

Secret-Directed Unwinding

Brijesh Dongol Matt Griffin Andrei Popescu
 Jamie Wright

February 6, 2026

Abstract

This entry formalizes the secret-directed unwinding disproof method for relative security. The method was presented in the CSF 2023 paper “Relative Security: Formally Modeling and (Dis)Proving Resilience Against Semantic Optimization Vulnerabilities” [1]. Secret-directed unwinding can be used to prove the existence of transient execution vulnerabilities.

The main characteristics of secret-directed unwinding are that (1) it is used to disprove rather than prove security and (2) it proceeds in a manner that is “directed” by given sequences of secrets. The second characteristic is shared with the unwinding method for bounded-deducibility security [2].

Contents

| | | |
|----------|--|-----------|
| 1 | Finitary Secret-Directed Unwinding | 1 |
| 2 | Secret-Directed Unwinding | 9 |
| 3 | Secret Directed (Finitary) Unwinding Incompleteness example | 19 |
| 4 | Secret Directed Unwinding Incompleteness example | 25 |

1 Finitary Secret-Directed Unwinding

This theory formalizes the finitary version of secret-directed unwinding, which allows one to disprove finitary relative security.

```
theory SD-Unwinding-fn  
imports Relative-Security.Relative-Security  
begin
```

```
context Rel-Sec  
begin
```

fun *validEtransO* **where** *validEtransO* (*s,secl*) (*s',secl'*) \longleftrightarrow
validTransV (*s,s'*) \wedge
 $(\neg \text{isSecV } s \wedge \text{secl} = \text{secl}' \vee$
 $\text{isSecV } s \wedge \text{secl} = \text{getSecV } s \# \text{secl}')$

definition *move1* Γ *sv1 secl1 sv2 secl2 \equiv
 $\forall \text{sv1}' \text{secl1}'. \text{validEtransO } (\text{sv1},\text{secl1}) (\text{sv1}',\text{secl1}') \longrightarrow \Gamma \text{sv1}' \text{secl1}' \text{sv2 secl2}$*

definition *move2* Γ *sv1 secl1 sv2 secl2 \equiv
 $\forall \text{sv2}' \text{secl2}'. \text{validEtransO } (\text{sv2},\text{secl2}) (\text{sv2}',\text{secl2}') \longrightarrow \Gamma \text{sv1 secl1 sv2}' \text{secl2}'$*

definition *move12* Γ *sv1 secl1 sv2 secl2 \equiv
 $\forall \text{sv1}' \text{secl1}' \text{sv2}' \text{secl2}'.$
 $\text{validEtransO } (\text{sv1},\text{secl1}) (\text{sv1}',\text{secl1}') \wedge \text{validEtransO } (\text{sv2},\text{secl2}) (\text{sv2}',\text{secl2}') \longrightarrow \Gamma \text{sv1}' \text{secl1}' \text{sv2}' \text{secl2}'$*

definition *unwindSDCond* ::
 $(\text{'stateV} \Rightarrow \text{'secret list} \Rightarrow \text{'stateV} \Rightarrow \text{'secret list} \Rightarrow \text{bool}) \Rightarrow \text{bool}$

where

unwindSDCond $\Gamma \equiv \forall \text{sv1 secl1 sv2 secl2}.$

reachV *sv1* \wedge *reachV* *sv2* \wedge

$\Gamma \text{sv1 secl1 sv2 secl2}$

\longrightarrow

$(\text{isIntV } \text{sv1} \longleftrightarrow \text{isIntV } \text{sv2}) \wedge$

$(\neg \text{isIntV } \text{sv1} \longrightarrow \text{move1 } \Gamma \text{sv1 secl1 sv2 secl2} \wedge \text{move2 } \Gamma \text{sv1 secl1 sv2 secl2}) \wedge$

$(\text{isIntV } \text{sv1} \longrightarrow \text{getActV } \text{sv1} = \text{getActV } \text{sv2} \longrightarrow \text{getObsV } \text{sv1} = \text{getObsV } \text{sv2} \wedge$
 $\text{move12 } \Gamma \text{sv1 secl1 sv2 secl2})$

proposition *unwindSDCond-aux*:

assumes *unw*: *unwindSDCond* Γ

and *1*: $\Gamma \text{sv1 secl1 sv2 secl2}$

reachV *sv1* *Van.validFromS* *sv1 trv1 completedFromV* *sv1 trv1*

reachV *sv2* *Van.validFromS* *sv2 trv2 completedFromV* *sv2 trv2*

Van.S *trv1* = *secl1* *Van.S* *trv2* = *secl2*

Van.A *trv1* = *Van.A* *trv2*

shows *Van.O* *trv1* = *Van.O* *trv2*

using *1* **proof**(*induct length trv1 + length trv2 arbitrary: sv1 sv2 trv1 trv2 secl1 secl2 rule: nat-less-induct*)

case (*1 trv1 trv2 sv1 sv2 secl1 secl2*)

show ?*case*

proof(*cases isIntV sv1*)

case *True* **hence** *isIntV sv2* **using** *1(2-)* *unw* **unfolding** *unwindSDCond-def*

by *auto*

note *isv12* = *True* *this*

have $\neg \text{finalV } \text{sv1}$ **using** *isv12* *Van.final-not-isInt* **by** *auto*

then obtain *sv1' trv1'* **where** *trv1*: *trv1* = *sv1* $\#$ *trv1'* *validTransV* (*sv1,sv1'*)

$Van.validFromS\ sv1'\ trv1'\ completedFromV\ sv1'\ trv1'$
using 1 by (*metis* $Van.completed-Cons\ Van.completed-Nil\ Van.validFromS-Cons-iff$
list.exhaust)
have $\neg finalV\ sv2$ **using** $isv12\ Van.final-not-isInt$ **by** *auto*
then obtain $sv2'\ trv2'$ **where** $trv2: trv2 = sv2 \# trv2'\ validTransV\ (sv2,sv2')$

$Van.validFromS\ sv2'\ trv2'\ completedFromV\ sv2'\ trv2'$
using 1 by (*metis* $Van.completed-Cons\ Van.completed-Nil\ Van.validFromS-Cons-iff$
list.exhaust)

define $secl1'$ **where** $secl1' \equiv if\ isSecV\ sv1\ then\ tl\ secl1\ else\ secl1$
have $v1: validEtransO\ (sv1,secl1)\ (sv1',secl1')$
using $trv1$ **unfolding** $validEtransO.simps\ secl1'-def$
using $1\ \langle \neg finalV\ sv1 \rangle$
by (*auto simp add: Van.S-def*)

define $secl2'$ **where** $secl2' \equiv if\ isSecV\ sv2\ then\ tl\ secl2\ else\ secl2$
have $v2: validEtransO\ (sv2,secl2)\ (sv2',secl2')$
using $trv2$ **unfolding** $validEtransO.simps\ secl2'-def$
using $1\ \langle \neg finalV\ sv2 \rangle$ **by** (*auto simp add: Van.S-def*)

have $sl12': secl1' = Van.S\ trv1'\ secl2' = Van.S\ trv2'$
using $1(2-)\ trv1\ trv2$ **unfolding** $secl1'-def\ secl2'-def$ **by** (*auto simp add: Van.S-def*)

have $r12': reachV\ sv1' \wedge reachV\ sv2'$
by (*metis* $1.prem(2)\ 1.prem(5)\ Van.reach.Step\ fst-conv\ snd-conv\ trv1(2)\ trv2(2)$)

have $gasv12: getActV\ sv1 = getActV\ sv2$ **using** $\langle Van.A\ trv1 = Van.A\ trv2 \rangle$
 $isv12$ **unfolding** $trv1\ trv2$
using $1.prem(4)\ 1.prem(7)\ \langle \neg finalV\ sv2 \rangle\ trv1(1)\ trv2(1)$ **by** *auto*
hence $osv12: getObsV\ sv1 = getObsV\ sv2$
using $\langle \Gamma\ sv1\ secl1\ sv2\ secl2 \rangle\ 1.prem(2,5)\ isv12\ unw$ **unfolding** $unwindSD-Cond-def$ **by** *auto*

have $gam: \Gamma\ sv1'\ secl1'\ sv2'\ secl2'$
using $\langle \Gamma\ sv1\ secl1\ sv2\ secl2 \rangle\ v1\ v2\ unw\ r12'\ isv12\ gasv12$ **unfolding** $unwindSD-Cond-def\ move12-def$
using $1(2-)$ **by** *blast*

have $A12: Van.A\ trv1' = Van.A\ trv2'$
using $1.prem(10)\ 1.prem(4)\ 1.prem(7)\ True\ isv12(2)\ trv1(1)\ trv2(1)$ **by** *fastforce*

have $O12': Van.O\ trv1' = Van.O\ trv2'$
apply(*rule* $1(1)[rule-format]$) **using** $trv1\ trv2\ gam\ sl12'\ r12'\ A12$ **by** *auto*

```

thus ?thesis unfolding trv1(1) trv2(1) using isv12 osv12
using 1.premis(4) 1.premis(7) <¬ finalV sv1> <¬ finalV sv2> trv1(1) trv2(1)
by auto
next
  case False hence ¬ isIntV sv2 using 1(2-) unw unfolding unwindSD-
  Cond-def by auto
  note isv12 = False this

show ?thesis
proof(cases length trv1 ≤ 1)
  case True note trv1 = True
  show ?thesis
proof(cases length trv2 ≤ 1)
  case True thus ?thesis
  using One-nat-def Van.O.length-Nil trv1 by presburger
next
  case False
    then obtain sv2' trv2' where trv2: trv2 = sv2 # trv2' validTransV
    (sv2,sv2')
    Van.validFromS sv2' trv2' completedFromV sv2' trv2'
    using <Van.validFromS sv2 trv2> <completedFromV sv2 trv2>
    by (smt (z3) One-nat-def Van.completed-Cons Van.length-toS Van.toS-eq-Nil

      Van.toS-fromS-nonSingl Van.toS-length-gt-eq Van.validFromS-Cons-iff
    diff-Suc-1
      diff-is-0-eq le-SucI length-Suc-conv list.size(3))
    define secl2' where secl2' ≡ if isSecV sv2 then tl secl2 else secl2
    have v2: validEtransO (sv2,secl2) (sv2',secl2')
    using trv2 unfolding validEtransO.simps secl2'-def
    using 1.premis(7) 1.premis(9) Van.final-def by auto
    have sl2': secl2' = Van.S trv2'
    using 1(2-) trv1 trv2 unfolding secl2'-def by auto
    have r2': reachV sv2'
    by (metis 1.premis(5) Van.reach.Step fst-conv snd-conv trv2(2))

    have gam: Γ sv1 secl1 sv2' secl2'
    using <Γ sv1 secl1 sv2 secl2> v2 unw r2' isv12 unfolding unwindSDCond-def
    move2-def
    using 1(2-) by blast

    have A12: Van.A trv1 = Van.A trv2'
    by (simp add: 1.premis(10) isv12(2) trv2(1))

    have O12': Van.O trv1 = Van.O trv2'
    apply(rule 1(1)[rule-format, OF - - gam])
    using trv1 trv2 gam sl2' r2' A12 1(2-) by auto

  thus ?thesis unfolding trv1(1) trv2(1) using isv12 by auto
qed

