

# Bounded-Deducibility Security

Andrei Popescu      Peter Lammich      Thomas Bauereiss

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Preliminaries</b>	<b>2</b>
2.1	Transition Systems . . . . .	6
2.1.1	Traces . . . . .	6
2.1.2	Reachability . . . . .	9
2.2	IO automata . . . . .	11
2.2.1	IO automata as transition systems . . . . .	11
2.2.2	State invariants . . . . .	14
2.2.3	Traces of actions . . . . .	14
<b>3</b>	<b>BD Security</b>	<b>17</b>
3.1	Abstract definition . . . . .	17
3.2	Instantiation for transition systems . . . . .	18
3.3	Instantiation for IO automata . . . . .	22
3.4	Trigger-preserving BD security . . . . .	24
3.4.1	Definition . . . . .	24
3.4.2	Incorporating static triggers into the bound . . . . .	25
3.4.3	Reflexive-transitive closure of declassification bounds . . . . .	26
<b>4</b>	<b>Unwinding proof method</b>	<b>27</b>
<b>5</b>	<b>Compositional Reasoning</b>	<b>36</b>
5.1	Preliminaries . . . . .	36
5.2	Decomposition into an arbitrary network of components . . . . .	37
5.3	A customization for linear modular reasoning . . . . .	38
5.4	Instances . . . . .	40
5.5	A graph alternative presentation . . . . .	41

## 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].

## 2 Preliminaries

```

function filtermap ::
('trans  $\Rightarrow$  bool)  $\Rightarrow$  ('trans  $\Rightarrow$  'a)  $\Rightarrow$  'trans list  $\Rightarrow$  'a list
where
filtermap pred func [] = []
|
 $\neg$  pred trn  $\Longrightarrow$  filtermap pred func (trn # tr) = filtermap pred func tr
|
pred trn  $\Longrightarrow$  filtermap pred func (trn # tr) = func trn # filtermap pred func tr
by auto (metis list.exhaust)
termination by lexicographic-order

```

```

lemma filtermap-map-filter: filtermap pred func xs = map func (filter pred xs)
by (induction xs) auto

```

```

lemma filtermap-append: filtermap pred func (tr @ tr1) = filtermap pred func tr @ filtermap pred func tr1
proof(induction tr arbitrary: tr1)
  case (Cons trn tr)
  thus ?case by (cases pred trn) auto
qed auto

```

```

lemma filtermap-Nil-list-ex: filtermap pred func tr = []  $\longleftrightarrow$   $\neg$  list-ex pred tr
proof(induction tr)
  case (Cons trn tr)
  thus ?case by (cases pred trn) auto
qed auto

```

```

lemma filtermap-Nil-never: filtermap pred func tr = []  $\longleftrightarrow$  never pred tr
proof(induction tr)
  case (Cons trn tr)
  thus ?case by (cases pred trn) auto
qed auto

```

```

lemma length-filtermap: length (filtermap pred func tr)  $\leq$  length tr
proof(induction tr)
  case (Cons trn tr)

```

**thus** ?case **by** (cases pred trn) auto  
**qed** auto

**lemma** filtermap-list-all[simp]: filtermap pred func tr = map func tr  $\longleftrightarrow$  list-all pred tr  
**proof**(induction tr)  
  **case** (Cons trn tr)  
  **thus** ?case **apply** (cases pred trn)  
  **by** (simp-all) (metis impossible-Cons length-filtermap length-map)  
**qed** auto

**lemma** filtermap-eq-Cons:

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

**shows**  $\exists$  trn tr2 tr1.

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

**using** assms **proof**(induction tr arbitrary: a al1)

**case** (Cons trn tr a al1)

**show** ?case

**proof**(cases pred trn)

**case** False

**hence** filtermap pred func tr = a # al1 **using** Cons **by** simp

**from** Cons(1)[OF this] **obtain** trnn tr2 tr1 **where**

    1: tr = tr2 @ [trnn] @ tr1  $\wedge$  never pred tr2  $\wedge$  pred trnn  $\wedge$  func trnn = a  $\wedge$

    filtermap pred func tr1 = al1 **by** blast

**show** ?thesis **apply**(rule exI[of - trnn], rule exI[of - trn # tr2], rule exI[of - tr1])

**using** Cons(2) 1 False **by** simp

**next**

**case** True

**hence** filtermap pred func tr = al1 **using** Cons **by** simp

**show** ?thesis **apply**(rule exI[of - trn], rule exI[of - []], rule exI[of - tr])

**using** Cons(2) True **by** simp

**qed**

**qed** auto

**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

**using** assms **proof**(induction al1 arbitrary: tr)

**case** Nil **show** ?case

**apply** (rule exI[of - []], rule exI[of - tr]) **using** Nil **by** auto

**next**

**case** (Cons a al1 tr)

**hence** filtermap pred func tr = a # (al1 @ al2) **by** simp

**from** filtermap-eq-Cons[OF this] **obtain** trn tr2 tr1

**where** tr: tr = tr2 @ [trn] @ tr1 **and** n: never pred tr2  $\wedge$  pred trn  $\wedge$  func trn = a

**and** f: filtermap pred func tr1 = al1 @ al2 **by** blast

**from** Cons(1)[OF f] **obtain** tr11 tr22 **where** tr1: tr1 = tr11 @ tr22

**and** f1: filtermap pred func tr11 = al1 **and** f2: filtermap pred func tr22 = al2 **by** blast

**show** ?case **apply** (rule exI[of - tr2 @ [trn] @ tr11], rule exI[of - tr22])

**using** n filtermap-Nil-never f1 f2 **unfolding** tr tr1 filtermap-append **by** auto

**qed**

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

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

**proof**(*induction tr*)

**case** (*Cons trn1 tr*)

**thus** ?*case by* (*cases pred trn1*) *auto*

**qed** *auto*

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

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

**proof**(*induction tr*)

**case** (*Cons trn1 tr*)

**thus** ?*case by* (*cases pred trn1*) *auto*

**qed** *auto*

**lemma** *filtermap-eq-RCons*:

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

**shows**  $\exists\ trn\ tr1\ tr2.$

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

**using** *assms* **proof**(*induction tr arbitrary: a al1 rule: rev-induct*)

**case** (*snoc trn tr a al1*)

**show** ?*case*

**proof**(*cases pred trn*)

**case** *False*

**hence**  $filtermap\ pred\ func\ tr = al1\ \#\# \ a$  **using** *snoc by simp*

**from** *snoc(1)[OF this]* **obtain** *trnn tr2 tr1* **where**

$1: tr = tr1\ @\ [trnn]\ @\ tr2 \wedge never\ pred\ tr2 \wedge pred\ trnn \wedge func\ trnn = a \wedge$

$filtermap\ pred\ func\ tr1 = al1$  **by** *blast*

**show** ?*thesis* **apply**(*rule exI[of - trnn], rule exI[of - tr1], rule exI[of - tr2 ## trn]*)

**using** *snoc(2) 1 False by simp*

**next**

**case** *True*

**hence**  $filtermap\ pred\ func\ tr = al1$  **using** *snoc by simp*

**show** ?*thesis* **apply**(*rule exI[of - trn], rule exI[of - tr], rule exI[of - []]*)

**using** *snoc(2) True by simp*

**qed**

**qed** *auto*

**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**—

```

from filtermap-eq-Cons[OF assms] obtain trna tra tr2
where 0: tr = tra @ [trna] @ tr2  $\wedge$  never pred tra  $\wedge$  pred trna  $\wedge$  func trna = a
and 1: filtermap pred func tr2 = al1 ## b by auto
from filtermap-eq-RCons[OF 1] obtain trnb tr1 trb where
2: tr2 = tr1 @ [trnb] @ trb  $\wedge$  never pred trb  $\wedge$ 
pred trnb  $\wedge$  func trnb = b  $\wedge$  filtermap pred func tr1 = al1 by blast
show ?thesis apply(rule exI[of - tra], rule exI[of - trna], rule exI[of - tr1],
rule exI[of - trnb], rule exI[of - trb])
using 2 0 by auto
qed

```

```

lemma filter-Nil-never: [] = filter pred xs  $\implies$  never pred xs
by (induction xs) (auto split: if-splits)

```

```

lemma never-Nil-filter: never pred xs  $\longleftrightarrow$  [] = filter pred xs
by (induction xs) (auto split: if-splits)

```

```

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
using assms proof (induction ys rule: rev-induct)
case Nil then show ?case by auto
next
case (snoc y ys)
show ?case
proof (cases)
assume Q y
moreover then have x = y using snoc.prem1 by auto
ultimately show thesis using snoc(3) snoc(2) by auto
next
assume  $\neg$ Q y
show thesis
proof (rule snoc.IH)
show xs ## x = filter Q ys using  $\langle \neg$ Q y  $\rangle$  snoc(3) by auto
next
fix us vs
assume ys = us @ x # vs and never Q vs and Q x and xs = filter Q us
then show thesis using  $\langle \neg$ Q y  $\rangle$  snoc(2) by auto
qed
qed
qed

```

```

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)
unfolding filtermap-append[of pred func [x, x'] zs, simplified]
filtermap-append[of pred func [y, y'] zs, simplified]
by simp

```

**lemma** *filtermap-Cons-cong*:  
 $filtermap\ pred\ func\ xs = filtermap\ pred\ func\ ys$   
 $\implies filtermap\ pred\ func\ (x\ \#\ xs) = filtermap\ pred\ func\ (x\ \#\ ys)$   
**by** (*cases pred x*) *auto*

