rangle$

lemma *exitI-noVal2*[*consumes 2, induct pred: exit*]:
assumes $rs: \text{reachNT } s$ **and** $K: K \ s \ v$
and I :
 $\bigwedge s \ a \ ou \ s'. \llbracket \text{reach } s; \text{reachNT } s; \text{step } s \ a = (ou, s'); K \ s \ v \rrbracket$
 $\implies (\varphi (\text{Trans } s \ a \ ou \ s') \longrightarrow f (\text{Trans } s \ a \ ou \ s') \neq v) \wedge K \ s' \ v$
shows $\text{exit } s \ v$
 $\langle \text{proof} \rangle$

lemma *exitI2-noVal2*:
assumes $rs: \text{reachNT } s$ **and** $K: K \ s \ v$
and $\text{invarNT } (\lambda s. K \ s \ v)$ **and** $\text{noVal2 } K \ v$
shows $\text{exit } s \ v$
 $\langle \text{proof} \rangle$

lemma *exit-validFrom*:
assumes $vl: vl \neq []$ **and** $i: \text{exit } s \ (\text{hd } vl)$ **and** $v: \text{validFrom } s \ tr$ **and** $V: V \ tr = vl$

and T : *never* T tr
shows *False*
 $\langle proof \rangle$

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 $unwind \ \Delta$

$\langle proof \rangle$

lemma *unwind-trace*:

assumes *unwind*: $unwind \ \Delta$ **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*: $unwind \ \Delta$

shows *secure*

$\langle proof \rangle$

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

lemma $disjAll-simps[simp]$:

$disjAll \{\} \equiv \lambda - - -. False$

$disjAll (insert \Delta \Delta s) \equiv \lambda s vl s1 vl1. \Delta s vl s1 vl1 \vee disjAll \Delta s s1 vl1$

$\langle proof \rangle$

lemma $disjAll-mono$:

assumes $disjAll \Delta s s1 vl1$

and $\Delta s \subseteq \Delta s'$

shows $disjAll \Delta s' s1 vl1$

$\langle proof \rangle$

lemma $iaction-mono$:

assumes $1: iaction \Delta s s1 vl1$ **and** $2: \bigwedge s vl s1 vl1. \Delta s vl s1 vl1 \implies \Delta' s vl s1 vl1$

shows $iaction \Delta' s s1 vl1$

$\langle proof \rangle$

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 $match \Delta' s s1 vl1 a ou s' vl'$

$\langle proof \rangle$

lemma $ignore-mono$:

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

shows $ignore \Delta' s s1 vl1 a ou s' vl'$

$\langle proof \rangle$

lemma $reaction-mono$:

assumes $1: reaction \Delta s s1 vl1$ **and** $2: \bigwedge s vl s1 vl1. \Delta s vl s1 vl1 \implies \Delta' s vl s1 vl1$

shows $reaction \Delta' s s1 vl1$

$\langle proof \rangle$

5.2 Decomposition into an arbitrary network of components

definition $unwind-to$ where

$unwind-to \Delta \Delta s \equiv$

$\forall s vl s1 vl1.$

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

\longrightarrow

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

\vee

$iaction (disjAll \Delta s) s vl s1 vl1$

\vee

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

lemma $unwind-toI[intro?]$:

assumes

$\bigwedge s \text{ vl } s1 \text{ vl1}.$
 $\llbracket \text{reachNT } s; \text{reach } s1; \Delta s \text{ vl } s1 \text{ vl1} \rrbracket$
 \implies
 $\text{vl} \neq [] \wedge \text{exit } s (\text{hd } \text{vl})$
 \vee
 $\text{iaction } (\text{disjAll } \Delta s) s \text{ vl } s1 \text{ vl1}$
 \vee
 $(\text{vl} \neq [] \vee \text{vl1} = []) \wedge \text{reaction } (\text{disjAll } \Delta s) s \text{ vl } s1 \text{ vl1}$
shows *unwind-to* $\Delta \Delta s$
<proof>

lemma *unwind-dec*:
assumes *ne*: $\bigwedge \Delta. \Delta \in \Delta s \implies \text{next } \Delta \subseteq \Delta s \wedge \text{unwind-to } \Delta (\text{next } \Delta)$
shows *unwind* $(\text{disjAll } \Delta s)$ (**is** *unwind ?* Δ)
<proof>

lemma *init-dec*:
assumes $\Delta 0: \Delta 0 \in \Delta s$
and *i*: $\bigwedge \text{vl } \text{vl1}. B \text{ vl } \text{vl1} \implies \Delta 0 \text{ ystate } \text{vl} \text{ ystate } \text{vl1}$
shows $\forall \text{vl } \text{vl1}. B \text{ vl } \text{vl1} \longrightarrow \text{disjAll } \Delta s \text{ ystate } \text{vl} \text{ ystate } \text{vl1}$
<proof>