```

```

next
  case False
    then obtain sv1' trv1' where trv1: trv1 = sv1 # trv1' validTransV
      (sv1,sv1')
      Van.validFromS sv1' trv1' completedFromV sv1' trv1'
      using ⟨Van.validFromS sv1 trv1⟩ ⟨completedFromV sv1 trv1⟩
      by (smt (z3) One-nat-def Simple-Transition-System.validFromS-Cons-iff
        Van.completed-Cons
        completedFromV-def le-SucI length-Suc-conv list.exhaust list.inject list.size(3)
        not-less-eq-eq)

      define secl1' where secl1' ≡ if isSecV sv1 then tl secl1 else secl1
      have v1: validEtransO (sv1,secl1) (sv1',secl1')
      using trv1 unfolding validEtransO.simps secl1'-def
      using 1.premis Van.final-def by auto
      have sl1': secl1' = Van.S trv1'
      using 1(2-) trv1 unfolding secl1'-def by auto
      have r1': reachV sv1'
      by (metis 1.premis(2) Van.reach.Step fst-conv snd-conv trv1(2))

      have gam: Γ sv1' secl1' sv2 secl2
      using ⟨Γ sv1 secl1 sv2 secl2⟩ v1 unw r1' isv12 unfolding unwindSDCond-def
        move1-def
      using 1(2-) by blast

      have A12: Van.A trv1' = Van.A trv2
      using 1.premis(10) isv12(1) trv1(1) by auto

      have O12': Van.O trv1' = Van.O trv2
      apply(rule 1(1)[rule-format, OF - - gam])
      using trv1 gam r1' A12 1(2-) sl1' by auto

      thus ?thesis unfolding trv1(1) using isv12 by auto
      qed
      qed
      qed

```

proposition *unwindSDCond-aux-strong*:
assumes *unw: unwindSDCond Γ*
and *1: Γ sv1 secl1 sv2 secl2*
reachV sv1 Van.validFromS sv1 (trv1 @ [trn1]) never isIntV trv1 **and** *11: isIntV*
trn1 **and**
2: reachV sv2 Van.validFromS sv2 (trv2 @ [trn2]) never isIntV trv2 **and** *22:*
isIntV trn2 **and**

$3: \text{Van.S } trv1 @ ssecl1 = secl1 \text{ Van.S } trv2 @ ssecl2 = secl2$
shows Γ (*lastt sv1 trv1*) *ssecl1* (*lastt sv2 trv2*) *ssecl2*
using 1 2 3 **proof**(*induct length trv1 + length trv2 arbitrary: sv1 sv2 trv1 trv2*
secl1 secl2 rule: nat-less-induct)
case (1 *trv1 trv2 sv1 sv2 secl1 secl2*)
show ?*case*
proof(*cases isIntV sv1*)
case *True* **hence** *sv2: isIntV sv2* **using** 1(2-) *unw* **unfolding** *unwindSD-*
Cond-def **by** *auto*
note *isv12 = True* **this**

have *trv1: trv1 = []* **using** *True* 1(4,5) **unfolding** *list-all-nth* **apply**(*cases*
trv1, auto)
by (*metis Van.validFromS-Cons-iff nth-Cons-0 zero-less-Suc*)
have *trv2: trv2 = []* **using** *sv2* 1(7,8) **unfolding** *list-all-nth* **apply**(*cases trv2,*
auto)
by (*metis Van.validFromS-Cons-iff nth-Cons-0 zero-less-Suc*)

show ?*thesis* **using** 1(2-) **by** (*simp add: trv1 trv2*)
next
case *False* **hence** \neg *isIntV sv2* **using** 1(2-) *unw* **unfolding** *unwindSD-*
Cond-def **by** *auto*
note *isv12 = False* **this**
have *trv1ne: trv1 \neq []* **using** *isv12* 1.prem(3) 11 **by** *force*
then obtain *sv1' trv1'* **where** *trv1: trv1 = sv1 # trv1' validTransV (sv1,sv1')*

Van.validFromS sv1' trv1' never isIntV trv1'
using \langle *Van.validFromS sv1 (trv1 @ [trn1])* \rangle \langle *never isIntV trv1* \rangle
by (*metis Simple-Transition-System.validFromS-Cons-iff Simple-Transition-System.validFromS-def*

Simple-Transition-System.validS-append1
Simple-Transition-System.validS-validTrans hd-append2 list-all-Cons-iff neq-Nil-conv
snoc-eq-iff-butlast)
have *trv2ne: trv2 \neq []* **using** *isv12* 1.prem(6) 22 **by** *force*
then obtain *sv2' trv2'* **where** *trv2: trv2 = sv2 # trv2' validTransV (sv2,sv2')*

Van.validFromS sv2' trv2' never isIntV trv2'
using \langle *Van.validFromS sv2 (trv2 @ [trn2])* \rangle \langle *never isIntV trv2* \rangle
by (*metis Simple-Transition-System.validFromS-Cons-iff Simple-Transition-System.validFromS-def*

Simple-Transition-System.validS-append1
Simple-Transition-System.validS-validTrans hd-append2 list-all-Cons-iff neq-Nil-conv
snoc-eq-iff-butlast)

show ?*thesis*
proof(*cases length trv1 = Suc 0 \wedge length trv2 = Suc 0*)
case *True*
hence *trv12: trv1 = [sv1] \wedge trv2 = [sv2]* **apply**(*intro conjI*)
subgoal **using** 1(4) *Van.validFromS-Cons-iff* **by** (*cases trv1, auto*)

```

    subgoal using 1(7) Van.validFromS-Cons-iff by (cases trv2, auto) .
  show ?thesis using 1(2) 1.prem8,9 trv12 unfolding lastt-def by auto
next
case False note f = False
show ?thesis
proof(cases length trv1 = Suc 0)
  case True
  hence length trv2 > Suc 0 using f trv2ne by (simp add: Suc-lessI)
  hence trv2'ne: trv2' ≠ [] by (simp add: trv2(1))

  define secl2' where secl2' ≡ if isSecV sv2 then tl secl2 else secl2
  have v2: validEtransO (sv2,secl2) (sv2',secl2')
  using trv2 unfolding validEtransO.simps secl2'-def
  using 1.prem9 trv2'ne by auto
  have sl2': secl2' = Van.S trv2' @ ssecl2
  using 1.prem9 trv2(1) v2 by (auto simp: trv2'ne)
  have r2': reachV sv2'
  by (metis 1.prem5) Van.reach.Step fst-conv snd-conv trv2(2))

  have gam: Γ sv1 secl1 sv2' secl2'
  using ⟨Γ sv1 secl1 sv2 secl2⟩ v2 unw r2' isv12 unfolding unwindSDCond-def
  move2-def
  using 1(2-) by blast

  have gam': Γ (lastt sv1 trv1) ssecl1 (lastt sv2' trv2') ssecl2
  apply(rule 1(1)[rule-format, OF - - gam])
  using trv2 gam sl2' ⟨reachV sv1⟩ ⟨reachV sv2⟩ r2'
  using 1.prem3,4,6,8
  by auto (metis Van.validFromS-Cons-iff Van.validFromS-def hd-append
  trv2'ne)

  show ?thesis unfolding trv2 using gam' trv2'ne unfolding lastt-def by
  auto
next
case False
  hence length trv1 > Suc 0 using f trv1ne by (simp add: Suc-lessI)
  hence trv1'ne: trv1' ≠ [] by (simp add: trv1(1))

  define secl1' where secl1' ≡ if isSecV sv1 then tl secl1 else secl1
  have v1: validEtransO (sv1,secl1) (sv1',secl1')
  using trv1 unfolding validEtransO.simps secl1'-def
  using 1.prem8 trv1'ne by auto
  have sl1': secl1' = Van.S trv1' @ ssecl1
  using 1.prem8 trv1(1) v1 by (auto simp: trv1'ne)
  have r1': reachV sv1'
  by (metis 1.prem2) Van.reach.Step fst-conv snd-conv trv1(2))

  have gam: Γ sv1' secl1' sv2 secl2
  using ⟨Γ sv1 secl1 sv2 secl2⟩ v1 unw r1' isv12 unfolding unwindSDCond-def

```

move1-def
using 1(2-) **by** *blast*

have $gam': \Gamma (lastt\ sv1'\ trv1')\ ssecl1\ (lastt\ sv2\ trv2)\ ssecl2$
apply(rule 1(1)[rule-format, OF - - gam])
using $trv1\ gam\ sl1'\ \langle reachV\ sv2 \rangle\ \langle reachV\ sv1 \rangle\ r1'$
using 1.premis
by *auto* (*metis* *Van.validFromS-Cons-iff* *Van.validFromS-def* *hd-append*
trv1'ne)

show ?thesis **unfolding** *trv1* **using** gam' *trv1'ne* **unfolding** *lastt-def* **by**
auto
qed
qed
qed
qed

lemma *S-eq-empty-ConsE*:
assumes $\langle Van.S\ (x\ \#)\ xs = Opt.S\ (y\ \#)\ ys \rangle$ **and** $\langle xs \neq [] \rangle$ **and** $\langle ys \neq [] \rangle$
shows $\langle (isSecO\ y \wedge isSecV\ x \longrightarrow getSecV\ x = getSecO\ y \wedge Van.S\ xs = Opt.S\ ys) \wedge$
 $(isSecO\ y \wedge \neg isSecV\ x \longrightarrow Van.S\ xs = (getSecO\ y\ \#)\ Opt.S\ ys) \wedge$
 $(\neg isSecO\ y \wedge isSecV\ x \longrightarrow (getSecV\ x\ \#)\ Van.S\ xs = Opt.S\ ys) \wedge$
 $(\neg isSecO\ y \wedge \neg isSecV\ x \longrightarrow Van.S\ xs = Opt.S\ ys) \rangle$
using *assms* **unfolding** *Van.S.Cons-unfold* *Opt.S.Cons-unfold*
by (*simp* *split: if-splits*)

theorem *unwindSD-rsecure*:
assumes $tr14: istateO\ s1\ Opt.validFromS\ s1\ tr1\ completedFromO\ s1\ tr1$
 $istateO\ s4\ Opt.validFromS\ s4\ tr2\ completedFromO\ s4\ tr2$
 $Opt.A\ tr1 = Opt.A\ tr2\ Opt.O\ tr1 \neq Opt.O\ tr2$
and *init*: $\bigwedge sv1\ sv2. \llbracket istateV\ sv1; corrState\ sv1\ s1; istateV\ sv2; corrState\ sv2\ s4 \rrbracket \Longrightarrow$
 $\Gamma\ sv1\ (Opt.S\ tr1)\ sv2\ (Opt.S\ tr2)$
and *unw*: *unwindSDCond* Γ
shows $\neg rsecure$
unfolding *rsecure-def2* **unfolding** *not-all not-imp*
apply(rule *exI*[of - *s1*]) **apply**(rule *exI*[of - *tr1*])
apply(rule *exI*[of - *s4*]) **apply**(rule *exI*[of - *tr2*])
apply(rule *conjI*)
subgoal **using** *tr14* **by** (*intro* *conjI*)
subgoal **by** (*metis* *Van.Istate* *init* *unw* *unwindSDCond-aux*) .

end

end

2 Secret-Directed Unwinding

This theory formalizes the secret-directed unwinding disproof method for relative security.