**lemma** *set-filtermap*:  
 $set\ (filtermap\ pred\ func\ xs) \subseteq \{func\ x \mid x . x \in xs \wedge pred\ x\}$   
**by** (*induct xs, simp*) (*case-tac pred a, auto*)

## 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]*:  
 $tr \neq [] \implies srcOfTrFrom\ s\ tr = srcOfTr\ tr$   
**by** (*cases tr*) *auto*

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

**lemma** *tgtOfTrFrom-tgtOfTr[simp]*:  
 $tr \neq [] \implies tgtOfTrFrom\ s\ tr = tgtOfTr\ tr$   
**by** (*cases tr*) *auto*

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
  =>
  valid [trn]
|
Cons[intro]:
[[validTrans trn; tgtOf trn = srcOf (hd tr); valid tr]]
  =>
  valid (trn # tr)

```

```

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

```

```

lemma Nil-not-valid[simp]: ¬ valid []
by (metis valid.simps neq-Nil-conv)

```

```

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

```

```

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

```

```

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

```

```

lemma valid2-hd-Rcons[simp]:
assumes valid2 tr

```

**shows**  $hd (tr \#\# tran) = hd tr$   
**by** (*metis Nil-not-valid2 assms hd-append*)

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

**lemma** *valid2-Cons*:  
**assumes** *valid2 tr* **and**  $tgtOf\ trn = srcOf (hd\ tr)$  **and** *validTrans trn*  
**shows** *valid2 (trn \# tr)*  
**using** *assms* **proof**(*induct arbitrary: trn*)  
  **case** *Singl* **show** *?case*  
  **unfolding** *two-singl-Rcons* **apply**(*rule valid2.Rcons*) **using** *Singl*  
  **by** (*auto intro: valid2.Singl*)  
**next**  
  **case** *Rcons* **show** *?case*  
  **unfolding** *append.append-Cons[symmetric]* **apply**(*rule valid2.Rcons*) **using** *Rcons* **by** *auto*  
**qed**

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

**lemma** *valid-Cons-iff*:  
 $valid (trn \# tr) \longleftrightarrow validTrans\ trn \wedge ((tgtOf\ trn = srcOf (hd\ tr) \wedge valid\ tr) \vee tr = [])$   
**unfolding** *valid.simps[of trn \# tr]* **by** *auto*

**lemma** *valid-append*:  
 $tr \neq [] \implies tr1 \neq [] \implies$   
 $valid (tr @ tr1) \longleftrightarrow valid\ tr \wedge valid\ tr1 \wedge tgtOf (last\ tr) = srcOf (hd\ tr1)$   
**by** (*induct tr*) (*auto simp add: valid-Cons-iff*)

**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!]*:  $validFrom\ s\ []$   
**unfolding** *validFrom-def* **by** *auto*

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



**unfolding** *validFrom-def* **by** *auto*

**lemma** *validFrom-append*:

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

**unfolding** *validFrom-def* **using** *valid-append*

**by** (*cases*  $\text{tr} = [] \vee \text{tr1} = []$ ) *fastforce+*

**lemma** *validFrom-Cons*:

$\text{validFrom } s \text{ (trn \# tr)} \longleftrightarrow \text{validTrans trn} \wedge \text{srcOf trn} = s \wedge \text{validFrom (tgtOf trn) tr}$

**unfolding** *validFrom-def* **by** *auto*

### 2.1.2 Reachability

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

*Istate*: *reach* *istate*

|

*Step*: *reach* *s*  $\Longrightarrow$  *validTrans* *trn*  $\Longrightarrow$  *srcOf* *trn* = *s*  $\Longrightarrow$  *tgtOf* *trn* = *s'*  $\Longrightarrow$  *reach* *s'*

**lemma** *valid-reach-src-tgt*:

**assumes** *valid* *tr* **and** *reach* (*srcOf* (*hd* *tr*))

**shows** *reach* (*tgtOf* (*last* *tr*))

**using** *assms* *Step* **by** *induct* *auto*

**lemma** *valid-init-reach*:

**assumes** *valid* *tr* **and** *srcOf* (*hd* *tr*) = *istate*

**shows** *reach* (*tgtOf* (*last* *tr*))

**using** *valid-reach-src-tgt* *assms* *reach.Istate* **by** *metis*

**lemma** *reach-init-valid*:

**assumes** *reach* *s*

**shows**

*s* = *istate*

$\vee$

$(\exists \text{tr. } \text{valid } \text{tr} \wedge \text{srcOf (hd tr)} = \text{istate} \wedge \text{tgtOf (last tr)} = s)$

**using** *assms* **proof** *induction*

**case** (*Step* *s* *trn* *s'*)

**thus** *?case* **proof**(*elim* *disjE* *exE* *conjE*)

**assume** *s*: *s* = *istate*

**show** *?thesis*

**apply** (*intro* *disjI2* *exI*[*of* - [*trn*]])

**using** *s* *Step* **by** *auto*

**next**

**fix** *tr* **assume** *v*: *valid* *tr* **and** *s*: *srcOf* (*hd* *tr*) = *istate* **and** *t*: *tgtOf* (*last* *tr*) = *s*

**show** *?thesis*

**apply** (*intro* *disjI2* *exI*[*of* - *tr* *##* *trn*])

**using** *Step* *v* *t* *s* **by** (*auto* *intro*: *valid-Rcons*)

**qed**  
**qed** *auto*

**lemma** *reach-validFrom*:  
**assumes** *reach s'*  
**shows**  $\exists s tr. s = \text{istate} \wedge (s = s' \vee (\text{validFrom } s \text{ } tr \wedge \text{tgtOf } (\text{last } tr) = s'))$   
**using** *reach-init-valid[OF assms]* **unfolding** *validFrom-def* **by** *auto*

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

**lemma** *reachFrom-Step1*:  
 $\llbracket \text{validTrans } trn; \text{srcOf } trn = s; \text{tgtOf } trn = s' \rrbracket \Longrightarrow \text{reachFrom } s \text{ } s'$   
**by** (*auto intro: reachFrom.Step*)

**lemma** *reachFrom-Step-Left*:  
 $\text{reachFrom } s' \text{ } s'' \Longrightarrow \text{validTrans } trn \Longrightarrow \text{srcOf } trn = s \Longrightarrow \text{tgtOf } trn = s' \Longrightarrow \text{reachFrom } s \text{ } s''$   
**by** (*induction s'' rule: reachFrom.induct*) (*auto intro: reachFrom.Step*)

**lemma** *reachFrom-trans*:  $\text{reachFrom } s0 \text{ } s1 \Longrightarrow \text{reachFrom } s1 \text{ } s2 \Longrightarrow \text{reachFrom } s0 \text{ } s2$   
**by** (*induction s1 arbitrary: s2 rule: reachFrom.induct*) (*auto intro: reachFrom-Step-Left*)

**lemma** *reachFrom-reach*:  $\text{reachFrom } s \text{ } s' \Longrightarrow \text{reach } s \Longrightarrow \text{reach } s'$   
**by** (*induction rule: reachFrom.induct*) (*auto intro: reach.Step*)

**lemma** *valid-validTrans-set*:  
**assumes** *valid tr and trn  $\in\in$  tr*  
**shows** *validTrans trn*  
**using** *assms* **by** (*induct tr arbitrary: trn*) *auto*

**lemma** *validFrom-validTrans-set*:  
**assumes** *validFrom s tr and trn  $\in\in$  tr*  
**shows** *validTrans trn*  
**by** (*metis assms validFrom-def empty-iff list.set valid-validTrans-set*)

**lemma** *valid-validTrans-nth*:  
**assumes** *v: valid tr and i:  $i < \text{length } tr$*   
**shows** *validTrans (tr!i)*  
**using** *valid-validTrans-set[OF v] i* **by** *auto*

**lemma** *valid-validTrans-nth-srcOf-tgtOf*:  
**assumes** *v: valid tr and i:  $\text{Suc } i < \text{length } tr$*   
**shows**  $\text{srcOf } (tr!(\text{Suc } i)) = \text{tgtOf } (tr!i)$   
**by** (*metis Cons-nth-drop-Suc valid-append Suc-lessD append-self-conv2 hd-drop-conv-nth i id-take-nth-drop list.distinct(1) v valid-ConsE*)

**lemma** *validFrom-reach*:  $\text{validFrom } s \ tr \implies \text{reach } s \implies tr \neq [] \implies \text{reach } (\text{tgtOf } (\text{last } tr))$   
**by** (*intro valid-reach-src-tgt*) (*auto simp add: validFrom-def*)

**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** *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*))

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

**sublocale** *Transition-System*

**where** *istate* = *istate* **and** *validTrans* = *validTrans* **and** *srcOf* = *srcOf* **and** *tgtOf* = *tgtOf* .

**lemma** *reach-step*:

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

**using** *reach.Step* **where** *trn* = *Trans s a ou (snd (step s a))* **for** *ou*

**by** (*cases step s a*) *auto*

