Secret-Directed Unwinding

Brijesh Dongol Matt Griffin Andrei Popescu
Jamie Wright

June 5, 2024

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

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

theory *SD-Unwinding-fin*
imports *Relative-Security.Relative-Security*
begin

context *Rel-Sec*
begin

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

definition *move1* Γ *sv1 secl1 sv2 secl2* ≡
∀ *sv1' secl1'. validEtransO* (*sv1*,*secl1*) (*sv1'*,*secl1'*) ⟶ Γ *sv1' secl1' sv2 secl2*

definition *move2* Γ *sv1 secl1 sv2 secl2* ≡
∀ *sv2' secl2'. validEtransO* (*sv2*,*secl2*) (*sv2'*,*secl2'*) ⟶ Γ *sv1 secl1 sv2' secl2'*

definition *move12* Γ *sv1 secl1 sv2 secl2* ≡
∀ *sv1' secl1' sv2' secl2'*.
validEtransO (*sv1*,*secl1*) (*sv1'*,*secl1'*) ∧ *validEtransO* (*sv2*,*secl2*) (*sv2'*,*secl2'*)
⟶ Γ *sv1' secl1' sv2' secl2'*

definition *unwindSDCond* ::
(*'stateV* ⇒ *'secret list* ⇒ *'stateV* ⇒ *'secret list* ⇒ *bool*) ⇒ *bool*
where
unwindSDCond Γ ≡ ∀ *sv1 secl1 sv2 secl2*.
reachV sv1 ∧ *reachV sv2* ∧
Γ *sv1 secl1 sv2 secl2*
⟶
(*isIntV sv1* ⟷ *isIntV sv2*) ∧
(¬ *isIntV sv1* ⟶ *move1* Γ *sv1 secl1 sv2 secl2* ∧ *move2* Γ *sv1 secl1 sv2 secl2*) ∧
(*isIntV sv1* ⟶ *getActV sv1* = *getActV sv2* ⟶ *getObsV sv1* = *getObsV sv2* ∧
move12 Γ *sv1 secl1 sv2 secl2*)

proposition *unwindSDCond-aux*:
assumes *unw*: *unwindSDCond* Γ
and *1*: Γ *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 ¬ *finalV 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 ¬ *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'* ≡ *if isSecV sv1 then tl secl1 else secl1*
have *v1*: *validEtransO* (*sv1*,*secl1*) (*sv1'*,*secl1'*)
using *trv1* **unfolding** *validEtransO.simps secl1'-def*
using *1* ‹¬ *finalV sv1*›
by (*auto simp add*: *Van.S-def*)

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* ‹¬ *finalV sv2*› **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'* ∧ *reachV sv2'*
by (*metis 1.prems*(*2*) *1.prems*(*5*) *Van.reach.Step fst-conv snd-conv trv1*(*2*) *trv2*(*2*))

have *gasv12*: *getActV sv1* = *getActV sv2* **using** ‹*Van.A trv1* = *Van.A trv2*› *isv12* **unfolding** *trv1 trv2*
using *1.prems*(*4*) *1.prems*(*7*) ‹¬ *finalV sv2*› *trv1*(*1*) *trv2*(*1*) **by** *auto*
hence *osv12*: *getObsV sv1* = *getObsV sv2*
using ‹Γ *sv1 secl1 sv2 secl2*› *1.prems*(*2*,*5*) *isv12 unw* **unfolding** *unwindSD-Cond-def* **by** *auto*

have *gam*: Γ *sv1' secl1' sv2' secl2'*
using ‹Γ *sv1 secl1 sv2 secl2*› *v1 v2 unw r12' isv12 gasv12* **unfolding** *unwindS-DCond-def move12-def*
using *1*(*2*−) **by** *blast*

have *A12*: *Van.A trv1'* = *Van.A trv2'*
using *1.prems*(*10*) *1.prems*(*4*) *1.prems*(*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.prems*(*4*) *1.prems*(*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.prems(7) 1.prems(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.prems(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.prems(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.prems 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.prems*(*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.prems*(*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: *Van.S trv1* @ *ssecl1* = *secl1 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** ¬ *isIntV sv2* **using** *1*(*2*−) *unw* **unfolding** *unwindSD-Cond-def* **by** *auto*
note *isv12* = *False this*
have *trv1ne*: *trv1* ≠ [] **using** *isv12* *1.prems*(*3*) *11* **by** *force*
then obtain *sv1′ trv1′* **where** *trv1*: *trv1* = *sv1* # *trv1′* *validTransV* (*sv1*,*sv1′*)