theory *SD-Unwinding*

imports *Relative-Security.Relative-Security*

begin

context *Rel-Sec*

begin

fun *lvalidEtransO* **where** *lvalidEtransO* (*s,secl*) (*s',secl'*) \longleftrightarrow
validTransV (*s,s'*) \wedge
 $(\neg \text{isSecV } s \wedge \text{secl} = \text{secl}' \vee$
 $\text{isSecV } s \wedge \text{secl} = \text{LCons } (\text{getSecV } s) \text{ secl}')$

definition *lmove1* Γ *sv1 secl1 sv2 secl2 \equiv
 $\forall \text{ sv1}' \text{ secl1}'. \text{lvalidEtransO } (\text{sv1}, \text{secl1}) (\text{sv1}', \text{secl1}') \longrightarrow \Gamma \text{ sv1}' \text{ secl1}' \text{ sv2 secl2}$*

definition *lmove2* Γ *sv1 secl1 sv2 secl2 \equiv
 $\forall \text{ sv2}' \text{ secl2}'. \text{lvalidEtransO } (\text{sv2}, \text{secl2}) (\text{sv2}', \text{secl2}') \longrightarrow \Gamma \text{ sv1 secl1 sv2}' \text{ secl2}'$*

definition *lmove12* Γ *sv1 secl1 sv2 secl2 \equiv
 $\forall \text{ sv1}' \text{ secl1}' \text{ sv2}' \text{ secl2}'.$
 $\text{lvalidEtransO } (\text{sv1}, \text{secl1}) (\text{sv1}', \text{secl1}') \wedge \text{lvalidEtransO } (\text{sv2}, \text{secl2}) (\text{sv2}', \text{secl2}') \longrightarrow \Gamma \text{ sv1}' \text{ secl1}' \text{ sv2}' \text{ secl2}'$*

definition *lunwindSDCond* ::
 $(\text{'stateV} \Rightarrow \text{'secret llist} \Rightarrow \text{'stateV} \Rightarrow \text{'secret llist} \Rightarrow \text{bool}) \Rightarrow \text{bool}$

where

lunwindSDCond $\Gamma \equiv \forall \text{ sv1 secl1 sv2 secl2}.$

$\text{reachV } \text{sv1} \wedge \text{reachV } \text{sv2} \wedge$

$\Gamma \text{ sv1 secl1 sv2 secl2}$

\longrightarrow

$(\text{isIntV } \text{sv1} \longleftrightarrow \text{isIntV } \text{sv2}) \wedge$

$(\neg \text{isIntV } \text{sv1} \longrightarrow \text{lmove1 } \Gamma \text{ sv1 secl1 sv2 secl2} \wedge \text{lmove2 } \Gamma \text{ sv1 secl1 sv2 secl2})$

\wedge

$(\text{isIntV } \text{sv1} \wedge \text{getActV } \text{sv1} = \text{getActV } \text{sv2} \longrightarrow \text{getObsV } \text{sv1} = \text{getObsV } \text{sv2} \wedge$
 $\text{lmove12 } \Gamma \text{ sv1 secl1 sv2 secl2})$

lemma *lunwindSDCond-imp*:

assumes *lunwindSDCond* Γ *reachV sv1 reachV sv2* Γ *sv1 secl1 sv2 secl2*

shows

$(\text{isIntV } \text{sv1} \longleftrightarrow \text{isIntV } \text{sv2}) \wedge$

$(\neg \text{isIntV } \text{sv1} \longrightarrow \text{lmove1 } \Gamma \text{ sv1 secl1 sv2 secl2} \wedge \text{lmove2 } \Gamma \text{ sv1 secl1 sv2 secl2})$

\wedge

$(\text{isIntV } \text{sv1} \wedge \text{getActV } \text{sv1} = \text{getActV } \text{sv2} \longrightarrow \text{getObsV } \text{sv1} = \text{getObsV } \text{sv2} \wedge$

$lmove12 \Gamma sv1 secl1 sv2 secl2$)
using *assms* **unfolding** *lunwindSDCond-def* **by** *auto*

lemma *lunwindSDCond-lmove12*:
assumes *unw*: *lunwindSDCond* Γ **and** *gam*: *reachV sv1 reachV sv2* $\Gamma sv1 secl1 sv2 secl2$
and *i*: *isIntV sv1* \longrightarrow *getActV sv1* = *getActV sv2*
shows $lmove12 \Gamma sv1 secl1 sv2 secl2$
proof(*cases isIntV sv1*)
 case *True*
 then show *?thesis* **using** *unw gam i* **unfolding** *lunwindSDCond-def* **by** *blast*
next
 case *False*
 then show *?thesis* **using** *unw gam* **unfolding** *lunwindSDCond-def*
 by (*smt (verit) Van.reach.Step fst-conv lmove12-def lmove1-def lmove2-def*
 lvalidEtransO.simps snd-conv)
qed

proposition *unwindSDCond-aux-inductive*:
assumes *unw*: *lunwindSDCond* Γ
and *1*: $\Gamma sv1 secl1 sv2 secl2$
reachV sv1 Van.validFromS sv1 (trv1 @ [ssv1]) never isIntV trv1 **and** *11*: *isIntV ssv1* **and**
2: *reachV sv2 Van.validFromS sv2 (trv2 @ [ssv2]) never isIntV trv2* **and** *22*: *isIntV ssv2* **and**
3: *lappend (llist-of (Van.S trv1)) ssecl1 = secl1 lappend (llist-of (Van.S trv2)) ssecl2 = secl2*
shows $\Gamma (lastt sv1 trv1) ssecl1 (lastt sv2 trv2) ssecl2$
using *1 2 3* **proof**(*induct length trv1 + length trv2 arbitrary: sv1 sv2 trv1 trv2 secl1 secl2 rule: nat-less-induct*)
 case (*1 trv1 trv2 sv1 sv2 secl1 secl2*)
 show *?case*
 proof(*cases isIntV sv1*)
 case *True* **hence** *sv2: isIntV sv2* **using** *1(2-)* *unw* **unfolding** *lunwindSDCond-def* **by** *auto*
 note *isv12 = True this*

 have *trv1*: *trv1* = [] **using** *True 1(4,5)* **unfolding** *list-all-nth* **apply**(*cases trv1, auto*)
 by (*metis Van.validFromS-Cons-iff nth-Cons-0 zero-less-Suc*)
 have *trv2*: *trv2* = [] **using** *sv2 1(7,8)* **unfolding** *list-all-nth* **apply**(*cases trv2, auto*)
 by (*metis Van.validFromS-Cons-iff nth-Cons-0 zero-less-Suc*)

 show *?thesis* **using** *1(2-)* **by** (*simp add: trv1 trv2 Van.S-def*)
next

case *False* **hence** \neg *isIntV sv2* **using** 1(2-) *unw* **unfolding** *lunwindSD-Cond-def* **by** *auto*
note *isv12 = False this*
have *trv1ne: trv1 \neq []* **using** *isv12 1.prem(3) 11* **by** *force*
then obtain *sv1' trv1'* **where** *trv1: trv1 = sv1 # trv1' validTransV (sv1,sv1')*

Van.validFromS sv1' trv1' never isIntV trv1'
using \langle *Van.validFromS sv1 (trv1 @ [ssv1])* \rangle \langle *never isIntV trv1* \rangle
by (*metis Simple-Transition-System.validFromS-Cons-iff Simple-Transition-System.validFromS-def*)

Simple-Transition-System.validS-append1
Simple-Transition-System.validS-validTrans hd-append2 list-all-Cons-iff neq-Nil-conv
snoc-eq-iff-butlast
have *trv2ne: trv2 \neq []* **using** *isv12 1.prem(6) 22* **by** *force*
then obtain *sv2' trv2'* **where** *trv2: trv2 = sv2 # trv2' validTransV (sv2,sv2')*

Van.validFromS sv2' trv2' never isIntV trv2'
using \langle *Van.validFromS sv2 (trv2 @ [ssv2])* \rangle \langle *never isIntV trv2* \rangle
by (*metis Simple-Transition-System.validFromS-Cons-iff Simple-Transition-System.validFromS-def*)

Simple-Transition-System.validS-append1
Simple-Transition-System.validS-validTrans hd-append2 list-all-Cons-iff neq-Nil-conv
snoc-eq-iff-butlast

show *?thesis*
proof(*cases length trv1 = Suc 0 \wedge length trv2 = Suc 0*)
case *True*
hence *trv12: trv1 = [sv1] \wedge trv2 = [sv2]* **apply**(*intro conjI*)
subgoal **using** 1(4) *Van.validFromS-Cons-iff* **by** (*cases trv1, auto*)
subgoal **using** 1(7) *Van.validFromS-Cons-iff* **by** (*cases trv2, auto*) .
show *?thesis* **using** 1(2) 1.prem(8,9) *trv12* **unfolding** *lastt-def*
using *Van.S.FiltermapBL FiltermapBL.simps(4)* **by** *fastforce*
next
case *False* **note** *f = False*
show *?thesis*
proof(*cases length trv1 = Suc 0*)
case *True*
hence *length trv2 > Suc 0* **using** *f trv2ne* **by** (*simp add: Suc-lessI*)
hence *trv2'ne: trv2' \neq []* **by** (*simp add: trv2(1)*)

define *secl2'* **where** *secl2' \equiv if isSecV sv2 then ltl secl2 else secl2*
have *v2: lvalidEtransO (sv2,secl2) (sv2',secl2')*
using *trv2* **unfolding** *lvalidEtransO.simps secl2'-def*
using 1.prem(9) *trv2'ne* *Van.S.FiltermapBL FiltermapBL.simps(3)* **by**
fastforce
have *sl2': secl2' = lappend (llist-of (Van.S trv2')) ssecl2*
using 1.prem(9) *trv2(1) v2* **by** (*auto simp: trv2'ne*)
have *r2': reachV sv2'*
by (*metis 1.prem(5) Van.reach.Step fst-conv snd-conv trv2(2)*)

```

have gam:  $\Gamma$  sv1 secl1 sv2' secl2'
using  $\langle \Gamma$  sv1 secl1 sv2 secl2  $\rangle$  v2 unv r2' isv12 unfolding lunwindSDCond-def
lmove2-def
using 1(2-) by blast

have gam':  $\Gamma$  (lastt sv1 trv1) ssecl1 (lastt sv2' trv2') ssecl2
apply(rule 1(1)[rule-format, OF - - gam])
using trv2 gam sl2'  $\langle$ reachV sv1  $\rangle$   $\langle$ reachV sv2  $\rangle$  r2'
using 1.premis(3,4,6,8)
by auto (metis Van.validFromS-Cons-iff Van.validFromS-def hd-append
trv2'ne)

show ?thesis unfolding trv2 using gam' trv2'ne unfolding lastt-def by
auto
next
case False
hence length trv1 > Suc 0 using f trv1ne by (simp add: Suc-lessI)
hence trv1'ne: trv1'  $\neq$  [] by (simp add: trv1(1))

define secl1' where secl1'  $\equiv$  if isSecV sv1 then ltl secl1 else secl1
have v1: lvalidEtransO (sv1,secl1) (sv1',secl1')
using trv1 unfolding lvalidEtransO.simps secl1'-def
using 1.premis(8) trv1'ne Van.S.FiltermapBL FiltermapBL.Cons-unfold by
fastforce
have sl1': secl1' = lappend (llist-of (Van.S trv1')) ssecl1
using 1.premis(8) trv1(1) v1 by (auto simp: trv1'ne)
have r1': reachV sv1'
by (metis 1.premis(2) Van.reach.Step fst-conv snd-conv trv1(2))

have gam:  $\Gamma$  sv1' secl1' sv2 secl2
using  $\langle \Gamma$  sv1 secl1 sv2 secl2  $\rangle$  v1 unv r1' isv12 unfolding lunwindSDCond-def
lmove1-def
using 1(2-) by blast

have gam':  $\Gamma$  (lastt sv1' trv1') ssecl1 (lastt sv2 trv2) ssecl2
apply(rule 1(1)[rule-format, OF - - gam])
using trv1 gam sl1'  $\langle$ reachV sv2  $\rangle$   $\langle$ reachV sv1  $\rangle$  r1'
using 1.premis
by auto (metis Van.validFromS-Cons-iff Van.validFromS-def hd-append
trv1'ne)

show ?thesis unfolding trv1 using gam' trv1'ne unfolding lastt-def by
auto
qed
qed
qed
qed