```

lemma reach-PairI:
  assumes reach s and step s a = (ou, s')
  shows reach s'
  using assms
  by (auto intro: reach.Step[where trn = Trans s a ou s'])

lemma reach-step-induct[consumes 1, case-names Istate Step]:
  assumes s: reach s
    and istate: P istate
    and step:  $\bigwedge s a. reach\ s \implies P\ s \implies P\ (snd\ (step\ s\ a))$ 
  shows P s
proof (use s in induction)
  case Istate
  then show ?case
    by (rule istate)
next
  case (Step s trn s')
  then obtain a ou where trn = Trans s a ou s'
    by (cases trn) auto
  then show ?case
    using Step step[of s a]
    by auto
qed

lemma reachFrom-step-induct[consumes 1, case-names Refl Step]:
  assumes s: reachFrom s s'
    and refl: P s
    and step:  $\bigwedge s' a ou s''. reachFrom\ s\ s' \implies P\ s' \implies step\ s'\ a = (ou, s'') \implies P\ s''$ 
  shows P s'
proof (use s in induction)
  case Refl
  then show ?case
    by (rule refl)
next
  case (Step s' trn s'')
  then obtain a ou where trn = Trans s' a ou s''
    by (cases trn) auto
  then show ?case
    using Step step[of s' a ou s'']
    by auto
qed

lemma valid-filter-no-state-change:
  valid tr  $\implies$  ( $\bigwedge trn. trn \in tr \implies \neg(PP\ trn) \implies srcOf\ trn = tgtOf\ trn) \implies$ 
   $\exists trn. trn \in tr \wedge PP\ trn \implies valid\ (filter\ PP\ tr) \wedge srcOfTr\ tr = srcOfTr\ (filter\ PP\ tr)$ 
   $\wedge tgtOfTr\ tr = tgtOfTr\ (filter\ PP\ tr)$ 
proof (induct rule: valid.induct)
  case (Singl trn) then show ?case by auto

```

```

next
case (Cons trn tr) then show ?case
proof (cases PP trn)
  case True note * = this show ?thesis
  proof (cases  $\exists$  trn. trn  $\in$  tr  $\wedge$  PP trn)
    case True then show ?thesis using * Cons by fastforce
  next
  case False then show ?thesis
  proof -
    have **: filter PP tr = [] using False by auto
    show ?thesis
    proof (cases tr = [])
      case True then show ?thesis using Cons by simp
    next
    case False
      with Cons(3) Cons(5) ** have srcOfTr tr = tgtOfTr tr
      proof (induction tr)
        case (Sngl a)
          have  $\neg$  (PP a) using Sngl(3) by auto
          then show ?case using Sngl(2) by auto
        next
        case (Cons a as)
          have **:  $\neg$  (PP a) using Cons(6) by auto
          then have *: srcOf a = tgtOf a using Cons(5) by auto
          show ?case
          proof (cases as = [])
            case True with * show ?thesis by simp
          next
            case False
              then have srcOfTr as = tgtOfTr as using Cons ** by auto
              then show ?thesis using * Cons(2) by auto
            qed
          qed
          then show ?thesis using * ** Cons False by simp
        qed
      qed
    qed
  next
  case False then show ?thesis using Cons by auto
qed
qed

```