Van.validFromS sv1′ trv1′ never isIntV trv1′
using ‹*Van.validFromS sv1* (*trv1* @ [*trn1*])› ‹*never isIntV trv1*›
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-simps(*1*) *neq-Nil-conv snoc-eq-iff-butlast*)
have *trv2ne*: *trv2* ≠ [] **using** *isv12* *1.prems*(*6*) *22* **by** *force*
then obtain *sv2′ trv2′* **where** *trv2*: *trv2* = *sv2* # *trv2′* *validTransV* (*sv2*,*sv2′*)

Van.validFromS sv2′ trv2′ never isIntV trv2′
using ‹*Van.validFromS sv2* (*trv2* @ [*trn2*])› ‹*never isIntV trv2*›
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-simps(*1*) *neq-Nil-conv snoc-eq-iff-butlast*)

show *?thesis*
proof(*cases length trv1* = *Suc 0* ∧ *length trv2* = *Suc 0*)
case *True*
hence *trv12*: *trv1* = [*sv1*] ∧ *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.prems*(*8*,*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.prems*(*9*) *trv2'ne* **by** *auto*
have *sl2'*: *secl2' = Van.S trv2' @ ssecl2*
using *1.prems*(*9*) *trv2*(*1*) *v2* **by** (*auto simp*: *trv2'ne*)
have *r2'*: *reachV sv2'*
by (*metis 1.prems*(*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 *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.prems*(*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' ≠ []* **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.prems*(*8*) *trv1'ne* **by** *auto*
have *sl1'*: *secl1' = Van.S trv1' @ ssecl1*
using *1.prems*(*8*) *trv1*(*1*) *v1* **by** (*auto simp*: *trv1'ne*)
have *r1'*: *reachV sv1'*
by (*metis 1.prems*(*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 *gam'*: Γ (*lastt sv1' trv1'*) *ssecl1* (*lastt sv2 trv2*) *ssecl2*
apply(*rule 1*(*1*)[*rule-format*, *OF - - gam*])

using *trv1 gam sl1′* ‹*reachV sv2*› ‹*reachV sv1*› *r1′*
using *1.prems*
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 ‹ $Van.S\ (x\ \#\ xs) = Opt.S\ (y\ \#\ ys)$› **and** ‹$xs \neq []$› **and** ‹$ys \neq []$›
shows ‹$(isSecO\ y \wedge isSecV\ x \longrightarrow getSecV\ x = getSecO\ y \wedge Van.S\ xs = Opt.S\ ys)$
$\wedge (isSecO\ y \wedge \neg isSecV\ x \longrightarrow Van.S\ xs = (getSecO\ y\ \#\ Opt.S\ ys))$
$\wedge (\neg isSecO\ y \wedge isSecV\ x \longrightarrow (getSecV\ x\ \#\ Van.S\ xs) = Opt.S\ ys)$
$\wedge (\neg isSecO\ y \wedge \neg isSecV\ x \longrightarrow Van.S\ xs = Opt.S\ ys)$›
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*: $\bigwedge sv1\ sv2.\ [\![istateV\ sv1;\ corrState\ sv1\ s1;\ istateV\ sv2;\ corrState\ sv2\ s4]\!] \Longrightarrow$
$\Gamma\ sv1\ (Opt.S\ tr1)\ sv2\ (Opt.S\ tr2)$
and *unw*: *unwindSDCond* Γ
shows ¬ *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') $\wedge$
$(\neg$ *isSecV s* $\wedge$ *secl* $=$ *secl'* $\vee$
isSecV s $\wedge$ *secl* $=$ *LCons* (*getSecV s*) *secl'*)

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

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

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

definition *lunwindSDCond* ::
(*'stateV* $\Rightarrow$ *'secret llist* $\Rightarrow$ *'stateV* $\Rightarrow$ *'secret llist* $\Rightarrow$ *bool*) $\Rightarrow$ *bool*
where
lunwindSDCond Γ $\equiv$ $\forall$ *sv1 secl1 sv2 secl2*.
reachV sv1 $\wedge$ *reachV sv2* $\wedge$
Γ *sv1 secl1 sv2 secl2*
$\longrightarrow$
(*isIntV sv1* $\longleftrightarrow$ *isIntV sv2*) $\wedge$
$(\neg$ *isIntV sv1* $\longrightarrow$ *lmove1* Γ *sv1 secl1 sv2 secl2* $\wedge$ *lmove2* Γ *sv1 secl1 sv2 secl2*)
$\wedge$
(*isIntV sv1* $\wedge$ *getActV sv1* $=$ *getActV sv2* $\longrightarrow$ *getObsV sv1* $=$ *getObsV sv2* $\wedge$
lmove12 Γ *sv1 secl1 sv2 secl2*)