```

proposition *unwindSDCond-inductive*:
assumes *unw*: *lunwindSDCond* Γ
and *gam*: Γ *sv1* *secl1* *sv2* *secl2* **and**
trv1: *reachV* *sv1* *Van.validFromS* *sv1* (*trv1* @ [*ssv1*]) *never isIntV* *trv1* *isIntV* *ssv1*
and
trv2: *reachV* *sv2* *Van.validFromS* *sv2* (*trv2* @ [*ssv2*]) *never isIntV* *trv2* *isIntV* *ssv2*
and
s: *lappend* (*l*list-of (*map* *getSecV* (*filter* *isSecV* *trv1*))) *ssecl1* = *secl1*
lappend (*l*list-of (*map* *getSecV* (*filter* *isSecV* *trv2*))) *ssecl2* = *secl2*
shows (*getActV* *ssv1* = *getActV* *ssv2* \longrightarrow Γ *ssv1* *ssecl1* *ssv2* *ssecl2*) \wedge
(*getActV* *ssv1* = *getActV* *ssv2* \longrightarrow *getObsV* *ssv1* = *getObsV* *ssv2*)
proof –
define *ssecl1'* **where** *ssecl1'* = (*if* *trv1* = [] \vee (*trv1* \neq [] \wedge \neg *isSecV* (*last* *trv1*))
then *ssecl1* else (*getSecV* (*last* *trv1*)) \$ *ssecl1*)
define *ssecl2'* **where** *ssecl2'* = (*if* *trv2* = [] \vee (*trv2* \neq [] \wedge \neg *isSecV* (*last* *trv2*))
then *ssecl2* else (*getSecV* (*last* *trv2*)) \$ *ssecl2*)
have *s'*: *lappend* (*l*list-of (*Van.S* *trv1*)) *ssecl1'* = *secl1*
lappend (*l*list-of (*Van.S* *trv2*)) *ssecl2'* = *secl2*
subgoal unfolding *ssecl1'*-def *s*[*symmetric*] *Van.S.map-filter*
by *simp* (*metis* *List-Filtermap.filtermap-def* *filtermap-butlast* *holds-filtermap-RCons*
*lappend-l*list-of-*LCons* *snoc-eq-iff-butlast*)
subgoal unfolding *ssecl2'*-def *s*[*symmetric*] *Van.S.map-filter*
by *simp* (*metis* *List-Filtermap.filtermap-def* *append-butlast-last-id* *filtermap-butlast*
holds-filtermap-RCons *lappend-l*list-of-*LCons*) .

have *gg*: Γ (*lastt* *sv1* *trv1*) *ssecl1'* (*lastt* *sv2* *trv2*) *ssecl2'*
using *unwindSDCond-aux-inductive*[*OF* *unw* *gam* *trv1* *trv2* *s'*] .
have *rsv1*: *reachV* (*lastt* *sv1* *trv1*)
by (*metis* *Van.reach-validFromS-reach* *Van.validFromS-def* *Van.validS-append1*
append-is-Nil-conv *assms*(3) *assms*(4) *hd-append* *lastt-def*)
have *rsv2*: *reachV* (*lastt* *sv2* *trv2*)
by (*metis* *Van.lvalidFromS-lappend-finite* *Van.lvalidFromS-l*list-of-*validFromS*
Van.reach-validFromS-reach *assms*(7) *assms*(8) *lappend-l*list-of-*l*list-of-*lastt-def*)

have *trv1* = [] \longleftrightarrow *trv2* = [] **using** *trv1*(2–4) *trv2*(2–4) **unfolding** *list-all-nth*

apply *safe*
subgoal by *simp* (*smt* (*verit*) *Simple-Transition-System.validFromS-def* *assms*(3)
assms(7) *gam*
hd-append *hd-conv-nth* *length-greater-0-conv* *lunwindSDCond-def* *snoc-eq-iff-butlast*
unw)
subgoal by *simp* (*smt* (*verit*) *Simple-Transition-System.validFromS-def* *assms*(3)
assms(7) *gam*
hd-append *hd-conv-nth* *length-greater-0-conv* *lunwindSDCond-def* *snoc-eq-iff-butlast*
unw) .

hence *ddd*: (*trv1* = [] \wedge *trv2* = []) \vee (*trv1* \neq [] \wedge *trv2* \neq []) **by** *blast*

show *?thesis*

```

using ddd proof(elim conjE disjE)
  assume trv12: trv1 = [] trv2 = []
  hence sv12: ssv1 = lastt sv1 trv1 ssv2 = lastt sv2 trv2 and sv12': ssv1 = sv1
  ssv2 = sv2
  and secl: ssecl1 = ssecl1' ssecl2 = ssecl2'
  using trv1(2) trv2(2) ssecl1'-def ssecl2'-def by auto

  show ?thesis proof safe
    show g:  $\Gamma$  ssv1 ssecl1 ssv2 ssecl2 unfolding sv12 secl using gg .
    assume getActV ssv1 = getActV ssv2
    thus getObsV ssv1 = getObsV ssv2
    using lunwindSDCond-imp[OF unsw rsv1 rsv2, unfolded sv12[symmetric], OF
g] trv1(4) by auto
    qed
  next
    assume trv12: trv1  $\neq$  [] trv2  $\neq$  []
    have n:  $\neg$  isIntV (last trv1)
      by (metis append-butlast-last-id list.pred-inject(2) list-all-append trv1(3)
trv12(1))
    have v1: validTransV (lastt sv1 trv1, ssv1)
    by (metis Van.validFromS-def Van.validS-validTrans lastt-def list.sel(1) not-Cons-self
snoc-eq-iff-butlast trv1(2) trv12(1))
    have v2: validTransV (lastt sv2 trv2, ssv2)
    by (metis Nil-is-append-conv Van.validFromS-def Van.validS-validTrans lastt-def
list.discI list.sel(1) trv12(2) trv2(2))
    show ?thesis proof safe
      assume gssv12: getActV ssv1 = getActV ssv2
      show g:  $\Gamma$  ssv1 ssecl1 ssv2 ssecl2
      apply(rule lunwindSDCond-lmove12[OF unsw rsv1 rsv2 gg, unfolded lmove12-def,
rule-format])
      unfolding lvalidEtransO.simps ssecl1'-def ssecl2'-def
      using trv12 v1 v2 n by (auto simp: lastt-def)

      show getObsV ssv1 = getObsV ssv2 using gssv12
      using lunwindSDCond-imp[OF unsw - - g]
      by (metis Van.reach.Step fst-conv rsv1 rsv2 snd-conv trv1(4) v1 v2)
    qed
  qed
qed

```

proposition lunwindSDCond-aux:
assumes unsw: lunwindSDCond Γ
and 1: Γ sv1 secl1 sv2 secl2
reachV sv1 Van.lvalidFromS sv1 trv1 lcompletedFromV sv1 trv1
reachV sv2 Van.lvalidFromS sv2 trv2 lcompletedFromV sv2 trv2
Van.lS trv1 = secl1 Van.lS trv2 = secl2
Van.lA trv1 = Van.lA trv2
shows Van.lO trv1 = Van.lO trv2
proof–

{fix *obl1 obl2*
assume $\exists sv1\ trv1\ secl1\ sv2\ trv2\ secl2. obl1 = Van.lO\ trv1 \wedge obl2 = Van.lO\ trv2 \wedge$
 $\Gamma\ sv1\ secl1\ sv2\ secl2 \wedge$
 $reachV\ sv1 \wedge Van.lvalidFromS\ sv1\ trv1 \wedge lcompletedFromV\ sv1\ trv1 \wedge$
 $reachV\ sv2 \wedge Van.lvalidFromS\ sv2\ trv2 \wedge lcompletedFromV\ sv2\ trv2 \wedge$
 $Van.lS\ trv1 = secl1 \wedge Van.lS\ trv2 = secl2 \wedge Van.lA\ trv1 = Van.lA\ trv2$
hence *obl1 = obl2*
proof (*coinduct rule: llist.coinduct*)
case (*Eq-llist obl1 obl2*)
then obtain *sv1 trv1 secl1 sv2 trv2 secl2* **where** *obl: obl1 = Van.lO trv1 obl2 = Van.lO trv2*
and *gam: $\Gamma\ sv1\ secl1\ sv2\ secl2$*
and *trv1: $reachV\ sv1\ Van.lvalidFromS\ sv1\ trv1\ lcompletedFromV\ sv1\ trv1$*
and *trv2: $reachV\ sv2\ Van.lvalidFromS\ sv2\ trv2\ lcompletedFromV\ sv2\ trv2$*
and *Str: $Van.lS\ trv1 = secl1\ Van.lS\ trv2 = secl2$* **and** *Atr: $Van.lA\ trv1 = Van.lA\ trv2$*
by *blast*
show *?case proof(intro conjI impI)*
show *lnull: lnull obl1 = lnull obl2*
using *obl Atr unfolding lnull-def*
by (*metis LNil-eq-lmap Van.lA.lmap-lfilter Van.lO.lmap-lfilter*)

assume *ln: $\neg\ lnull\ obl1 \neg\ lnull\ obl2$*
then obtain *ob1 obl1' ob2 obl2'* **where**
obl1: $obl1 = LCons\ ob1\ obl1'$ **and** *obl2: $obl2 = LCons\ ob2\ obl2'$*
unfolding *lnull-def* **using** *llist.exhaust-sel* **by** *blast*
hence *lhd: $lhd\ obl1 = ob1\ lhd\ obl2 = ob2$*
and *ltl: $ltl\ obl1 = obl1'\ ltl\ obl2 = obl2'$*
by *auto*

obtain *ftrv1 sv1' trv1'* **where**
trv1-eq: $trv1 = lappend\ (llist-of\ ftrv1)\ (sv1'\ \$\ trv1')$ **and** *ftrv1a: never isIntV ftrv1*
and *sv1': $isIntV\ sv1'\ getObsV\ sv1' = ob1$* **and** *trv1': $Van.lO\ trv1' = obl1'$*
using *Van.lO.eq-LCons[OF obl(1)[symmetric, unfolded obl1]]* **by** *auto*
define *sv11* **where** *sv11 = lastt sv1 ftrv1*
have *trv11': $Van.lvalidFromS\ sv1'\ (sv1'\ \$\ trv1')$*
and *ftrv1b: $Van.validFromS\ sv1\ ftrv1$*
*(ftrv1 = [] $\wedge\ sv1 = sv1' \wedge sv11 = sv1$) \vee (ftrv1 \neq [] $\wedge\ validTransV\ (sv11,$
*sv1'))
using *trv1(2)*
unfolding *trv1-eq* **unfolding** *Van.lvalidFromS-lappend-LCons*
unfolding *lastt-def sv11-def* **by** *auto*
note *ftrv1 = ftrv1a ftrv1b*
have *fftrv1: filter isIntV ftrv1 = []* **by** (*metis ftrv1(1) never-Nil-filter*)
have *ftrv1c: $Van.validFromS\ sv1\ (ftrv1\ @\ [sv1'])$*