```

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

```

### 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*  $\varphi \equiv \forall s a. \text{reach } s \wedge \varphi s \longrightarrow \varphi (\text{snd } (\text{step } s a))$

**lemma** *holdsIstate-invar*:

**assumes** *h*: *holdsIstate*  $\varphi$  **and** *i*: *invar*  $\varphi$  **and** *a*: *reach s*  
**shows**  $\varphi s$   
**by** (*use a in*  $\langle$ *induction rule: reach-step-induct* $\rangle$ )  
*(use h i in*  $\langle$ *auto simp: holdsIstate-def invar-def* $\rangle$ )

### 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*  $s (a \# al) = (\text{case step } s a \text{ of } (ou,s') \Rightarrow (\text{case sstep } s' al \text{ of } (oul, s'') \Rightarrow (ou \# oul, s'')))$

**lemma** *length-traceOf[simp]*:  
*length (traceOf s al) = length al*  
**by** (*induct al arbitrary: s*) (*auto split: prod.splits*)

**lemma** *traceOf-Nil[simp]*:  
*traceOf s al = []  $\longleftrightarrow$  al = []*  
**by** (*metis length-traceOf length-0-conv*)

**lemma** *sstep-outOf-traceOf[simp]*:  
*sstep s al = (ou,s')  $\Longrightarrow$  map outOf (traceOf s al) = ou*  
**by** (*induct al arbitrary: s ou s'*) (*auto split: prod.splits*)

**lemma** *sstep-tgtOf-traceOf[simp]*:  
*al  $\neq [] \Longrightarrow$  sstep s al = (ou,s')  $\Longrightarrow$  tgtOf (last (traceOf s al)) = s'*  
**by** (*induct al arbitrary: s ou s'*) (*auto split: prod.splits*)

**lemma** *srcOf-traceOf[simp]*:  
*al  $\neq [] \Longrightarrow$  srcOf (hd (traceOf s al)) = s*  
**by** (*induct al arbitrary: s*) (*auto split: prod.splits*)

**lemma** *actOf-traceOf[simp]*:  
*map actOf (traceOf s al) = al*  
**by** (*induct al arbitrary: s*) (*auto split: prod.splits*)

**lemma** *traceOf-append*:  
 $al \neq [] \implies s1 = \text{tgtOf } (\text{last } (\text{traceOf } s \text{ al})) \implies$   
 $\text{traceOf } s \text{ (al @ al1)} = \text{traceOf } s \text{ al @ traceOf } s1 \text{ al1}$   
**by** (*induct al arbitrary: s s1 al1*) (*auto split: prod.splits*)

**lemma** *sstep-append*:  
**assumes**  $sstep \ s \ al = (oul, s1)$  **and**  $sstep \ s1 \ al1 = (oul1, s2)$   
**shows**  $sstep \ s \ (al @ al1) = (oul @ oul1, s2)$   
**using** *assms* **by** (*induct al arbitrary: oul s s1 oul1 s2*) (*auto split: prod.splits*)

**lemma** *reach-sstep*:  
**assumes**  $reach \ s$  **and**  $sstep \ s \ al = (ou, s1)$   
**shows**  $reach \ s1$   
**using** *assms* **apply** (*induction al arbitrary: ou s1 s*)  
**by** (*auto split: prod.splits*) (*metis reach-PairI*)

**lemma** *traceOf-consR[simp]*:  
**assumes**  $al \neq []$  **and**  $s1 = \text{tgtOf } (\text{last } (\text{traceOf } s \text{ al}))$  **and**  $\text{step } s1 \ a = (ou, s2)$   
**shows**  $\text{traceOf } s \ (al \## a) = \text{traceOf } s \ al \## \text{Trans } s1 \ a \ ou \ s2$   
**using** *assms* **by** (*induct al arbitrary: s*) (*auto split: prod.splits*)

**lemma** *sstep-consR[simp]*:  
**assumes**  $sstep \ s \ al = (oul, s1)$  **and**  $\text{step } s1 \ a = (ou, s2)$   
**shows**  $sstep \ s \ (al \## a) = (oul \## ou, s2)$   
**using** *assms* **by** (*induct al arbitrary: oul s s1 ou s2*) (*auto split: prod.splits*)

**lemma** *fst-sstep-consR*:  
 $\text{fst } (sstep \ s \ (al \## a)) = \text{fst } (sstep \ s \ al) \## (\text{fst } (\text{step } (snd \ (sstep \ s \ al)) \ a))$   
**by** (*cases sstep s al, cases step (snd (sstep s al)) a*) *auto*

**lemma** *valid-traceOf[simp]*:  $al \neq [] \implies \text{valid } (\text{traceOf } s \ al)$   
**proof** (*induct al arbitrary: s*)  
  **case** (*Cons a al*)  
  **thus** *?case* **by** (*cases al = []*) (*auto split: prod.splits*)  
**qed** *auto*

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

**lemma** *validFrom-traceOf2*:  
**assumes**  $\text{validFrom } s \ tr$   
**shows**  $tr = \text{traceOf } s \ (\text{map } \text{actOf } tr)$   
**using** *assms*

**by** (*induction tr arbitrary: s*) (*auto split: prod.splits simp: validFrom-def elim!: validTrans.elims*)

**lemma** *set-traceOf-validTrans:*

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

**by** (*metis assms validFrom-traceOf validFrom-validTrans-set*)

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

**by** (*induction al arbitrary: s al1*) (*auto split: prod.splits*)

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

**by** (*cases sstep s al, cases sstep (snd (sstep s al)) al1*) (*auto simp add: sstep-append*)

**lemma** *snd-sstep-step-constant:*

**assumes**  $\forall a. a \in \text{al} \longrightarrow \text{snd (step s a)} = s$

**shows**  $\text{snd (sstep s al)} = s$

**using** *assms* **by** (*induction al*) (*auto split: prod.splits*)

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

**lemma** *const-tr-same-src-tgt:*

**assumes** *valid tr const-tr tr*

**shows**  $\text{srcOfTr } tr = \text{tgtOfTr } tr$

**using** *assms unfolding const-tr-def* **by** *induction auto*

**lemma** *traceOf-snoc:*

$\text{traceOf } s \text{ (al ## a)} =$

$\text{traceOf } s \text{ al ##}$

$\text{Trans (snd (sstep s al))}$

$a$

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

$(\text{snd (step (snd (sstep s al)) a)})$

**by** (*metis (no-types, lifting) traceOf-Nil traceOf-append-sstep prod.case-eq-if traceOf.simps*)

**lemma** *traceOf-append-unfold:*

$\text{traceOf } s \text{ (al1 @ al2)} =$

$\text{traceOf } s \text{ al1 @ traceOf (if al1 = [] then s else tgtOf (last (traceOf s al1))) al2$

**using** *traceOf-append* **by** (*cases al1 = []*) *auto*

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

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

**by** (*auto split: prod.splits*)

**definition** *commute s a1 a2*

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

**definition** *absorb*  $:: 'state \Rightarrow 'act \Rightarrow 'act \Rightarrow bool$  **where**



$absorb\ s\ a1\ a2 \equiv snd\ (sstep\ s\ [a1,\ a2]) = snd\ (step\ s\ a2)$

**lemma** *validFrom-commute*:

**assumes**  $validFrom\ s0\ (tr1\ @\ transOf\ s\ a\ \# \ transOf\ (snd\ (step\ s\ a))\ a'\ \# \ tr2)$

**and**  $commute\ s\ a\ a'$

**shows**  $validFrom\ s0\ (tr1\ @\ transOf\ s\ a'\ \# \ transOf\ (snd\ (step\ s\ a'))\ a\ \# \ tr2)$

**using** *assms unfolding commute-def* **by** (*auto split: prod.splits simp add: validFrom-append validFrom-Cons*)

**lemma** *validFrom-absorb*:

**assumes**  $validFrom\ s0\ (tr1\ @\ transOf\ s\ a\ \# \ transOf\ (snd\ (step\ s\ a))\ a'\ \# \ tr2)$

**and**  $absorb\ s\ a\ a'$

**shows**  $validFrom\ s0\ (tr1\ @\ transOf\ s\ a'\ \# \ tr2)$

**using** *assms unfolding absorb-def* **by** (*auto split: prod.splits simp add: validFrom-append validFrom-Cons*)

**lemma** *validTrans-Trans-srcOf-actOf-tgtOf*:

$validTrans\ trn \implies Trans\ (srcOf\ trn)\ (actOf\ trn)\ (outOf\ trn)\ (tgtOf\ trn) = trn$

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

**lemma** *validTrans-step-srcOf-actOf-tgtOf*:

$validTrans\ trn \implies step\ (srcOf\ trn)\ (actOf\ trn) = (outOf\ trn,\ tgtOf\ trn)$

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

**lemma** *sstep-Cons*:

$sstep\ s\ (a\ \# \ al) = (fst\ (step\ s\ a)\ \# \ fst\ (sstep\ (snd\ (step\ s\ a))\ al),\ snd\ (sstep\ (snd\ (step\ s\ a))\ al))$

**by** (*auto split: prod.splits*)

**declare** *sstep.simps(2)[simp del]*

**lemma** *length-fst-sstep*:  $length\ (fst\ (sstep\ s\ al)) = length\ al$

**by** (*induction al arbitrary: s*) (*auto simp: sstep-Cons*)

## 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*

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

**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**

$\varphi$  :: *'trans*  $\Rightarrow$  *bool* **and**  $f$  :: *'trans*  $\Rightarrow$  *'value*

**and**

$\gamma$  :: *'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  $TT = never\ T$  .

**lemma** *O-map-filter*:  $O\ tr = map\ g\ (filter\ \gamma\ tr)$  **unfolding** *O-def filtermap-map-filter ..*

**lemma** *V-map-filter*:  $V\ tr = map\ f\ (filter\ \varphi\ tr)$  **unfolding** *V-def filtermap-map-filter ..*

**lemma** *V-simps[simp]*:

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

**unfolding** *V-def by auto*

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

**by** *auto*

**lemma** *O-simps[simp]*:

$O\ [] = [] \ \neg\ \gamma\ trn \implies O\ (trn\ \#\ tr) = O\ tr\ \gamma\ trn \implies O\ (trn\ \#\ tr) = g\ trn\ \#\ O\ tr$

**unfolding** *O-def by auto*

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

**by** *auto*

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

**unfolding** *V-def using filtermap-append by auto*

**lemma** *V-snoc*:

$\neg\ \varphi\ trn \implies V\ (tr\ \#\#\ trn) = V\ tr\ \varphi\ trn \implies V\ (tr\ \#\#\ trn) = V\ tr\ \#\#\ f\ trn$

**unfolding** *V-def by auto*

**lemma** *O-snoc*:

$\neg\ \gamma\ trn \implies O\ (tr\ \#\#\ trn) = O\ tr\ \gamma\ trn \implies O\ (tr\ \#\#\ trn) = O\ tr\ \#\#\ g\ trn$

**unfolding** *O-def by auto*

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

**unfolding** *V-def using filtermap-Nil-list-ex by auto*

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

**unfolding** *V-def using filtermap-Nil-never by auto*

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

**unfolding** *V-def filtermap-map-filter by (induction tr) auto*

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

$list\text{-}ex\ \varphi\ tr \iff length\ (V\ tr) > 0$

**by** (*metis V-Nil-list-ex length-greater-0-conv*)

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

**by** (*auto simp: V-def length-filtermap*)

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

**by** (*auto simp: V-def length-filtermap*)

**lemma** *V-eq-Cons*:

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

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

**using** *assms filtermap-eq-Cons unfolding V-def by auto*

**lemma** *V-eq-append*:

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

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

**using** *assms filtermap-eq-append[of  $\varphi\ f$ ] unfolding V-def by auto*

**lemma** *V-eq-RCons*:

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