lemma *lunwindSDCond-imp*:
assumes *lunwindSDCond* Γ *reachV sv1 reachV sv2* Γ *sv1 secl1 sv2 secl2*
shows
(*isIntV sv1* $\longleftrightarrow$ *isIntV sv2*) $\wedge$
$(\neg$ *isIntV sv1* $\longrightarrow$ *lmove1* Γ *sv1 secl1 sv2 secl2* $\wedge$ *lmove2* Γ *sv1 secl1 sv2 secl2*)
$\wedge$
(*isIntV sv1* $\wedge$ *getActV sv1* $=$ *getActV sv2* $\longrightarrow$ *getObsV sv1* $=$ *getObsV sv2* $\wedge$
lmove12 Γ *sv1 secl1 sv2 secl2*)
using *assms* **unfolding** *lunwindSDCond-def* **by** *auto*

lemma *lunwindSDCond-lmove12*:

assumes *unw*: *lunwindSDCond* Γ **and** *gam*: *reachV sv1 reachV sv2* Γ *sv1 secl1 sv2 secl2*
and *i*: *isIntV sv1* ⟶ *getActV sv1* = *getActV sv2*
shows *lmove12* Γ *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*: Γ *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 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** *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** ¬ *isIntV sv2* **using** *1*(*2*−) *unw* **unfolding** *lunwindSDCond-def* **by** *auto*
    note *isv12* = *False this*
    have *trv1ne*: *trv1* ≠ [] **using** *isv12 1.prems*(*3*) *11* **by** *force*
    then obtain *sv1′ trv1′* **where** *trv1*: *trv1* = *sv1* # *trv1′ validTransV* (*sv1*,*sv1′*)

Van.validFromS sv1' trv1' never isIntV trv1'
using ‹*Van.validFromS sv1* (*trv1* @ [*ssv1*])› ‹*never isIntV trv1*›
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-simps(*1*) *neq-Nil-conv*
snoc-eq-iff-butlast)
have *trv2ne*: *trv2* ≠ [] **using** *isv12 1.prems*(*6*) *22* **by** *force*
then obtain *sv2' trv2'* **where** *trv2*: *trv2* = *sv2* # *trv2' validTransV* (*sv2*,*sv2'*)

Van.validFromS sv2' trv2' never isIntV trv2'
using ‹*Van.validFromS sv2* (*trv2* @ [*ssv2*])› ‹*never isIntV trv2*›
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-simps(*1*) *neq-Nil-conv*
snoc-eq-iff-butlast)

show *?thesis*
proof(*cases length trv1* = *Suc 0* ∧ *length trv2* = *Suc 0*)
case *True*
hence *trv12*: *trv1* = [*sv1*] ∧ *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.prems*(*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'* ≠ [] **by** (*simp add*: *trv2*(*1*))

define *secl2'* **where** *secl2'* ≡ *if isSecV sv2 then ltl secl2 else secl2*
have *v2*: *lvalidEtransO* (*sv2*,*secl2*) (*sv2'*,*secl2'*)
using *trv2* **unfolding** *lvalidEtransO.simps secl2'-def*
using *1.prems*(*9*) *trv2'ne Van.S.FiltermapBL FiltermapBL.simps*(*3*) **by**
fastforce
have *sl2'*: *secl2'* = *lappend* (*llist-of* (*Van.S trv2'*)) *ssecl2*
using *1.prems*(*9*) *trv2*(*1*) *v2* **by** (*auto simp*: *trv2'ne*)
have *r2'*: *reachV sv2'*
by (*metis 1.prems*(*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** *lunwindSDCond-def*
lmove2-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.prems*(*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'* ≠ [] **by** (*simp add*: *trv1*(*1*))

define *secl1'* **where** *secl1'* ≡ *if isSecV sv1 then ltl secl1 else secl1*
have *v1*: *lvalidEtransO* (*sv1*,*secl1*) (*sv1'*,*secl1'*)
using *trv1* **unfolding** *lvalidEtransO.simps secl1'-def*
using *1.prems*(*8*) *trv1'ne Van.S.FiltermapBL FiltermapBL.Cons-unfold* **by** *fastforce*
have *sl1'*: *secl1'* = *lappend* (*llist-of* (*Van.S trv1'*)) *ssecl1*
using *1.prems*(*8*) *trv1*(*1*) *v1* **by** (*auto simp*: *trv1'ne*)
have *r1'*: *reachV sv1'*
by (*metis 1.prems*(*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** *lunwindSDCond-def lmove1-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 *trv1 gam sl1'* ‹*reachV sv2*› ‹*reachV sv1*› *r1'*
using *1.prems*
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* (*llist-of* (*map getSecV* (*filter isSecV trv1*))) *ssecl1* = *secl1*
lappend (*llist-of* (*map getSecV* (*filter isSecV trv2*))) *ssecl2* = *secl2*
shows (*getActV ssv1* = *getActV ssv2* ⟶ Γ *ssv1 ssecl1 ssv2 ssecl2*) ∧
(*getActV ssv1* = *getActV ssv2* ⟶ *getObsV ssv1* = *getObsV ssv2*)
proof−
define *ssecl1′* **where** *ssecl1′* = (*if trv1* = [] ∨ (*trv1* ≠ [] ∧ ¬ *isSecV* (*last trv1*)) *then ssecl1 else* (*getSecV* (*last trv1*)) $ *ssecl1*)
define *ssecl2′* **where** *ssecl2′* = (*if trv2* = [] ∨ (*trv2* ≠ [] ∧ ¬ *isSecV* (*last trv2*)) *then ssecl2 else* (*getSecV* (*last trv2*)) $ *ssecl2*)
have *s′*: *lappend* (*llist-of* (*Van.S trv1*)) *ssecl1′* = *secl1*
lappend (*llist-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-llist-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-llist-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-llist-of-validFromS Van.reach-validFromS-reach assms*(*7*) *assms*(*8*) *lappend-llist-of-llist-of lastt-def*)