by (*metis Van.lvalidFromS-lappend-finite lappend-llist-of-LCons trv1(2) trv1-eq*)**

```

define ssv1' where ssv1'  $\equiv$  if trv1' = [] then sv1' else lhd trv1'

obtain ftrv2 sv2' trv2' where
trv2-eq: trv2 = lappend (llist-of ftrv2) (sv2' $ trv2') and ftrv2a: never isIntV
ftrv2
and sv2': isIntV sv2' getObsV sv2' = ob2 and trv2': Van.lO trv2' = obl2'
using Van.lO.eq-LCons[OF obl(2)][symmetric, unfolded obl2]] by auto
define sv22 where sv22 = lastt sv2 ftrv2
have trv22': Van.lvalidFromS sv2' (sv2' $ trv2')
and ftrv2b: Van.validFromS sv2 ftrv2
(ftrv2 = []  $\wedge$  sv2 = sv2'  $\wedge$  sv22 = sv2)  $\vee$  (ftrv2  $\neq$  []  $\wedge$  validTransV (sv22,
sv2'))
using trv2(2)
unfolding trv2-eq unfolding Van.lvalidFromS-lappend-LCons
unfolding lastt-def sv22-def by auto
note ftrv2 = ftrv2a ftrv2b
have fftrv2: filter isIntV ftrv2 = [] by (metis ftrv2(1) never-Nil-filter)
have ftrv2c: Van.validFromS sv2 (ftrv2 @ [sv2'])
by (metis Van.lvalidFromS-lappend-finite lappend-llist-of-LCons trv2(2) trv2-eq)

define ssv2' where ssv2'  $\equiv$  if trv2' = [] then sv2' else lhd trv2'
have rsv1': reachV sv1'
by (metis Van.reach.Step Van.reach-validFromS-reach
fst-conv ftrv1b(1) ftrv1b(2) lastt-def sv11-def snd-conv trv1(1))
have rsv2': reachV sv2'
by (metis Van.reach.Step Van.reach-validFromS-reach fst-conv ftrv2b(1)
ftrv2b(2) lastt-def sv22-def snd-conv trv2(1))

have rsv11: reachV sv11
by (metis Van.reach-validFromS-reach ftrv1b(1) lastt-def sv11-def trv1(1))
have rsv22: reachV sv22
by (metis Van.reach-validFromS-reach ftrv2b(1) lastt-def sv22-def trv2(1))

define secl1' secl2' where secl1': secl1'  $\equiv$  Van.lS trv1' and secl2': secl2'  $\equiv$ 
Van.lS trv2'

define ssecl1' where ssecl1'  $\equiv$  Van.lS (sv1' $ trv1')
define ssecl2' where ssecl2'  $\equiv$  Van.lS (sv2' $ trv2')

have trv12'ne: trv1'  $\neq$  []  $\wedge$  trv2'  $\neq$  []
by (metis Van.lO.lmap-lfilter Van.lO.simps(4) fftrv1 fftrv2 lappend-LNil2
lbutlast-lappend
lbutlast-singl lfilter-LNil lfilter-llist-of llist.distinct(1) llist-of.simps(1) obl(1)
obl(2) obl1 obl2 trv1-eq trv2-eq)

have gasv12': getActV sv1' = getActV sv2' using Atr trv12'ne unfolding
trv1-eq trv2-eq

```

unfolding *Van.lA.lmap-lfilter*
using *fftrv1 fftrv2 sv1'(1) sv2'(1)*
by (*auto simp: lbutlast-lappend lmap-lappend-distrib lappend-eq-LNil-iff split: if-splits*)

have *ggam-gao: (getActV sv1' = getActV sv2' \longrightarrow Γ sv1' ssecl1' sv2' ssecl2')*
 \wedge (*getActV sv1' = getActV sv2' \longrightarrow getObsV sv1' = getObsV sv2'*)
apply(*rule unwindSDCond-inductive[OF unw gam*
trv1(1) ftrv1c ftrv1(1) sv1'(1)
trv2(1) ftrv2c ftrv2(1) sv2'(1)
])
subgoal unfolding *Str(1)[symmetric]* **unfolding** *trv1-eq*
unfolding *ssecl1'-def sv11-def Van.S.map-filter Van.lS.lmap-lfilter*
by (*auto simp: lastt-def lbutlast-lappend lmap-lappend-distrib lappend-eq-LNil-iff split: if-splits*)
subgoal unfolding *Str(2)[symmetric]* **unfolding** *trv2-eq*
unfolding *ssecl2'-def sv11-def Van.S.map-filter Van.lS.lmap-lfilter*
by (*auto simp: lastt-def lbutlast-lappend lmap-lappend-distrib lappend-eq-LNil-iff*)

note *ggam = ggam-gao[THEN conjunct1, rule-format, OF gasv12']* **note**
gao = ggam-gao[THEN conjunct2, rule-format, OF gasv12']

have *getObsV sv1' = getObsV sv2' using gao gasv12' by simp*

thus *lhd obl1 = lhd obl2*
unfolding *lhd sv1'(2)[symmetric] sv2'(2)[symmetric]* .

show \exists *sv1 trv1 secl1 sv2 trv2 secl2*.
ltl obl1 = Van.lO trv1 \wedge ltl obl2 = Van.lO trv2 \wedge
 Γ *sv1 secl1 sv2 secl2 \wedge*
reachV sv1 \wedge Van.lvalidFromS sv1 trv1 \wedge lcompletedFromV sv1 trv1 \wedge
reachV sv2 \wedge Van.lvalidFromS sv2 trv2 \wedge lcompletedFromV sv2 trv2 \wedge
Van.lS trv1 = secl1 \wedge Van.lS trv2 = secl2 \wedge Van.lA trv1 = Van.lA trv2
proof(*intro exI[of - ssv1'] exI[of - trv1'], rule exI[of - secl1'],*
intro exI[of - ssv2'] exI[of - trv2'], rule exI[of - secl2'],
intro conjI)
show *reachV ssv1'*
unfolding *ssv1'-def using rsv1' apply(cases trv1' = [], simp-all)*
by (*metis Van.lvalidFromS-Cons-iff Van.reach.simps fst-conv snd-conv trv11'*)
show *reachV ssv2'*
unfolding *ssv2'-def using rsv2' apply(cases trv2' = [], simp-all)*
by (*metis Van.lvalidFromS-Cons-iff Van.reach.simps fst-conv snd-conv trv22'*)

show *ltl obl1 = Van.lO trv1' unfolding ltl trv1' ..*
show *ltl obl2 = Van.lO trv2' unfolding ltl trv2' ..*

show *Van.lvalidFromS ssv1' trv1' using trv11' Van.lvalidFromS-Cons-iff*
ssv1'-def by auto
show *lc1': lcompletedFromV ssv1' trv1' using trv1(3) unfolding trv1-eq*

```

unfolding Van.lcompletedFrom-def
by (metis lfinite-code(2) lfinite-lappend lfinite-llist-of llast-LCons2
llast-lappend-LCons llast-last-llist-of llist.exhaust-sel trv12'ne)
show Van.lvalidFromS ssv2' trv2' using trv22' Van.lvalidFromS-Cons-iff
ssv2'-def by auto
show lc2': lcompletedFromV ssv2' trv2' using trv2(3) unfolding trv2-eq
unfolding Van.lcompletedFrom-def
by (metis lfinite-code(2) lfinite-lappend lfinite-llist-of llast-LCons2 llast-lappend-LCons
llast-last-llist-of llist.exhaust-sel trv12'ne)

show Van.lA trv1' = Van.lA trv2'
using Atr unfolding trv1-eq trv2-eq using ftrv1(1) ftrv2(1) sv1'(1) sv2'(1)
unfolding Van.lA.lmap-lfilter
by (simp add: ftrv1 ftrv2 lbutlast-lappend trv12'ne)

show Van.lS trv1' = secl1' unfolding secl1' ..
show Van.lS trv2' = secl2' unfolding secl2' ..

show  $\Gamma$  ssv1' secl1' ssv2' secl2'
apply(rule lunwindSDCond-lmove12[OF unW rsv1' rsv2' ggam, unfolded
lmove12-def, rule-format, OF gasv12\uparrow])
apply(rule conjI)
subgoal unfolding lvalidEtransO.simps apply(rule conjI)
subgoal using trv12'ne Van.lvalidFromS-Cons-iff trv11' unfolding
ssv1'-def by auto
subgoal using trv12'ne unfolding ssecl1'-def secl1' by (auto simp:
Van.lS.lmap-lfilter) .
subgoal unfolding lvalidEtransO.simps apply(rule conjI)
subgoal using trv12'ne Van.lvalidFromS-Cons-iff trv22' unfolding
ssv2'-def by auto
subgoal using trv12'ne unfolding ssecl2'-def secl2' by (auto simp:
Van.lS.lmap-lfilter) . .
qed
qed
qed
}
thus ?thesis using assms by blast
qed

theorem unwindSD-lrsecure:
assumes tr14: istateO s1 Opt.lvalidFromS s1 tr1 lcompletedFromO s1 tr1
istateO s2 Opt.lvalidFromS s2 tr2 lcompletedFromO s2 tr2
Opt.lA tr1 = Opt.lA tr2 Opt.lO tr1  $\neq$  Opt.lO tr2
and init:  $\bigwedge$ sv1 sv2. istateV sv1  $\implies$  corrState sv1 s1  $\implies$  istateV sv2  $\implies$  corrState
sv2 s2  $\implies$ 
 $\Gamma$  sv1 (Opt.lS tr1) sv2 (Opt.lS tr2)
and unW: lunwindSDCond  $\Gamma$ 
shows  $\neg$  lrsecure
unfolding lrsecure-def2 unfolding not-all not-imp

```

```

apply(rule exI[of - s1]) apply(rule exI[of - tr1])
apply(rule exI[of - s2]) apply(rule exI[of - tr2])
apply(rule conjI)
  subgoal using tr1 by auto
  subgoal unfolding not-ex apply safe
  subgoal for sv1 trv1 sv2 trv2 apply(erule cnf.clause2raw-notE[of Van.lO trv1
  ≠ Van.lO trv2], simp)
  apply(rule lunwindSDCond-aux[OF unw, OF init])
  using Van.Istate by auto . .

end

end

```

3 Secret Directed (Finatary) Unwinding Incompleteness example

Demonstrating a counterexample which is secure but fails in the finatary unwinding

```

theory SD-Incomplete-fin
  imports SD-Unwinding-fin
begin

no-notation bot ( $\perp$ )

abbreviation noninform ( $\perp$ ) where  $\perp \equiv$  undefined

datatype State = s0 | s0' | s1 | s1' | s2
type-synonym secret = State

fun transit::State  $\Rightarrow$  State  $\Rightarrow$  bool(infix  $\rightarrow I$  55) where
  transit s s' =
    (if (s = s0  $\wedge$  s' = s1)  $\vee$ 
      (s = s1  $\wedge$  s' = s2)  $\vee$ 
      (s = s0'  $\wedge$  s' = s1')  $\vee$ 
      (s = s1'  $\wedge$  s' = s2) then True
      else False)

lemma transit-s0-s1:s0  $\rightarrow I$  s1 by auto
lemma transit-s1-s2:s1  $\rightarrow I$  s2 by auto