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

**using** *assms filtermap-eq-RCons[of  $\varphi\ f$ ] unfolding V-def by blast*

**lemma** *V-eq-Cons-RCons*:

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

**shows**  $\exists\ trv\ trnv\ tr1\ trnw\ trw.$

$tr = trv\ @\ [trnv]\ @\ tr1\ @\ [trnw]\ @\ trw \wedge$

$never\ \varphi\ trv \wedge \varphi\ trnv \wedge f\ trnw = v \wedge V\ tr1 = vl1 \wedge \varphi\ trnw \wedge f\ trnw = w \wedge never\ \varphi\ trw$

**using** *assms filtermap-eq-Cons-RCons[of  $\varphi\ f$ ] unfolding V-def by blast*

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

**unfolding** *O-def using filtermap-append by auto*

**lemma** *O-Nil-list-ex*:  $O\ tr = [] \iff \neg\ list\ ex\ \gamma\ tr$

**unfolding** *O-def using filtermap-Nil-list-ex by auto*

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

**unfolding** *O-def using filtermap-Nil-never by auto*

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

**unfolding** *O-def filtermap-map-filter by (induction tr) auto*

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

**by** *(auto simp: O-def length-filtermap)*

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

**by** *(auto simp: O-def length-filtermap)*

**lemma** *O-eq-Cons*:

**assumes**  $O\ tr = obs \# \ obsl1$

**shows**  $\exists\ trn\ tr2\ tr1. tr = tr2\ @\ [trn]\ @\ tr1 \wedge never\ \gamma\ tr2 \wedge \gamma\ trn \wedge g\ trn = obs \wedge O\ tr1 = obsl1$

**using** *assms filtermap-eq-Cons unfolding O-def by auto*

**lemma** *O-eq-append*:

**assumes**  $O\ tr = obsl1\ @\ obsl2$

**shows**  $\exists\ tr1\ tr2. tr = tr1\ @\ tr2 \wedge O\ tr1 = obsl1 \wedge O\ tr2 = obsl2$

**using** *assms filtermap-eq-append[of  $\gamma\ g$ ] unfolding O-def by auto*

**lemma** *O-eq-RCons*:  
**assumes**  $O\ tr = ou1\ \#\# \ ou$   
**shows**  $\exists\ trn\ tr1\ tr2. tr = tr1\ @\ [trn]\ @\ tr2 \wedge \gamma\ trn \wedge g\ trn = ou \wedge O\ tr1 = ou1 \wedge never\ \gamma\ tr2$   
**using** *assms filtermap-eq-RCons*[of  $\gamma\ g$ ] **unfolding** *O-def* **by** *blast*

**lemma** *O-eq-Cons-RCons*:  
**assumes**  $O\ tr0 = ou\ \#\ ou1\ \#\# \ ouu$   
**shows**  $\exists\ tr\ trn\ tr1\ trnn\ trr.$   
 $tr0 = tr\ @\ [trn]\ @\ tr1\ @\ [trnn]\ @\ trr \wedge$   
 $never\ \gamma\ tr \wedge \gamma\ trn \wedge g\ trn = ou \wedge O\ tr1 = ou1 \wedge \gamma\ trnn \wedge g\ trnn = ouu \wedge never\ \gamma\ trr$   
**using** *assms filtermap-eq-Cons-RCons*[of  $\gamma\ g$ ] **unfolding** *O-def* **by** *blast*

**lemma** *O-eq-Cons-RCons-append*:  
**assumes**  $O\ tr0 = ou\ \#\ ou1\ \#\# \ ouu\ @\ oull$   
**shows**  $\exists\ tr\ trn\ tr1\ trnn\ trr.$   
 $tr0 = tr\ @\ [trn]\ @\ tr1\ @\ [trnn]\ @\ trr \wedge$   
 $never\ \gamma\ tr \wedge \gamma\ trn \wedge g\ trn = ou \wedge O\ tr1 = ou1 \wedge \gamma\ trnn \wedge g\ trnn = ouu \wedge O\ trr = oull$

**proof**–

**from** *O-eq-append*[of  $tr0\ ou\ \#\ ou1\ \#\# \ ouu\ oull$ ] *assms*  
**obtain**  $tr00\ trrr$  **where**  $1: tr0 = tr00\ @\ trrr$   
**and**  $2: O\ tr00 = ou\ \#\ ou1\ \#\# \ ouu$  **and**  $3: O\ trrr = oull$  **by** *auto*  
**from** *O-eq-Cons-RCons*[OF 2] **obtain**  $tr\ trn\ tr1\ trnn\ trr$  **where**  
 $4: tr00 = tr\ @\ [trn]\ @\ tr1\ @\ [trnn]\ @\ trr \wedge$   
 $never\ \gamma\ tr \wedge$   
 $\gamma\ trn \wedge g\ trn = ou \wedge O\ tr1 = ou1 \wedge \gamma\ trnn \wedge g\ trnn = ouu \wedge never\ \gamma\ trr$  **by** *auto*  
**show** *?thesis* **apply**(*rule exI*[of -  $tr$ ], *rule exI*[of -  $trn$ ], *rule exI*[of -  $tr1$ ],  
*rule exI*[of -  $trnn$ ], *rule exI*[of -  $trr\ @\ trrr$ ])  
**using** 1 3 4 **by** (*simp add: O-append O-Nil-never*)

**qed**

**lemma** *O-Nil-tr-Nil*:  $O\ tr \neq [] \implies tr \neq []$   
**by** (*induction tr*) *auto*

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

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

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

**lemma** *list-ex-length-O*:  
**assumes** *list-ex*  $\gamma\ tr$  **shows**  $length\ (O\ tr) > 0$   
**by** (*metis assms O-Nil-list-ex length-greater-0-conv*)

**lemma** *list-ex-iff-length-O*:  
 $list-ex\ \gamma\ tr \longleftrightarrow length\ (O\ tr) > 0$

**by** (*metis O-Nil-list-ex length-greater-0-conv*)

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

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

**unfolding** *list-ex-iff-length-O* **by** *auto*

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

**using** *O-list-all* **by** *auto*

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

**using** *O-Nil-never* **by** *auto*

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

**using** *V-list-all* **by** *auto*

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

**using** *V-Nil-never* **by** *auto*

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

*Istate*: *reachNT* *istate*

|

*Step*:

$[[reachNT \ (srcOf \ trn); \ validTrans \ trn; \ \neg \ T \ trn]]$

$\implies reachNT \ (tgtOf \ trn)$

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

**using** *assms* **apply** *induct* **by** (*auto* *intro*: *reach.intros*)

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

**by** (*cases*  $\varphi \ trn$ ) *auto*

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

**by** (*cases*  $\varphi \ trn$ ) *auto*

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

**by** (*cases*  $\varphi \ trn$ ) *auto*

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

**by** (*metis* *V-iff-non- $\varphi$*  *V-imp-Nil*)

**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* .

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

**assumes** *reachNT* s

**and** P *istate*

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

**shows** P s

**using** *assms*

**by** (*induction* rule: *reachNT.induct*) (*auto* *elim*: *validTrans.elims*)

**lemma** *reachNT-PairI*:

**assumes** *reachNT* s **and** *step* s a = (ou, s') **and** ¬ T (Trans s a ou s')

**shows** *reachNT* s'

**using** *assms* *reachNT.simps*[of s']

**by** *auto*

**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)

**using** *assms*

**unfolding** *reachNT.simps*[of s]

**by** (*fastforce* *intro*: *reachNT-reach* *elim*: *validTrans.elims*)

**definition** *invarNT* **where**

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

**lemma** *invarNT-disj*:

**assumes** *invarNT* *Inv1* **and** *invarNT* *Inv2*

**shows** *invarNT* (λ s. *Inv1* s ∨ *Inv2* s)

**using** *assms* **unfolding** *invarNT-def* **by** *blast*

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

**lemma** *holdsIstate-invarNT*:  
**assumes** *h: holdsIstate Inv and i: invarNT Inv and a: reachNT s*  
**shows** *Inv s*  
**using** *a using h i unfolding holdsIstate-def invarNT-def*  
**by** (*induction rule: reachNT-step-induct*) *auto*

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

This indeed strengthens the original notion of BD Security.

**lemma** *secureTT-secure*: *secureTT*  $\implies$  *secure*  
**unfolding** *secureTT-def secure-def*  
**by** *blast*

**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*  
**using** *assms unfolding secureTT-def*  
**by** *blast*

**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*  
**using** *assms unfolding secure-def*  
**by** *blast*

**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**  $B = B'$  .

**lemma** *map-Some-these: never Option.is-none xs  $\implies$  map Some (these xs) = xs*

**proof** (*induction xs*)

**case** (*Cons x xs*) **then show** *?case by (cases x) auto*

**qed** *auto*

**lemma** *V'-never-none-T[simp]: Prime.V tr = vl  $\implies$  never Option.is-none vl  $\longleftrightarrow$  never T tr*

**proof** (*induction tr arbitrary: vl*)

**case** (*Cons trn tr*) **then show** *?case by (cases  $\varphi' \text{ trn}$ ) auto*

**qed** *auto*

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

**proof** (*induction tr*)

**case** (*Cons trn tr*) **then show** *?case by (cases  $\varphi' \text{ trn}$ ) auto*

**qed** *auto*

**lemma** *V-Some-never-T: Prime.V tr = map Some vl  $\implies$  never T tr*

**proof** (*induction tr arbitrary: vl*)

**case** (*Cons trn tr*) **then show** *?case by (cases  $\varphi' \text{ trn}$ ) auto*

**qed** *auto*

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*

**unfolding** *secureTT-def secure-def*  
**by** (*auto simp: list-all-iff*)

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**

**assume** *secure: Orig.secureTT*

**then show** *Prime.secure*

**proof** (*unfold Prime.secure-def, intro allI impI, elim conjE*)

**fix** *tr vl vl1*

**assume** *tr: Orig.validFrom istate tr and V: V tr = vl and B: B (these vl) (these vl1)*

**and** *vl: never Option.is-none vl and vl1: never Option.is-none vl1*

**with secure obtain** *tr1 where Orig.validFrom istate tr1 and never T tr1*

**and** *Prime.O tr1 = Prime.O tr and Orig.V tr1 = these vl1*

**by** (*elim Orig.secureTT-E*) (*auto simp: V'-V*)

**then show**  $\exists tr1. Orig.validFrom istate tr1 \wedge O tr1 = O tr \wedge V tr1 = vl1$  **using** *vl1*

**by** (*intro exI[of - tr1]*) (*auto simp: V'-V map-Some-these iff: list-all-iff*)

**qed**

**next**

**assume** *secure': Prime.secure*

**then show** *Orig.secureTT*

**proof** (*unfold Orig.secureTT-def, intro allI impI, elim conjE*)

**fix** *tr vl vl1*

**assume** *Orig.validFrom istate tr and never T tr and B vl vl1 and Orig.V tr = vl*

**with secure' obtain** *tr1 where Orig.validFrom istate tr1 and Prime.O tr1 = Prime.O tr*

**and** *V: Prime.V tr1 = map Some vl1*

**by** (*elim Prime.secure-E*) (*auto iff: V'-V list-all-iff*)

**moreover have** *never T tr1 using V by (intro V-Some-never-T)*

**ultimately show**  $\exists tr1. Orig.validFrom istate tr1 \wedge never T tr1 \wedge O tr1 = O tr \wedge Orig.V tr1 = vl1$

**by** (*intro exI[of - tr1]*) (*auto simp: V'-V*)

**qed**

**qed**

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*