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

5.3 A customization for linear modular reasoning

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

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

shows *unwind-cont* $\Delta \Delta s$
 ⟨*proof*⟩

definition *unwind-exit* **where**

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

lemma *unwind-exitI*[*intro?*]:

assumes

$\bigwedge s \text{ vl } s1 \text{ vl1}.$
 $\llbracket \text{reachNT } s; \text{reach } s1; \Delta e \text{ } s \text{ vl } s1 \text{ vl1} \rrbracket$
 \implies
 $\text{vl} \neq [] \wedge \text{exit } s \text{ (hd vl)}$

shows *unwind-exit* Δe
 ⟨*proof*⟩

lemma *unwind-cont-mono*:

assumes Δs : *unwind-cont* $\Delta \Delta s$

and $\Delta s'$: $\Delta s \subseteq \Delta s'$

shows *unwind-cont* $\Delta \Delta s'$
 ⟨*proof*⟩

fun *allConsec* :: 'a list \Rightarrow ('a * 'a) set **where**

allConsec [] = {}
 | *allConsec* [a] = {}
 | *allConsec* (a # b # as) = insert (a,b) (*allConsec* (b#as))

lemma *set-allConsec*:

assumes $\Delta \in \text{set } \Delta s'$ **and** $\Delta s = \Delta s' \#\#\Delta 1$

shows $\exists \Delta 2. (\Delta, \Delta 2) \in \text{allConsec } \Delta s$
 ⟨*proof*⟩

lemma *allConsec-set*:

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

shows $\Delta 1 \in \text{set } \Delta s \wedge \Delta 2 \in \text{set } \Delta s$
 ⟨*proof*⟩

theorem *unwind-decomp-secure*:

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

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

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

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: $\bigwedge vl\ vl1. B\ vl\ vl1 \implies \Delta1\ ystate\ vl\ ystate\ vl1$

and *c1*: *unwind-cont* $\Delta1\ \{\Delta1, \Delta2, \Delta e\}$

and *c2*: *unwind-cont* $\Delta2\ \{\Delta2, \Delta3, \Delta e\}$

and *l*: *unwind-cont* $\Delta3\ \{\Delta3, \Delta e\}$

and *e*: *unwind-exit* Δe

shows *secure*

<proof>

corollary *unwind-decomp4-secure*:

assumes

i: $\bigwedge vl\ vl1. B\ vl\ vl1 \implies \Delta1\ ystate\ vl\ ystate\ vl1$

and *c1*: *unwind-cont* $\Delta1\ \{\Delta1, \Delta2, \Delta e\}$

and *c2*: *unwind-cont* $\Delta2\ \{\Delta2, \Delta3, \Delta e\}$

and *c3*: *unwind-cont* $\Delta3\ \{\Delta3, \Delta4, \Delta e\}$

and *l*: *unwind-cont* $\Delta4\ \{\Delta4, \Delta e\}$

and *e*: *unwind-exit* Δe

shows *secure*

<proof>

corollary *unwind-decomp5-secure*:

assumes

i: $\bigwedge vl\ vl1. B\ vl\ vl1 \implies \Delta1\ ystate\ vl\ ystate\ vl1$

and *c1*: *unwind-cont* $\Delta1\ \{\Delta1, \Delta2, \Delta e\}$

and *c2*: *unwind-cont* $\Delta2\ \{\Delta2, \Delta3, \Delta e\}$

and *c3*: *unwind-cont* $\Delta3\ \{\Delta3, \Delta4, \Delta e\}$

and *c4*: *unwind-cont* $\Delta4\ \{\Delta4, \Delta5, \Delta e\}$

and *l*: *unwind-cont* $\Delta5\ \{\Delta5, \Delta e\}$

and *e*: *unwind-exit* Δe

shows *secure*

<proof>

5.5 A graph alternative presentation

theorem *unwind-decomp-secure-graph*:

assumes *n*: $\forall \Delta \in \text{Domain } Gr. \exists \Delta s. \Delta s \subseteq \text{Domain } Gr \wedge (\Delta, \Delta s) \in Gr$

and *i*: $\Delta0 \in \text{Domain } Gr \wedge vl\ vl1. B\ vl\ vl1 \implies \Delta0\ ystate\ vl\ ystate\ vl1$

and *c*: $\bigwedge \Delta. \text{unwind-exit } \Delta \vee (\forall \Delta s. (\Delta, \Delta s) \in Gr \implies \text{unwind-cont } \Delta\ \Delta s)$

shows *secure*

<proof>

References

- [1] T. Bauerei, 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. Bauerei, 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. Bauerei, 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 fr 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.