```

lemma *transit-s0'-s1':s0' →I s1'* **by** *auto*

lemma *transit-s1'-s2:s1' →I s2* **by** *auto*

lemma *transit-iff:s →I s' ↔ (s = s0 ∧ s' = s1) ∨
(s = s1 ∧ s' = s2) ∨
(s = s0' ∧ s' = s1') ∨
(s = s1' ∧ s' = s2)* **by** *auto*

definition *final x ≡ ∀ y. ¬ (→I) x y*

lemma *final-s2[simp]:final s2 unfolding final-def transit-iff* **by** *auto*

lemma *final-iff: final s ↔ s = s2 unfolding final-def transit-iff* **by** (*auto,metis State.exhaust*)

Vanilla-semantics system model

type-synonym *stateV = State*

fun *validTransV* **where** *validTransV (s,s') = s →I s'*

Secrets at initial states, interaction at everywhere besides final (i.e. s2)

fun *isIntV :: stateV ⇒ bool* **where** *isIntV s = (s ≠ s2)*

fun *getIntV :: stateV ⇒ nat × nat* **where**

getIntV s =

(case s of

s0 ⇒ (1, 0)

|s0' ⇒ (1, 1)

|s1 ⇒ (0, ⊥)

|s1' ⇒ (1, ⊥)

|_ ⇒ (⊥, ⊥)

)

definition *isSecV :: stateV ⇒ bool* **where** *isSecV s = (s ∈ {s0, s0'})*

fun *getSecV :: stateV ⇒ secret* **where** *getSecV s = s*

lemma *getSecV-neq: getSecV s0 ≠ getSecV s0'* **by** *auto*

The optimization-enhanced system model

type-synonym *stateO = State*

fun *validTransO* **where** *validTransO (s,s') = s →I s'*

Secrets and interaction at initial states

fun *isIntO :: stateO ⇒ bool* **where** *isIntO s = (s ∈ {s0, s0'})*

```

fun getIntO :: stateO ⇒ nat × nat where
getIntO s =
  (case s of
    s0 ⇒ (1, 0)
  | s0' ⇒ (1, 1)
  | - ⇒ (⊥, ⊥)
  )

```

definition isSecO :: stateO ⇒ bool **where** isSecO s = (s ∈ {s₀, s₀'})

lemma isSecO[simp]: isSecO s₀ **unfolding** isSecO-def **by** auto

lemma isSec1[simp]: ¬isSecO s₁ **unfolding** isSecO-def **by** auto

lemma isSec0'[simp]: isSecO s₀' **unfolding** isSecO-def **by** auto

lemma isSec1'[simp]: ¬isSecO s₁' **unfolding** isSecO-def **by** auto

fun getSecO :: stateO ⇒ secret **where** getSecO s = s

corrState

fun corrState :: stateV ⇒ stateO ⇒ bool **where**

corrState cfgO cfgA = True

interpretation Rel-Sec

where validTransV = validTransV **and** istateV = λs. s = s₀ ∨ s = s₀'

and finalV = final

and isSecV = isSecV **and** getSecV = getSecV

and isIntV = isIntV **and** getIntV = getIntV

and validTransO = validTransO **and** istateO = λs. s = s₀ ∨ s = s₀'

and finalO = final

and isSecO = isSecO **and** getSecO = getSecO

and isIntO = isIntO **and** getIntO = getIntO

and corrState = corrState

apply(unfold-locales)

subgoal **by** (auto simp: final-def)

subgoal **by** (auto simp: final-iff)

subgoal **by** (auto simp: final-iff isSecV-def)

subgoal **by** (auto simp: final-def)

subgoal **by** (auto simp: final-iff)

subgoal **by** (auto simp: final-iff isSecO-def) .

lemma getAct0:getActO s₀ = getActO s₀' **unfolding** Opt.getAct-def **by** auto

lemma getObs0:getObsO s₀ ≠ getObsO s₀' **unfolding** Opt.getObs-def **by** auto

lemma getActV0:getActV s₀ = getActV s₀' **unfolding** Van.getAct-def **by** auto

lemma getObsV0:getObsV s₀ ≠ getObsV s₀' **unfolding** Van.getObs-def **by** auto

lemma *getAct1:getActV s₁ ≠ getActV s₁' unfolding Van.getAct-def by auto*

lemma *validFromO:Opt.validFromS s₀ [s₀, s₁, s₂] unfolding Opt.validFromS-def
Opt.validS-def apply clarsimp
by (metis less-Suc0 not-less-less-Suc-eq nth-Cons-0
nth-Cons-Suc)*

lemma *validFromO':Opt.validFromS s₀' [s₀', s₁', s₂] unfolding Opt.validFromS-def
Opt.validS-def apply clarsimp
by (metis less-Suc0 not-less-less-Suc-eq nth-Cons-0
nth-Cons-Suc)*

lemma *tr1-shape-s0-aux:Van.validFromS s₀ tr1 ⇒ completedFromO s₀ tr1 ⇒
tr1 = [s₀, s₁, s₂]
unfolding completedFromO-def apply(erule disjE)
subgoal by(simp add: final-iff)
unfolding Van.validFromS-def Van.validS-def final-iff
apply(cases tr1, auto split: if-splits)
subgoal for tr1' apply(cases tr1', auto split: if-splits, force+)
subgoal premises p for a tr1''
using p apply-apply(erule allE[of - 0], auto)
using p apply-apply(erule allE[of - 1], auto)
using p apply-apply(erule allE[of - 2], auto)
by (metis Suc-lessI grOI length-0-conv length-Suc-conv nth-Cons-0) . .*

lemma *tr1-shape-s0:s1 = s₀ ⇒ Van.validFromS s1 tr1 ⇒ completedFromO s1
tr1 ⇒ tr1 = [s₀, s₁, s₂]
using tr1-shape-s0-aux by auto*

lemma *tr1-shape-s0'-aux:Van.validFromS s₀' tr1 ⇒ completedFromO s₀' tr1 ⇒
tr1 = [s₀', s₁', s₂]
unfolding completedFromO-def apply(erule disjE)
subgoal by(simp add: final-iff)
unfolding Van.validFromS-def Van.validS-def final-iff
apply(cases tr1, auto split: if-splits)
subgoal for tr1' apply(cases tr1', auto split: if-splits, force+)
subgoal premises p for a tr1''
using p apply-apply(erule allE[of - 0], auto)
using p apply-apply(erule allE[of - 1], auto)
using p apply-apply(erule allE[of - 2], auto)
by (metis Suc-lessI grOI length-0-conv length-Suc-conv nth-Cons-0) . .*

lemma *tr1-shape-s0':s1 = s₀' ⇒ Van.validFromS s1 tr1 ⇒ completedFromO s1
tr1 ⇒ tr1 = [s₀', s₁', s₂]
using tr1-shape-s0'-aux by auto*

proposition $\neg rsecure$

unfolding *rsecure-def2* **unfolding** *not-all not-imp*
apply(*rule exI*[*of* - s_0],*rule exI*[*of* - [s_0, s_1, s_2]])
apply(*rule exI*[*of* - s_0'],*rule exI*[*of* - [s_0', s_1', s_2]])
apply(*intro conjI*, *simp-all add: validFromO validFromO' getAct0 getObs0*)
apply(*intro allI*)
subgoal for *sv1* **apply**(*cases sv1*, *simp-all, intro allI impI*)

subgoal for *trv1 sv2* **apply**(*cases sv2*, *simp-all, intro allI impI*)

subgoal for *trv2*

apply(*frule tr1-shape-s0*[*of* - *trv1*], *simp-all*)

by(*frule tr1-shape-s0*[*of* - *trv2*], *simp-all*)

apply(*intro allI impI*)

subgoal for *trv2*

apply(*frule tr1-shape-s0*[*of* - *trv1*], *simp-all*)

by(*frule tr1-shape-s0'*[*of* - *trv2*], *simp-all add: getAct1*) .

apply(*intro allI impI*)

subgoal for *trv1 sv2* **apply**(*cases sv2*, *simp-all, intro allI impI*)

subgoal for *trv2*

apply(*frule tr1-shape-s0'*[*of* - *trv1*], *simp-all*)

apply(*frule tr1-shape-s0*[*of* - *trv2*], *simp-all*)

using *getAct1* **by** *auto*

apply(*intro allI impI*)

subgoal for *trv2*

apply(*frule tr1-shape-s0'*[*of* - *trv1*], *simp-all*)

by(*frule tr1-shape-s0'*[*of* - *trv2*], *simp-all add: getAct1*) . . .

thm *unwindSD-rsecure*

lemma *tr1-shape-s0-aux**Opt:Opt.validFromS* s_0 *tr1* \implies *completedFromO* s_0 *tr1*
 \implies *tr1* = [s_0, s_1, s_2]

unfolding *completedFromO-def* **apply**(*erule disjE*)

subgoal by(*simp add: final-iff*)

unfolding *Opt.validFromS-def Opt.validS-def final-iff*

apply(*cases tr1*, *auto split: if-splits*)

subgoal for *tr1'* **apply**(*cases tr1'*, *auto split: if-splits, force+*)

subgoal premises *p* **for** *a tr1''*

using *p* **apply-apply**(*erule allE*[*of* - *0*], *auto*)

using p **apply-apply**($erule\ allE[of\ -\ 1]$, $auto$)
using p **apply-apply**($erule\ allE[of\ -\ 2]$, $auto$)
by ($metis\ Suc-lessI\ gr0I\ length-0-conv\ length-Suc-conv\ nth-Cons-0$) . .

lemma $tr1\text{-shape-}s0\text{-}Opt:s1 = s_0 \implies Opt.validFromS\ s1\ tr1 \implies completedFromO\ s1\ tr1 \implies tr1 = [s_0, s_1, s_2]$
using $tr1\text{-shape-}s0\text{-}auxOpt$ **by** $auto$

lemma $tr1\text{-shape-}s0'\text{-}auxOpt:Opt.validFromS\ s_0'\ tr1 \implies completedFromO\ s_0'\ tr1 \implies tr1 = [s_0', s_1', s_2]$
unfolding $completedFromO\text{-}def$ **apply**($erule\ disjE$)
subgoal **by**($simp\ add:\ final\text{-}iff$)
unfolding $Opt.validFromS\text{-}def\ Opt.validS\text{-}def\ final\text{-}iff$
apply($cases\ tr1$, $auto\ split:\ if\text{-}splits$)
subgoal **for** $tr1'$ **apply**($cases\ tr1'$, $auto\ split:\ if\text{-}splits$, $force+$)
subgoal **premises** p **for** $a\ tr1''$
using p **apply-apply**($erule\ allE[of\ -\ 0]$, $auto$)
using p **apply-apply**($erule\ allE[of\ -\ 1]$, $auto$)
using p **apply-apply**($erule\ allE[of\ -\ 2]$, $auto$)
by ($metis\ Suc-lessI\ gr0I\ length-0-conv\ length-Suc-conv\ nth-Cons-0$) . .