**using** *secureTT-iff-secure' Orig.secureTT-secure*

**by** *simp*

**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**  $B = B^{**}$  .

**lemma** *secureTT-iff-secureTT'*: *Orig.secureTT*  $\longleftrightarrow$  *Prime.secureTT*

**proof**

**assume** *Orig.secureTT*

**then show** *Prime.secureTT*

**proof** (*unfold Prime.secureTT-def, intro allI impI, elim conjE*)

**fix** *tr vl vl1*

**assume** *tr: validSystemTrace tr and TT: TT tr and B: B\*\* vl vl1 and V: V tr = vl*

**from** *B show*  $\exists tr1. \text{validSystemTrace } tr1 \wedge TT \ tr1 \wedge O \ tr1 = O \ tr \wedge V \ tr1 = vl1$

**proof** (*induction rule: rtranclp-induct*)

**case** *base*

**show**  $\exists tr1. \text{validSystemTrace } tr1 \wedge TT \ tr1 \wedge O \ tr1 = O \ tr \wedge V \ tr1 = vl$

**using** *tr TT V*

**by** (*intro exI[where x = tr] auto*)

**next**

**case** (*step vl' vl1'*)

**then obtain** *tr1*

**where** *tr1: validSystemTrace tr1 TT tr1 and O1: O tr1 = O tr and V1: V tr1 = vl'*

**by** *blast*

**show**  $\exists tr1. \text{validSystemTrace } tr1 \wedge TT \ tr1 \wedge O \ tr1 = O \ tr \wedge V \ tr1 = vl1'$

**by** (*rule Orig.secureTT-E[OF <Orig.secureTT> tr1 <B vl' vl1'> V1]*) (*use O1 V in auto*)

**qed**

**qed**

**next**

**assume** *Prime.secureTT*

**then show** *Orig.secureTT*

**unfolding** *Prime.secureTT-def Orig.secureTT-def*

**by** *blast*

**qed**

**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*  $\varphi \ trn$  *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)*  
**unfolding** *consume-def* **by** (*auto split: if-splits*)

**lemma** *ex-consume-φ*:  
**assumes**  $\neg \varphi$  *trn*  
**obtains** *vl'* **where** *consume trn vl vl'*  
**using** *assms unfolding consume-def* **by** *auto*

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

**definition** *iaction* **where**  
*iaction Δ s vl s1 vl1*  $\equiv$   
 $\exists$  *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*  
 $\wedge$   
 $\Delta s vl s1' vl1'$

**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**–  
**have** *al1  $\neq []$  using l by auto*  
**from** *sstep-tgtOf-traceOf[OF this s] assms*  
**show** *?thesis unfolding iaction-def by auto*  
**qed**

**lemma** *sstep-eq-singleiff*[simp]: *sstep s1 [a1] = ([ou1], s1')  $\longleftrightarrow$  step s1 a1 = (ou1, s1')*  
**using** *sstep-Cons* **by** *auto*

**lemma** *iactionI*[intro?]:  
**assumes** *step s1 a1 = (ou1, s1')* **and**  $\varphi$  (*Trans s1 a1 ou1 s1'*)  
**and** *consume (Trans s1 a1 ou1 s1') vl1 vl1'*  
**and**  $\neg \gamma$  (*Trans s1 a1 ou1 s1'*) **and**  $\Delta s vl s1' vl1'$   
**shows** *iaction Δ s vl s1 vl1*  
**using** *assms*  
**by** (*intro iactionI-ms[of - [a1] [ou1]]*) (*auto simp: consume-def consumeList-def*)

**definition** *match where*

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

$\exists al1 vl1'.$

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

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

$O tr1 = O [trn] \wedge$

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

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

**proof**–

**from** *sstep-tgtOf-traceOf*[*OF l s*] *assms*

**show** *?thesis unfolding match-def by (intro exI[of - al1]) auto*

**qed**

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

**using** *assms by (intro matchI-ms[of s1 [a1] [ou1] s1'])*

*(auto simp: consume-def consumeList-def split: if-splits)*

**definition** *ignore where*

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

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

**definition** *reaction where*

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

$\forall a ou s' vl'.$

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

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

$\text{consume } trn vl vl'$

$\longrightarrow$

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

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

**lemma** *reactionI*[*intro?*]:

**assumes**

$\bigwedge a ou s' vl'$ .

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

$\implies$

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

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

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

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

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

**shows** *exit* *s v*

**using** *K unfolding exit-def proof*(*intro allI conjI impI*)

**fix** *tr trn* **assume** *K s* **and** *validFrom* *s* (*tr*  $\#\#$  *trn*)  $\wedge$  *never* *T* (*tr*  $\#\#$  *trn*)  $\wedge$   $\varphi$  *trn*

**thus**  $f trn \neq v$

**using** *I unfolding validFrom-def by* (*induction tr arbitrary: s trn*)

(*auto,metis neq-Nil-conv rotate1.simps(2) rotate1-is-Nil-conv valid-ConsE*)

**qed**

**definition** *noVal* **where**

*noVal* *K v*  $\equiv$

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

**lemma** *noVal-disj*:

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

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

**using** *assms unfolding noVal-def by metis*

**lemma** *noVal-conj*:

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

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

**using** *assms unfolding noVal-def by blast*

**definition** *no $\varphi$*  **where**

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

**lemma** *no $\varphi$ -noVal*: *no $\varphi$*  *K*  $\implies$  *noVal* *K v*

**unfolding** *no $\varphi$ -def noVal-def by auto*

**lemma** *exitI*[*consumes 2, induct pred: exit*]:  
**assumes** *rs: reachNT s and K: K s*  
**and I:**  
 $\bigwedge s a ou s'.$   
 $\llbracket reach\ s; reachNT\ s; step\ s\ a = (ou, s'); K\ s \rrbracket$   
 $\implies (\varphi (Trans\ s\ a\ ou\ s') \longrightarrow f (Trans\ s\ a\ ou\ s') \neq v) \wedge K\ s'$   
**shows** *exit s v*  
**proof**–  
**let**  $?K = \lambda s. reachNT\ s \wedge K\ s$   
**show** *?thesis using assms by (intro exit-coind[of ?K])*  
*(metis reachNT-reach IO-Automaton.validTrans reachNT.Step trans.exhaust-sel)+*  
**qed**

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

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

**lemma** *noVal2-disj*:  
**assumes** *noVal2 Inv1 v and noVal2 Inv2 v*  
**shows** *noVal2 ( $\lambda s v. Inv1\ s\ v \vee Inv2\ s\ v$ ) v*  
**using** *assms unfolding noVal2-def by metis*

**lemma** *noVal2-conj*:  
**assumes** *noVal2 Inv1 v and noVal2 Inv2 v*  
**shows** *noVal2 ( $\lambda s v. Inv1\ s\ v \wedge Inv2\ s\ v$ ) v*  
**using** *assms unfolding noVal2-def by blast*

**lemma** *noVal-noVal2*: *noVal K v  $\implies$  noVal2 ( $\lambda s v. K\ s$ ) v*  
**unfolding** *noVal-def noVal2-def by auto*

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

**shows** *exit s v*

**proof**–

let  $?K = \lambda s. \text{reachNT } s \wedge K s v$

**show** *?thesis using assms by* (intro *exit-coind*[of *?K*])

(metis *reachNT-reach IO-Automaton.validTrans reachNT.Step trans.exhaust-sel*)+

**qed**

**lemma** *exitI2-noVal2*:

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

**and** *invarNT ( $\lambda s. K s v$ ) and noVal2 K v*

**shows** *exit s v*

**proof**–

let  $?K = \lambda s. \text{reachNT } s \wedge K s v$

**show** *?thesis using assms unfolding invarNT-def noVal2-def*

**by** (intro *exit-coind*[of *?K*]) (metis *IO-Automaton.validTrans reachNT.Step trans.exhaust-sel*)+

**qed**

**lemma** *exit-validFrom*:

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

**and** *T: never T tr*

**shows** *False*

**using** *i v V T proof*(induction *tr arbitrary: s*)

case *Nil thus ?case by* (metis *V-simps(1) vl*)

**next**

case (*Cons trn tr s*)

**show** *?case*

**proof**(cases  $\varphi$  *trn*)

case *True*

hence  $f \text{trn} = \text{hd } vl$  **using** *Cons by* (metis *V-simps(3) hd-Cons-tl list.inject vl*)

**moreover** have *validFrom s [trn] using*  $\langle \text{validFrom } s (\text{trn} \# \text{tr}) \rangle$

**unfolding** *validFrom-def by auto*

**ultimately show** *?thesis using Cons True unfolding exit-def*

**by** (*elim allE*[of *- []*]) *auto*

**next**

case *False*

hence  $V \text{tr} = vl$  **using** *Cons by auto*

**moreover** have *never T tr by* (metis *Cons.premis list-all-simps*)

**moreover** from  $\langle \text{validFrom } s (\text{trn} \# \text{tr}) \rangle$  **have** *validFrom (tgtOf trn) tr and s: s = srcOf trn*

**by** (metis *list.distinct(1) validFrom-def valid-ConsE Cons.premis(2)*)

*validFrom-def list.discI list.sel(1)*)+

**moreover** have *exit (tgtOf trn) (hd vl) using*  $\langle \text{exit } s (\text{hd } vl) \rangle$

**unfolding** *exit-def s by simp*

(metis (*no-types*) *Cons.premis(2) Cons.premis(4) append-Cons list.sel(1)*)

*list.distinct list-all-simps valid.Cons validFrom-def valid-ConsE*)

**ultimately show** *?thesis using Cons(1) by auto*

**qed**

**qed**



**definition** *unwind* where

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

**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*  $\Delta$

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

**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**  $\exists tr1. validFrom\ s1\ tr1 \wedge O\ tr1 = O\ tr \wedge V\ tr1 = vl1$

**proof**–

**let**  $?S = \lambda tr\ vl1.$

$\forall s\ vl\ s1. reachNT\ s \wedge reach\ s1 \wedge \Delta\ s\ vl\ s1\ vl1 \wedge validFrom\ s\ tr \wedge never\ T\ tr \wedge V\ tr = vl \longrightarrow$   
 $(\exists tr1. validFrom\ s1\ tr1 \wedge O\ tr1 = O\ tr \wedge V\ tr1 = vl1)$

**let**  $?f = \lambda tr\ vl1. length\ tr + length\ vl1$

**have**  $?S\ tr\ vl1$

**proof**(*induct rule: measure-induct2*[*of ?f ?S*])

**case** (*IH*  $tr\ vl1$ )

**show**  $?case$

**proof**(*intro allI impI, elim conjE*)

**fix**  $s\ vl\ s1$  **assume**  $rs: reachNT\ s$  **and**  $rs1: reach\ s1$  **and**  $\Delta: \Delta\ s\ vl\ s1\ vl1$

**and**  $v: validFrom\ s\ tr$  **and**  $NT: never\ T\ tr$  **and**  $V: V\ tr = vl$

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

$iaction\ \Delta\ s\ vl\ s1\ vl1 \vee$

$(reaction\ \Delta\ s\ vl\ s1\ vl1 \wedge \neg iaction\ \Delta\ s\ vl\ s1\ vl1)$

(**is**  $?exit \vee ?iact \vee ?react \wedge -$ )

**using** *unwind unfolding unwind-def bymetis*

**thus**  $\exists tr1. validFrom\ s1\ tr1 \wedge O\ tr1 = O\ tr \wedge V\ tr1 = vl1$

**proof** *safe*

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