have *trv1* = [] ⟷ *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* = [] ∧ *trv2* = []) ∨ (*trv1* ≠ [] ∧ *trv2* ≠ []) **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*: Γ *ssv1 ssecl1 ssv2 ssecl2* **unfolding** *sv12 secl* **using** *gg* **.**
assume *getActV ssv1* = *getActV ssv2*
thus *getObsV ssv1* = *getObsV ssv2*
using *lunwindSDCond-imp*[*OF unw rsv1 rsv2*, *unfolded sv12*[*symmetric*], *OF g*] *trv1*(*4*) **by** *auto*
qed
next
assume *trv12*: *trv1* ≠ [] *trv2* ≠ []
have *n*: ¬ *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*: Γ *ssv1 ssecl1 ssv2 ssecl2*
apply(*rule lunwindSDCond-lmove12*[*OF unw 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 unw - - g*]
by (*metis Van.reach.Step fst-conv rsv1 rsv2 snd-conv trv1*(*4*) *v1 v2*)
qed
qed
qed

proposition *lunwindSDCond-aux*:
assumes *unw*: *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 ∃ *sv1 trv1 secl1 sv2 trv2 secl2*. *obl1* = *Van.lO trv1* ∧ *obl2* = *Van.lO trv2* ∧
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*
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*: Γ *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*: ¬ *lnull obl1* ¬ *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* = [] ∧ *sv1* = *sv1′* ∧ *sv11* = *sv1*) ∨ (*ftrv1* ≠ [] ∧ *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′* ≡ *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* = [] ∧ *sv2* = *sv2'* ∧ *sv22* = *sv2*) ∨ (*ftrv2* ≠ [] ∧ *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'* ≡ *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'* ≡ *Van.lS trv1'* **and** *secl2'*: *secl2'* ≡ *Van.lS trv2'*

define *ssecl1'* **where** *ssecl1'* ≡ *Van.lS* (*sv1'* $ *trv1'*)
define *ssecl2'* **where** *ssecl2'* ≡ *Van.lS* (*sv2'* $ *trv2'*)

have *trv12'ne*: *trv1'* ≠ [[]] ∧ *trv2'* ≠ [[]]
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'* ⟶ Γ *sv1' ssecl1' sv2' ssecl2'*)
∧ (*getActV sv1' = getActV sv2'* ⟶ *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 ∃ *sv1 trv1 secl1 sv2 trv2 secl2*.
ltl obl1 = Van.lO trv1 ∧ *ltl obl2 = Van.lO trv2* ∧
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*
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*: *fftrv1 fftrv2 lbutlast-lappend trv12′ne*)

show *Van.lS trv1′* = *secl1′* **unfolding** *secl1′* **..**
show *Van.lS trv2′* = *secl2′* **unfolding** *secl2′* **..**

show Γ *ssv1′ secl1′ ssv2′ secl2′*
apply(*rule lunwindSDCond-lmove12*[*OF unw rsv1′ rsv2′ ggam*, *unfolded lmove12-def*, *rule-format*, *OF gasv12′*])
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* ≠ *Opt.lO tr2*
and *init*: ⋀*sv1 sv2*. *istateV sv1* ⟹ *corrState sv1 s1* ⟹ *istateV sv2* ⟹ *corrState sv2 s2* ⟹
Γ *sv1* (*Opt.lS tr1*) *sv2* (*Opt.lS tr2*)
and *unw*: *lunwindSDCond* Γ
shows ¬ *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 *tr14* **by** *auto*
subgoal unfolding *not-ex* **apply** *safe*

subgoal for *sv1 trv1 sv2 trv2* **apply**(*erule cnf.clause2raw-notE*[*of Van.lO trv1* $\neq$ *Van.lO trv2*], *simp*)
apply(*rule lunwindSDCond-aux*[*OF unw*, *OF init*])
using *Van.Istate* **by** *auto* **. .**

end

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.