lemma $tr1\text{-shape-}s0'\text{-}Opt:s1 = s_0' \implies Opt.validFromS\ s1\ tr1 \implies completedFromO\ s1\ tr1 \implies tr1 = [s_0', s_1', s_2]$
using $tr1\text{-shape-}s0'\text{-}auxOpt$ **by** $auto$

lemma $OptS[simp]:Opt.S\ [s_0, s_1, s_2] = [s_0]$ **unfolding** $Opt.S\text{-}def$ **by** $auto$
lemma $OptS'[simp]:Opt.S\ [s_0', s_1', s_2] = [s_0']$ **unfolding** $Opt.S\text{-}def$ **by** $auto$

lemma $reachO0:reachO\ s_0$ **using** $Opt.reach.Istate$ **by** $auto$
lemma $reachV0:reachV\ s_0$ **using** $Van.reach.Istate$ **by** $auto$
lemma $reachO0':reachO\ s_0'$ **using** $Opt.reach.Istate$ **by** $auto$
lemma $reachV0':reachV\ s_0'$ **using** $Van.reach.Istate$ **by** $auto$

lemma $SD\text{-}incomplete:$
assumes
 $s1 = s_0 \vee s1 = s_0'$
 $Opt.validFromS\ s1\ tr1$
 $completedFromO\ s1\ tr1$
 $s4 = s_0 \vee s4 = s_0'$
 $Opt.validFromS\ s4\ tr2$
 $completedFromO\ s4\ tr2$
 $Opt.A\ tr1 = Opt.A\ tr2$
 $Opt.O\ tr1 \neq Opt.O\ tr2$
 $\bigwedge sv1\ sv2.$
 $sv1 = s_0 \vee sv1 = s_0' \implies$

```

    corrState sv1 s1  $\implies$  sv2 = s0  $\vee$  sv2 = s0'  $\implies$  corrState sv2 s4  $\implies$   $\Gamma$  sv1
  (Opt.S tr1) sv2 (Opt.S tr2)
  unwindSDCond  $\Gamma$ 
  shows False
  using assms(9)[OF assms(1) - assms(4), simplified] assms(1,4)
  apply-apply(elim disjE)
  subgoal using tr1-shape-s0-Opt[OF - assms(2,3), simplified]
    tr1-shape-s0-Opt[OF - assms(5,6), simplified]
    assms(7,8) by simp

  subgoal using tr1-shape-s0'-Opt[OF - assms(2,3), simplified]
    tr1-shape-s0'-Opt[OF - assms(5,6), simplified]
    assms(7,8) apply simp
  using assms(10)[unfolded unwindSDCond-def]
  apply-apply(erule allE[of - s0], erule allE[of - [s0]])
  apply-apply(erule allE[of - s0 $\uparrow$ ], elim allE[of - [s0 $\uparrow$ ]] impE)
  subgoal using reachV0' reachV0 by auto
  by (auto simp: getActV0 getObsV0)

  subgoal using tr1-shape-s0'-Opt[OF - assms(2,3), simplified]
    tr1-shape-s0-Opt[OF - assms(5,6), simplified]
    assms(7,8) apply simp
  using assms(10)[unfolded unwindSDCond-def]
  apply-apply(erule allE[of - s0 $\uparrow$ ], erule allE[of - [s0 $\uparrow$ ]])
  apply-apply(erule allE[of - s0], elim allE[of - [s0]] impE)
  subgoal using reachV0' reachV0 by auto
  using getActV0 getObsV0 by auto
  subgoal using tr1-shape-s0'-Opt[OF - assms(2,3), simplified]
    tr1-shape-s0'-Opt[OF - assms(5,6), simplified]
    assms(7,8) by simp .

end

```

4 Secret Directed Unwinding Incompleteness example

Demonstrating a counterexample which is secure but fails in the infinitary unwinding

```

theory SD-Incomplete
  imports SD-Unwinding
begin

```

```

no-notation bot ( $\perp$ )

```

abbreviation *noninform* (\perp) **where** $\perp \equiv \text{undefined}$

datatype *State* = $s_0 \mid s_0' \mid s_1 \mid s_1' \mid s_2$
type-synonym *secret* = *State*

fun *transit*::*State* \Rightarrow *State* \Rightarrow *bool*(**infix** $\rightarrow I$ 55) **where**
transit $s\ s' =$

(*if* ($s = s_0 \wedge s' = s_1$) \vee
($s = s_1 \wedge s' = s_2$) \vee
($s = s_0' \wedge s' = s_1'$) \vee
($s = s_1' \wedge s' = s_2$) *then True*
else False)

lemma *transit-s0-s1*: $s_0 \rightarrow I\ s_1$ **by** *auto*

lemma *transit-s1-s2*: $s_1 \rightarrow I\ s_2$ **by** *auto*

lemma *transit-s0'-s1'*: $s_0' \rightarrow I\ s_1'$ **by** *auto*

lemma *transit-s1'-s2*: $s_1' \rightarrow I\ s_2$ **by** *auto*

lemma *transit-iff*: $s \rightarrow I\ s' \longleftrightarrow (s = s_0 \wedge s' = s_1) \vee$
($s = s_1 \wedge s' = s_2$) \vee
($s = s_0' \wedge s' = s_1'$) \vee
($s = s_1' \wedge s' = s_2$) **by** *auto*

definition *final* $x \equiv \forall y. \neg (\rightarrow I)\ x\ y$

lemma *final-s2*[*simp*]:*final* s_2 **unfolding** *final-def transit-iff* **by** *auto*

lemma *final-iff*: *final* $s \longleftrightarrow s = s_2$ **unfolding** *final-def transit-iff* **by** (*auto,metis State.exhaust*)

Vanilla-semantics system model

type-synonym *stateV* = *State*

fun *validTransV* **where** *validTransV* (s, s') = $s \rightarrow I\ s'$

Secrets at initial states, interaction at everywhere besides final (i.e. s_2)

fun *isIntV* :: *stateV* \Rightarrow *bool* **where** *isIntV* $s = (s \neq s_2)$

fun *getIntV* :: *stateV* \Rightarrow *nat* \times *nat* **where**

getIntV $s =$

(*case* s *of*
 $s_0 \Rightarrow (1, 0)$
 $| s_0' \Rightarrow (1, 1)$
 $| s_1 \Rightarrow (0, \perp)$)

```

    |s1' ⇒ (1, ⊥)
    |- ⇒ (⊥, ⊥)
  )

```

definition $isSecV :: stateV \Rightarrow bool$ **where** $isSecV\ s = (s \in \{s_0, s_0'\})$
fun $getSecV :: stateV \Rightarrow secret$ **where** $getSecV\ s = s$

lemma $getSecV\ neg$: $getSecV\ s_0 \neq getSecV\ s_0'$ **by** *auto*

The optimization-enhanced system model

type-synonym $stateO = State$
fun $validTransO$ **where** $validTransO\ (s, s') = s \rightarrow I\ s'$

Secrets and interaction at initial states

fun $isIntO :: stateO \Rightarrow bool$ **where** $isIntO\ s = (s \in \{s_0, s_0'\})$

fun $getIntO :: stateO \Rightarrow nat \times nat$ **where**
 $getIntO\ s =$
 (case s of
 $s_0 \Rightarrow (1, 0)$
 $|s_0' \Rightarrow (1, 1)$
 $|- \Rightarrow (\perp, \perp)$
)

definition $isSecO :: stateO \Rightarrow bool$ **where** $isSecO\ s = (s \in \{s_0, s_0'\})$

lemma $isSecO[simp]$: $isSecO\ s_0$ **unfolding** $isSecO\ def$ **by** *auto*

lemma $isSec1[simp]$: $\neg isSecO\ s_1$ **unfolding** $isSecO\ def$ **by** *auto*

lemma $isSec0'[simp]$: $isSecO\ s_0'$ **unfolding** $isSecO\ def$ **by** *auto*

lemma $isSec1'[simp]$: $\neg isSecO\ s_1'$ **unfolding** $isSecO\ def$ **by** *auto*

fun $getSecO :: stateO \Rightarrow secret$ **where** $getSecO\ s = s$

$corrState$

fun $corrState :: stateV \Rightarrow stateO \Rightarrow bool$ **where**
 $corrState\ cfgO\ cfgA = True$

interpretation $Rel\ Sec$

where $validTransV = validTransV$ **and** $istateV = \lambda s. s = s_0 \vee s = s_0'$

and $finalV = final$

and $isSecV = isSecV$ **and** $getSecV = getSecV$

and $isIntV = isIntV$ **and** $getIntV = getIntV$

and $validTransO = validTransO$ **and** $istateO = \lambda s. s = s_0 \vee s = s_0'$

and $finalO = final$

and $isSecO = isSecO$ **and** $getSecO = getSecO$

and $isIntO = isIntO$ **and** $getIntO = getIntO$

and $corrState = corrState$
apply(*unfold-locales*)
subgoal by (*auto simp: final-def*)
subgoal by (*auto simp: final-iff*)
subgoal by (*auto simp: final-iff isSecV-def*)
subgoal by (*auto simp: final-def*)
subgoal by (*auto simp: final-iff*)
subgoal by (*auto simp: final-iff isSecO-def*) .

lemma $getAct0:getActO\ s_0 = getActO\ s_0'$ **unfolding** *Opt.getAct-def* **by** *auto*
lemma $getObs0:getObsO\ s_0 \neq getObsO\ s_0'$ **unfolding** *Opt.getObs-def* **by** *auto*

lemma $getActV0:getActV\ s_0 = getActV\ s_0'$ **unfolding** *Van.getAct-def* **by** *auto*
lemma $getObsV0:getObsV\ s_0 \neq getObsV\ s_0'$ **unfolding** *Van.getObs-def* **by** *auto*

lemma $getAct1:getActV\ s_1 \neq getActV\ s_1'$ **unfolding** *Van.getAct-def* **by** *auto*

lemma $validFromO:Opt.lvalidFromS\ s_0\ [[s_0, s_1, s_2]]$ **unfolding** *Opt.lvalidFromS-def*
Opt.lvalidS-def **apply** *clarsimp*
subgoal for i **by**(*cases i, simp-all add: eSuc-def*) .

lemma $validFromO':Opt.lvalidFromS\ s_0'\ [[s_0', s_1', s_2]]$ **unfolding** *Opt.lvalidFromS-def*
Opt.lvalidS-def **apply** *clarsimp*
subgoal for i **by**(*cases i, simp-all add: eSuc-def*) .

lemma $validTrFinite:Van.lvalidFromS\ s_0\ tr1 \implies lfinite\ tr1$
unfolding *Van.lvalidFromS-def* *Van.lvalidS-def*
by (*auto,metis State.distinct enat-ord-code(4) idiff-infinity llength-eq-infty-conv-lfinite*
)

lemma $validTrFinite':Van.lvalidFromS\ s_0'\ tr1 \implies lfinite\ tr1$
unfolding *Van.lvalidFromS-def* *Van.lvalidS-def*
by (*auto,metis State.distinct enat-ord-code(4) idiff-infinity llength-eq-infty-conv-lfinite*
)

lemma $validOptTrFinite:Opt.lvalidFromS\ s_0\ tr1 \implies lfinite\ tr1$
unfolding *Opt.lvalidFromS-def* *Opt.lvalidS-def*
by (*auto,metis State.distinct enat-ord-code(4) idiff-infinity llength-eq-infty-conv-lfinite*
)

lemma *validOptTrFinite'*: *Opt.lvalidFromS* s_0' $tr1 \implies lfinite$ $tr1$
unfolding *Opt.lvalidFromS-def* *Opt.lvalidS-def*
by (*auto,metis* *State.distinct enat-ord-code(4) idiff-infinity llength-eq-infnty-conv-lfinite*
)

lemma *enat-reduce*: *enat* $i < enat$ $x \implies i < x$ **by** *simp*

find-theorems *lnth*

lemma *tr1-shape-s0-aux*: *Van.lvalidFromS* s_0 $tr1 \implies lcompletedFromO$ s_0 $tr1 \implies$
 $tr1 = [[s_0, s_1, s_2]]$
unfolding *Opt.lcompletedFrom-def* **apply**(*erule impE*)
subgoal by(*simp add: validTrFinite*)
apply(*frule validTrFinite*)
unfolding *Van.lvalidFromS-def* *Van.lvalidS-def* *final-iff*
apply(*cases tr1, auto split: if-splits*)
subgoal for $tr1'$
apply(*unfold nth-list-of[symmetric, of tr1']*)
apply(*unfold nth-list-of[symmetric, of (s0 \$ tr1')*, *unfolded lfinite-LCons*])
apply(*unfold length-list-of[symmetric]*)
apply(*cases tr1', simp*)
subgoal premises p
using p **apply**–**apply**(*erule allE[of - 0], simp split: if-splits*)
using p **apply**–**apply**(*erule allE[of - 1], simp split: if-splits*)
using p **apply**–**apply**(*erule allE[of - 2], simp split: if-splits*)
by (*metis* *Suc-length-conv length-0-conv less-Suc0 lfinite.simps linorder-neqE-nat*
list.distinct(1) list.inject list-of-LCons list-of-LNil
lnth-0) . .