```

hence False using v V exit-validFrom NT by auto
thus ?thesis by auto
next
assume ?iact
thus ?thesis unfolding iaction-def Let-def proof safe
  fix al1 :: 'act list and vl1'
  let ?tr1 = traceOf s1 al1 let ?s1' = tgtOf (last ?tr1)
  assume φ1: list-ex φ (traceOf s1 al1) and c: consumeList ?tr1 vl1 vl1'
    and γ: never γ ?tr1 and Δ: Δ s vl ?s1' vl1'
  from φ1 have tr1: ?tr1 ≠ [] and len-V1: length (V ?tr1) > 0
    by (auto iff: list-ex-iff-length-V)
  with c have length vl1' < length vl1 unfolding consumeList-def by auto
  moreover have reach ?s1' using rs1 tr1 by (intro validFrom-reach) auto
  ultimately obtain tr1' where validFrom ?s1' tr1' and O tr1' = O tr and V tr1' = vl1'
    using IH[of tr vl1'] rs Δ v NT V by auto
  then show ?thesis using tr1 γ c unfolding consumeList-def
    by (intro exI[of - ?tr1 @ tr1'])
      (auto simp: O-append O-Nil-never V-append validFrom-append)
qed
next
assume react: ?react and iact: ¬ ?iact
show ?thesis
proof(cases tr)
  case Nil note tr = Nil
  hence vl: vl = [] using V by simp
  show ?thesis proof(cases vl1)
    case Nil note vl1 = Nil
    show ?thesis using IH[of tr vl1] Δ V NT V unfolding tr vl1 by auto
  next
  case Cons
  hence False using vl unwind rs rs1 Δ iact unfolding unwind-def by auto
  thus ?thesis by auto
qed
next
case (Cons trn tr') note tr = Cons
show ?thesis
proof(cases trn)
  case (Trans ss a ou s') note trn = Trans let ?trn = Trans s a ou s'
  have ss: ss = s using trn v unfolding tr validFrom-def by auto
  have Ta: ¬ T ?trn and s: s = srcOf trn and vtrans: validTrans ?trn
  and v': validFrom s' tr' and NT': never T tr'
  using v NT V unfolding tr validFrom-def trn by auto
  have rs': reachNT s' using rs vtrans Ta by (auto intro: reachNT-PairI)
  {assume φ ?trn hence vl ≠ [] ∧ f ?trn = hd vl using V unfolding tr trn ss by auto
  }
  then obtain vl' where c: consume ?trn vl vl'
  using ex-consume-φ ex-consume-NO by metis
  have V': V tr' = vl' using V c unfolding tr trn ss consume-def
  by (cases φ ?trn) (simp-all, metis list.sel(2-3))

```

```

have match  $\Delta s s1 vl1 a ou s' vl' \vee ignore \Delta s s1 vl1 a ou s' vl'$  (is ?match  $\vee$  ?ignore)
using react unfolding reaction-def using vtrans Ta c by auto
thus ?thesis proof safe
  assume ?match
  thus ?thesis unfolding match-def Let-def proof (elim exE conjE)
    fix al1 :: 'act list and vl1'
    let ?tr = traceOf s1 al1
    let ?s1' = tgtOf (last ?tr)
    assume al1: al1  $\neq$  []
      and c: consumeList ?tr vl1 vl1'
      and O: O ?tr = O [Trans s a ou s']
      and  $\Delta: \Delta s' vl' ?s1' vl1'$ 
    from c have len: length tr' + length vl1' < length tr + length vl1
      using tr unfolding consumeList-def by auto
    have reach ?s1' using rs1 al1 by (intro validFrom-reach) auto
    then obtain tr1' where validFrom ?s1' tr1' and O tr1' = O tr' and V tr1' = vl1'
      using IH[OF len] rs'  $\Delta v' NT' V' tr$  by auto
    then show ?thesis using c O al1 unfolding consumeList-def tr trn ss
      by (intro exI[of - ?tr @ tr1'])
      (cases  $\gamma$  ?trn; auto simp: O-append O-Nil-never V-append validFrom-append)
    qed
  next
    assume ?ignore
    thus ?thesis unfolding ignore-def Let-def proof (elim exE conjE)
      assume  $\gamma: \neg \gamma ?trn$  and  $\Delta: \Delta s' vl' s1 vl1$ 
      obtain tr1 where v1: validFrom s1 tr1 and O: O tr1 = O tr' and V: V tr1 = vl1
      using IH[of tr' vl1] rs' rs1  $\Delta v' NT' V' c$  unfolding tr by auto
      show ?thesis
      apply (intro exI[of - tr1])
      using v1 O V  $\gamma$  unfolding tr trn ss by auto
    qed
  qed
qed
qed
qed
qed
qed
qed
thus ?thesis using assms by auto
qed

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

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

**lemma** *disjAll-simps[simp]*:

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

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

**unfolding** *disjAll-def[abs-def]* **by** *auto*

**lemma** *disjAll-mono*:

**assumes** *disjAll*  $\Delta s s vl s1 vl1$

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

**shows** *disjAll*  $\Delta s' s vl s1 vl1$

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

**lemma** *iaction-mono*:

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

**shows** *iaction*  $\Delta' s vl s1 vl1$

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

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

**using** *assms unfolding match-def* **by** *fastforce*

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

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

**lemma** *reaction-mono*:

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

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

**proof**

**fix** *a ou s' vl'*

**assume** *step* *s a* = (*ou*, *s'*) **and**  $\neg T (Trans\ s\ a\ ou\ s')$  **and** *consume* (*Trans* *s a ou s'*) *vl vl'*

**hence** *match*  $\Delta s s1 vl1 a ou s' vl' \vee ignore\ \Delta s s1 vl1 a ou s' vl'$  (**is** *?m*  $\vee$  *?i*)

**using** *1* **unfolding** *reaction-def* **by** *auto*

**thus** *match*  $\Delta' s s1 vl1 a ou s' vl' \vee ignore\ \Delta' s s1 vl1 a ou s' vl'$  (**is** *?m'*  $\vee$  *?i'*)

**proof**

**assume** *?m* **from** *match-mono[OF this 2]* **show** *?thesis* **by** *simp*

**next**

**assume** *?i* **from** *ignore-mono[OF this 2]* **show** *?thesis* **by** *simp*

**qed**

qed

## 5.2 Decomposition into an arbitrary network of components

**definition** *unwind-to where*

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

**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*  $\Delta \Delta s$

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

**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 ?* $\Delta$ )

**proof**

**fix**  $s \text{ s1} :: \text{'state and } \text{vl } \text{vl1} :: \text{'value list}$

**assume**  $r: \text{reachNT } s \text{ reach } s1$  **and**  $\Delta: ?\Delta s \text{ vl } s1 \text{ vl1}$

**then obtain**  $\Delta$  **where**  $\Delta: \Delta \in \Delta s$  **and**  $2: \Delta s \text{ vl } s1 \text{ vl1}$  **unfolding** *disjAll-def by auto*

**let**  $?\Delta s' = \text{next } \Delta$  **let**  $?\Delta' = \text{disjAll } ?\Delta s'$

**have**  $(\text{vl} \neq [] \wedge \text{exit } s (\text{hd } \text{vl})) \vee$

$\text{iaction } ?\Delta' s \text{ vl } s1 \text{ vl1} \vee$

$((\text{vl} \neq [] \vee \text{vl1} = []) \wedge \text{reaction } ?\Delta' s \text{ vl } s1 \text{ vl1})$

**using**  $2 \Delta \text{ ne } r$  **unfolding** *unwind-to-def by auto*

**moreover have**  $\bigwedge s \text{ vl } s1 \text{ vl1}. ?\Delta' s \text{ vl } s1 \text{ vl1} \implies ?\Delta s \text{ vl } s1 \text{ vl1}$

**using** *ne[OF  $\Delta$ ] unfolding disjAll-def by auto*

**ultimately show**

$(\text{vl} \neq [] \wedge \text{exit } s (\text{hd } \text{vl})) \vee$

$\text{iaction } ?\Delta s \text{ vl } s1 \text{ vl1} \vee$

$((\text{vl} \neq [] \vee \text{vl1} = []) \wedge \text{reaction } ?\Delta s \text{ vl } s1 \text{ vl1})$

**using** *iaction-mono[of  $?\Delta'$  - - -  $?\Delta$ ] reaction-mono[of  $?\Delta'$  - - -  $?\Delta$ ] by blast*

qed

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

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

### 5.3 A customization for linear modular reasoning

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

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

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

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

**shows** *unwind-exit*  $\Delta e$   
**using** *assms unfolding unwind-exit-def* **by** *auto*

**lemma** *unwind-cont-mono*:  
**assumes**  $\Delta s$ : *unwind-cont*  $\Delta \Delta s$   
**and**  $\Delta s'$ :  $\Delta s \subseteq \Delta s'$   
**shows** *unwind-cont*  $\Delta \Delta s'$   
**using**  $\Delta s$  *disjAll-mono*[*OF - \Delta s*] **unfolding** *unwind-cont-def*  
**by** (*auto intro!*: *iaction-mono*[**where**  $\Delta = \text{disjAll } \Delta s$  **and**  $\Delta' = \text{disjAll } \Delta s'$ ]  
*reaction-mono*[**where**  $\Delta = \text{disjAll } \Delta s$  **and**  $\Delta' = \text{disjAll } \Delta s'$ ])

**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**  $\exists \Delta 2. (\Delta, \Delta 2) \in \text{allConsec } \Delta s$   
**using** *assms proof* (*induction*  $\Delta s'$  *arbitrary: \Delta s*)  
**case** *Nil* **thus** ?*case* **by** *auto*  
**next**  
**case** (*Cons*  $\Delta 3 \Delta s' \Delta s$ )  
**show** ?*case* **proof**(*cases*  $\Delta = \Delta 3$ )  
**case** *True*  
**show** ?*thesis* **proof**(*cases*  $\Delta s'$ )  
**case** *Nil*  
**show** ?*thesis* **unfolding**  $\langle \Delta s = (\Delta 3 \# \Delta s') \#\#\Delta 1 \rangle$  *Nil True* **by** (*rule* *exI*[*of - \Delta 1*]) *simp*  
**next**  
**case** (*Cons*  $\Delta 4 \Delta s''$ )  
**show** ?*thesis* **unfolding**  $\langle \Delta s = (\Delta 3 \# \Delta s') \#\#\Delta 1 \rangle$  *Cons True* **by** (*rule* *exI*[*of - \Delta 4*]) *simp*  
**qed**  
**next**  
**case** *False* **hence**  $\Delta \in \text{set } \Delta s'$  **using** *Cons* **by** *auto*  
**then** **obtain**  $\Delta 2$  **where**  $(\Delta, \Delta 2) \in \text{allConsec } (\Delta s' \#\#\Delta 1)$  **using** *Cons* **by** *auto*  
**thus** ?*thesis* **unfolding**  $\langle \Delta s = (\Delta 3 \# \Delta s') \#\#\Delta 1 \rangle$  **by** (*intro* *exI*[*of - \Delta 2*]) (*cases*  $\Delta s'$ , *auto*)  
**qed**  
**qed**

**lemma** *allConsec-set*:  
**assumes**  $(\Delta 1, \Delta 2) \in \text{allConsec } \Delta s$   
**shows**  $\Delta 1 \in \text{set } \Delta s \wedge \Delta 2 \in \text{set } \Delta s$   
**using** *assms* **by** (*induct*  $\Delta s$  *rule: allConsec.induct*) *auto*

**theorem** *unwind-decomp-secure*:  
**assumes**  $n$ :  $\Delta s \neq []$   
**and**  $i$ :  $\bigwedge vl \ vl1. B \ vl \ vl1 \implies hd \ \Delta s \ \text{istate } vl \ \text{istate } vl1$

```

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  $l$ :  $\text{unwind-cont } (\text{last } \Delta s) \{ \text{last } \Delta s, \Delta e \}$ 
and  $e$ :  $\text{unwind-exit } \Delta e$ 
shows secure
proof–
  let  $?\Delta 0 = \text{hd } \Delta s$  let  $?\Delta s = \text{insert } \Delta e (\text{set } \Delta s)$ 
  define next where next  $\Delta 1 =$ 
    (if  $\Delta 1 = \Delta e$  then  $\{ \}$ 
     else if  $\Delta 1 = \text{last } \Delta s$  then  $\{ \Delta 1, \Delta e \}$ 
     else  $\{ \Delta 1, \text{SOME } \Delta 2. (\Delta 1, \Delta 2) \in \text{allConsec } \Delta s, \Delta e \}$ ) for  $\Delta 1$ 
  show ?thesis
  proof(rule unwind-dec-secure)
    show  $? \Delta 0 \in ? \Delta s$  using  $n$  by auto
  next
    fix  $vl \text{ } vl1$  assume  $B \text{ } vl \text{ } vl1$ 
    thus  $? \Delta 0 \text{ } \text{istate } vl \text{ } \text{istate } vl1$  by fact
  next
    fix  $\Delta$ 
    assume  $1$ :  $\Delta \in ? \Delta s$  show  $\text{next } \Delta \subseteq ? \Delta s \wedge \text{unwind-to } \Delta (\text{next } \Delta)$ 
    proof–
      {assume  $\Delta = \Delta e$ 
       hence ?thesis using  $e$  unfolding next-def unwind-exit-def unwind-to-def by auto
      }
      moreover
      {assume  $\Delta = \text{last } \Delta s$  and  $\Delta \neq \Delta e$ 
       hence ?thesis using  $n \text{ } l$  unfolding next-def unwind-cont-def unwind-to-def by simp
      }
      moreover
      {assume  $1$ :  $\Delta \in \text{set } \Delta s$  and  $2$ :  $\Delta \neq \text{last } \Delta s \text{ } \Delta \neq \Delta e$ 
       then obtain  $\Delta' \Delta s'$  where  $\Delta s = \Delta s' \#\#\Delta'$  and  $\Delta$ :  $\Delta \in \text{set } \Delta s'$ 
       by (metis (no-types) append-Cons append-assoc in-set-conv-decomp last-snoc rev-exhaust)
       have  $\exists \Delta 2. (\Delta, \Delta 2) \in \text{allConsec } \Delta s$  using set-allConsec[OF \Delta \Delta s] .
       hence  $(\Delta, \text{SOME } \Delta 2. (\Delta, \Delta 2) \in \text{allConsec } \Delta s) \in \text{allConsec } \Delta s$  by (metis (lifting) someI-ex)
       hence ?thesis using  $1 \text{ } 2 \text{ } c$  unfolding next-def unwind-cont-def unwind-to-def
       by simp (metis (no-types) allConsec-set)
      }
    ultimately show ?thesis using  $1$  by blast
  qed
qed
qed

```

## 5.4 Instances

**corollary** *unwind-decomp3-secure*:

**assumes**

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

**and**  $c1$ :  $\text{unwind-cont } \Delta 1 \{ \Delta 1, \Delta 2, \Delta e \}$

**and**  $c2$ :  $\text{unwind-cont } \Delta 2 \{ \Delta 2, \Delta 3, \Delta e \}$

**and**  $l$ :  $\text{unwind-cont } \Delta 3 \{ \Delta 3, \Delta e \}$



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

**corollary** *unwind-decomp4-secure*:

**assumes**

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

**and**  $c1$ : *unwind-cont*  $\Delta 1$  { $\Delta 1$ ,  $\Delta 2$ ,  $\Delta e$ }

**and**  $c2$ : *unwind-cont*  $\Delta 2$  { $\Delta 2$ ,  $\Delta 3$ ,  $\Delta e$ }

**and**  $c3$ : *unwind-cont*  $\Delta 3$  { $\Delta 3$ ,  $\Delta 4$ ,  $\Delta e$ }

**and**  $l$ : *unwind-cont*  $\Delta 4$  { $\Delta 4$ ,  $\Delta e$ }

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

**shows** *secure*

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

**using** *assms* **by** *auto*

**corollary** *unwind-decomp5-secure*:

**assumes**

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

**and**  $c1$ : *unwind-cont*  $\Delta 1$  { $\Delta 1$ ,  $\Delta 2$ ,  $\Delta e$ }

**and**  $c2$ : *unwind-cont*  $\Delta 2$  { $\Delta 2$ ,  $\Delta 3$ ,  $\Delta e$ }

**and**  $c3$ : *unwind-cont*  $\Delta 3$  { $\Delta 3$ ,  $\Delta 4$ ,  $\Delta e$ }

**and**  $c4$ : *unwind-cont*  $\Delta 4$  { $\Delta 4$ ,  $\Delta 5$ ,  $\Delta e$ }

**and**  $l$ : *unwind-cont*  $\Delta 5$  { $\Delta 5$ ,  $\Delta e$ }

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

**shows** *secure*

**apply**(*rule unwind-decomp-secure*[of [ $\Delta 1$ ,  $\Delta 2$ ,  $\Delta 3$ ,  $\Delta 4$ ,  $\Delta 5$ ]  $\Delta e$ ])

**using** *assms* **by** *auto*

## 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**  $i$ :  $\Delta 0 \in \text{Domain } Gr \wedge \bigwedge vl\ vl1. B\ vl\ vl1 \implies \Delta 0\ ystate\ vl\ ystate\ vl1$

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

**shows** *secure*

**proof** –

**let**  $?pr = \lambda \Delta\ \Delta s. \Delta s \subseteq \text{Domain } Gr \wedge (\Delta, \Delta s) \in Gr$

**define**  $next$  **where**  $next\ \Delta = (\text{SOME } \Delta s. ?pr\ \Delta\ \Delta s)$  **for**  $\Delta$

**let**  $?\Delta s = \text{Domain } Gr$

**show**  $?thesis$

**proof**(*rule unwind-dec-secure*)

**show**  $\Delta 0 \in ?\Delta s$  **using**  $i$  **by** *auto*

**fix**  $vl\ vl1$  **assume**  $B\ vl\ vl1$

**thus**  $\Delta 0\ ystate\ vl\ ystate\ vl1$  **by** *fact*

**next**

**fix**  $\Delta$

**assume**  $\Delta \in ?\Delta s$

**hence**  $?pr \Delta (next \Delta)$  **using**  $n$  *someI-ex*[of  $?pr \Delta$ ] **unfolding** *next-def* **by** *auto*  
**hence**  $next \Delta \subseteq ?\Delta s \wedge (unwind-cont \Delta (next \Delta) \vee unwind-exit \Delta)$  **using**  $c$  **by** *auto*  
**thus**  $next \Delta \subseteq ?\Delta s \wedge unwind-to \Delta (next \Delta)$   
**unfolding** *unwind-to-def unwind-exit-def unwind-cont-def*  
**by** *blast*  
**qed**  
**qed**

## 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.