lemma *tr1-shape-s0:s1 = s0* $\implies Van.lvalidFromS$ $s1$ $tr1 \implies lcompletedFromO$ $s1$
 $tr1 \implies tr1 = [[s_0, s_1, s_2]]$
using *tr1-shape-s0-aux* **by** *auto*

lemma *tr1-shape-s0'-aux*: *Van.lvalidFromS* s_0' $tr1 \implies lcompletedFromO$ s_0' $tr1$
 $\implies tr1 = [[s_0', s_1', s_2]]$
unfolding *Opt.lcompletedFrom-def* **apply**(*erule impE*)
subgoal by(*simp add: validTrFinite'*)
apply(*frule validTrFinite'*)
unfolding *Van.lvalidFromS-def* *Van.lvalidS-def* *final-iff*
apply(*cases tr1, auto split: if-splits*)
subgoal for $tr1'$
apply(*unfold nth-list-of[symmetric, of tr1']*)
apply(*unfold nth-list-of[symmetric, of (s0' \$ tr1')*, *unfolded lfinite-LCons*])
apply(*unfold length-list-of[symmetric]*)
apply(*cases tr1', simp*)
subgoal premises p
using p **apply**–**apply**(*erule allE[of - 0], simp split: if-splits*)

using p **apply**–**apply**(*erule allE*[of - 1], *simp split: if-splits*)
using p **apply**–**apply**(*erule allE*[of - 2], *simp split: if-splits*)
by (*metis Suc-length-conv length-0-conv less-Suc0 lfinite.simps linorder-neqE-nat*
list.distinct(1) list.inject list-of-LCons list-of-LNil
lnth-0) . .

lemma *tr1-shape-s0'*: $s1 = s0' \implies \text{Van.lvalidFromS } s1 \text{ } tr1 \implies \text{lcompletedFromO}$
 $s1 \text{ } tr1 \implies tr1 = [[s0', s1', s2]]$
using *tr1-shape-s0'-aux* **by** *auto*

proposition *¬lrsecure*

unfolding *lrsecure-def2* **unfolding** *not-all not-imp*
apply(*rule exI*[of - s_0],*rule exI*[of - $[[s_0, s_1, s_2]]$])
apply(*rule exI*[of - s_0],*rule exI*[of - $[[s_0', s_1', s_2]]$])
apply(*intro conjI*, *simp-all add: validFromO validFromO' getAct0 getObs0*)
apply(*intro allI*)
subgoal for $sv1$ **apply**(*cases sv1*, *simp-all,intro allI impI*)

subgoal for $trv1 \ sv2$ **apply**(*cases sv2*, *simp-all, intro allI impI*)

subgoal for $trv2$
apply(*frule tr1-shape-s0*[of - $trv1$], *simp-all*)
by(*frule tr1-shape-s0*[of - $trv2$], *simp-all*)
apply(*intro allI impI*)
subgoal for $trv2$
apply(*frule tr1-shape-s0*[of - $trv1$], *simp-all*)
by(*frule tr1-shape-s0*'[of - $trv2$], *simp-all add: getAct1*) .

apply(*intro allI impI*)
subgoal for $trv1 \ sv2$ **apply**(*cases sv2*, *simp-all, intro allI impI*)

subgoal for $trv2$
apply(*frule tr1-shape-s0*'[of - $trv1$], *simp-all*)
apply(*frule tr1-shape-s0*[of - $trv2$], *simp-all*)
using *getAct1* **by** *auto*
apply(*intro allI impI*)
subgoal for $trv2$
apply(*frule tr1-shape-s0*'[of - $trv1$], *simp-all*)
by(*frule tr1-shape-s0*'[of - $trv2$], *simp-all add: getAct1*) . . .

lemma *tr1-shape-s0-auxOpt*: $\text{Opt.lvalidFromS } s_0 \text{ } tr1 \implies \text{lcompletedFromO } s_0 \text{ } tr1$
 $\implies tr1 = [[s_0, s_1, s_2]]$
unfolding *Opt.lcompletedFrom-def* **apply**(*erule impE*)
subgoal by(*simp add: validOptTrFinite*)
apply(*frule validOptTrFinite*)

unfolding *Opt.lvalidFromS-def Opt.lvalidS-def final-iff*
apply(*cases tr1, auto split: if-splits*)
subgoal for *tr1'*
apply(*unfold nth-list-of[symmetric, of tr1']*)
apply(*unfold nth-list-of[symmetric, of (s0 \$ tr1'), unfolded lfinite-LCons]*)
apply(*unfold length-list-of[symmetric]*)
apply(*cases tr1', simp*)
subgoal premises *p*
using *p apply-apply(erule allE[of - 0], simp split: if-splits)*
using *p apply-apply(erule allE[of - 1], simp split: if-splits)*
using *p apply-apply(erule allE[of - 2], simp split: if-splits)*
by (*metis Suc-length-conv length-0-conv less-Suc0 lfinite.simps linorder-neqE-nat*
list.distinct(1) list.inject list-of-LCons list-of-LNil
lnth-0) . .

lemma *tr1-shape-s0-Opt:s1 = s0 \implies Opt.lvalidFromS s1 tr1 \implies lcompleted-*
FromO s1 tr1 \implies tr1 = [[s0, s1, s2]]
using *tr1-shape-s0-auxOpt by auto*

lemma *tr1-shape-s0'-auxOpt:Opt.lvalidFromS s0' tr1 \implies lcompletedFromO s0' tr1*
 \implies tr1 = [[s0', s1', s2]]
unfolding *Opt.lcompletedFrom-def apply(erule impE)*
subgoal by(*simp add: validOptTrFinite')*
apply(*frule validOptTrFinite')*
unfolding *Opt.lvalidFromS-def Opt.lvalidS-def final-iff*
apply(*cases tr1, auto split: if-splits*)
subgoal for *tr1'*
apply(*unfold nth-list-of[symmetric, of tr1']*)
apply(*unfold nth-list-of[symmetric, of (s0' \$ tr1'), unfolded lfinite-LCons]*)
apply(*unfold length-list-of[symmetric]*)
apply(*cases tr1', simp*)
subgoal premises *p*
using *p apply-apply(erule allE[of - 0], simp split: if-splits)*
using *p apply-apply(erule allE[of - 1], simp split: if-splits)*
using *p apply-apply(erule allE[of - 2], simp split: if-splits)*
by (*metis Suc-length-conv length-0-conv less-Suc0 lfinite.simps linorder-neqE-nat*
list.distinct(1) list.inject list-of-LCons list-of-LNil
lnth-0) . .

lemma *tr1-shape-s0'-Opt:s1 = s0' \implies Opt.lvalidFromS s1 tr1 \implies lcompleted-*
FromO s1 tr1 \implies tr1 = [[s0', s1', s2]]
using *tr1-shape-s0'-auxOpt by auto*

lemma *OptS[simp]:Opt.S [s0, s1, s2] = [s0] unfolding Opt.S-def by auto*
lemma *OptS'[simp]:Opt.S [s0', s1', s2] = [s0'] unfolding Opt.S-def by auto*

lemma *reachO0:reachO s0 using Opt.reach.Istate by auto*

lemma *reachV0:reachV* s_0 **using** *Van.reach.Istate* **by** *auto*
lemma *reachO0':reachO* s_0' **using** *Opt.reach.Istate* **by** *auto*
lemma *reachV0':reachV* s_0' **using** *Van.reach.Istate* **by** *auto*

lemma *SD-incomplete:*

assumes
 $s1 = s_0 \vee s1 = s_0'$
Opt.lvalidFromS $s1$ $tr1$
lcompletedFromO $s1$ $tr1$
 $s4 = s_0 \vee s4 = s_0'$
Opt.lvalidFromS $s4$ $tr2$
lcompletedFromO $s4$ $tr2$
Opt.lA $tr1 = \text{Opt.lA } tr2$
Opt.lO $tr1 \neq \text{Opt.lO } tr2$
 $\bigwedge sv1 sv2.$
 $sv1 = s_0 \vee sv1 = s_0' \implies$
 $corrState sv1 s1 \implies sv2 = s_0 \vee sv2 = s_0' \implies corrState sv2 s4 \implies \Gamma sv1$
(*Opt.lS* $tr1$) $sv2$ (*Opt.lS* $tr2$)
lunwindSDCond Γ
shows *False*
using *assms(9)[OF assms(1) - assms(4), simplified]* *assms(1,4)*
apply-apply (*elim disjE*)
subgoal using *tr1-shape-s0-Opt[OF - assms(2,3), simplified]*
tr1-shape-s0-Opt[OF - assms(5,6), simplified]
assms(7,8) **by** *simp*

subgoal using *tr1-shape-s0-Opt[OF - assms(2,3), simplified]*
tr1-shape-s0'-Opt[OF - assms(5,6), simplified]
assms(7,8) **apply** *simp*
using *assms(10)[unfolded lunwindSDCond-def]*
apply-apply (*erule allE[of - s0], erule allE[of - [[s0]]]*)
apply-apply (*erule allE[of - s0'], elim allE[of - [[s0']] impE*)
subgoal using *reachV0' reachV0* **by** *auto*
by (*auto simp: getActV0 getObsV0*)

subgoal using *tr1-shape-s0'-Opt[OF - assms(2,3), simplified]*
tr1-shape-s0-Opt[OF - assms(5,6), simplified]
assms(7,8) **apply** *simp*
using *assms(10)[unfolded lunwindSDCond-def]*
apply-apply (*erule allE[of - s0'], erule allE[of - [[s0']]*)
apply-apply (*erule allE[of - s0], elim allE[of - [[s0]]] impE*)
subgoal using *reachV0' reachV0* **by** *auto*
using *getActV0 getObsV0* **by** *auto*
subgoal using *tr1-shape-s0'-Opt[OF - assms(2,3), simplified]*
tr1-shape-s0'-Opt[OF - assms(5,6), simplified]
assms(7,8) **by** *simp* .

end

References

- [1] A. P. Brijesh Dongol, Matt Griffin and J. Wright. Relative security: Formally modeling and (dis)proving resilience against semantic optimization vulnerabilities. In *37th IEEE Computer Security Foundations Symposium, CSF 2024*. To appear.
- [2] A. Popescu, T. Bauereiss, and P. Lammich. Bounded-Deducibility security (invited paper). In L. Cohen and C. Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPICs*, pages 3:1–3:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.