

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	3
3	Secret Directed (Finitary) Unwinding Incompleteness example	6
4	Secret Directed Unwinding Incompleteness example	10

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*

<proof>

proposition *unwindSDCond-aux-strong*:

assumes *unw*: *unwindSDCond* Γ

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

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*
 ⟨*proof*⟩

lemma *S-eq-empty-ConsE*:
assumes $\langle \text{Van.S } (x \# xs) = \text{Opt.S } (y \# ys) \rangle$ **and** $\langle xs \neq [] \rangle$ **and** $\langle ys \neq [] \rangle$
shows $\langle (isSecO\ y \wedge isSecV\ x \longrightarrow getSecV\ x = getSecO\ y \wedge \text{Van.S } xs = \text{Opt.S } ys) \wedge (isSecO\ y \wedge \neg isSecV\ x \longrightarrow \text{Van.S } xs = (getSecO\ y \# \text{Opt.S } ys)) \wedge (\neg isSecO\ y \wedge isSecV\ x \longrightarrow (getSecV\ x \# \text{Van.S } xs) = \text{Opt.S } ys) \wedge (\neg isSecO\ y \wedge \neg isSecV\ x \longrightarrow \text{Van.S } xs = \text{Opt.S } ys) \rangle$
 ⟨*proof*⟩

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: $\bigwedge sv1\ sv2. \llbracket istateV\ sv1; corrState\ sv1\ s1; istateV\ sv2; corrState\ sv2\ s4 \rrbracket \implies$*
 $\Gamma\ sv1\ (\text{Opt.S } tr1)\ sv2\ (\text{Opt.S } tr2)$
and *unw: unwindSDCond Γ*
shows $\neg rsecure$
 ⟨*proof*⟩

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

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

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

definition $lunwindSDCond ::$
 $('stateV \Rightarrow 'secret llist \Rightarrow 'stateV \Rightarrow 'secret llist \Rightarrow bool) \Rightarrow bool$

where

$lunwindSDCond \Gamma \equiv \forall sv1 secl1 sv2 secl2.$

$reachV sv1 \wedge reachV sv2 \wedge$

$\Gamma sv1 secl1 sv2 secl2$

\longrightarrow

$(isIntV sv1 \longleftrightarrow isIntV sv2) \wedge$

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

\wedge

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

lemma $lunwindSDCond-imp:$

assumes $lunwindSDCond \Gamma reachV sv1 reachV sv2 \Gamma sv1 secl1 sv2 secl2$

shows

$(isIntV sv1 \longleftrightarrow isIntV sv2) \wedge$

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

\wedge

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

$\langle proof \rangle$

lemma $lunwindSDCond-lmove12:$

assumes $unw: lunwindSDCond \Gamma$ **and** $gam: reachV sv1 reachV sv2 \Gamma sv1 secl1 sv2 secl2$

and $i: isIntV sv1 \longrightarrow getActV sv1 = getActV sv2$

shows $lmove12 \Gamma sv1 secl1 sv2 secl2$

$\langle proof \rangle$

proposition $unwindSDCond-aux-inductive:$

assumes $unw: lunwindSDCond \Gamma$

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**
 $3: lappend\ (l\ list-of\ (Van.S\ trv1))\ ssecl1 = secl1\ lappend\ (l\ list-of\ (Van.S\ trv2))\ ssecl2 = secl2$
shows $\Gamma\ (lastt\ sv1\ trv1)\ ssecl1\ (lastt\ sv2\ trv2)\ ssecl2$
 $\langle proof \rangle$

proposition *unwindSDCond-inductive:*

assumes $unw: lunwindSDCond\ \Gamma$

and $gam: \Gamma\ sv1\ secl1\ sv2\ secl2$ **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 $(getActV\ ssv1 = getActV\ ssv2 \longrightarrow \Gamma\ ssv1\ ssecl1\ ssv2\ ssecl2) \wedge$

$(getActV\ ssv1 = getActV\ ssv2 \longrightarrow getObsV\ ssv1 = getObsV\ ssv2)$

$\langle proof \rangle$

proposition *lunwindSDCond-aux:*

assumes $unw: lunwindSDCond\ \Gamma$

and $1: \Gamma\ sv1\ secl1\ sv2\ secl2$

$reachV\ sv1\ Van.lvalidFromS\ sv1\ trv1\ lcompletedFromV\ sv1\ trv1$

$reachV\ sv2\ Van.lvalidFromS\ sv2\ trv2\ lcompletedFromV\ sv2\ trv2$

$Van.lS\ trv1 = secl1\ Van.lS\ trv2 = secl2$

$Van.lA\ trv1 = Van.lA\ trv2$

shows $Van.lO\ trv1 = Van.lO\ trv2$

$\langle proof \rangle$

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 $\neg lrsecure$

$\langle proof \rangle$

end

end

3 Secret Directed (Finitary) Unwinding Incompleteness example

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

```
theory SD-Incomplete-fin
  imports SD-Unwinding-fin
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: $s_0 \rightarrow I s_1$   $\langle$ proof $\rangle$ 
lemma transit-s1-s2: $s_1 \rightarrow I s_2$   $\langle$ proof $\rangle$ 
```

```
lemma transit-s0'-s1': $s_0' \rightarrow I s_1'$   $\langle$ proof $\rangle$ 
lemma transit-s1'-s2: $s_1' \rightarrow I s_2$   $\langle$ proof $\rangle$ 
```

```
lemma transit-iff: $s \rightarrow I s' \iff (s = s_0 \wedge s' = s_1) \vee$ 
  ( $s = s_1 \wedge s' = s_2$ )  $\vee$ 
  ( $s = s_0' \wedge s' = s_1'$ )  $\vee$ 
  ( $s = s_1' \wedge s' = s_2$ )  $\langle$ proof $\rangle$ 
```

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

```
lemma final-s2[simp]:final  $s_2$   $\langle$ proof $\rangle$ 
```

```
lemma final-iff: final  $s \iff s = s_2$   $\langle$ proof $\rangle$ 
```

Vanilla-semantic 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*

$s_0 \Rightarrow (1, 0)$

$|s_0' \Rightarrow (1, 1)$

$|s_1 \Rightarrow (0, \perp)$

$|s_1' \Rightarrow (1, \perp)$

$|_ \Rightarrow (\perp, \perp)$

)

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

fun $getSecV :: stateV \Rightarrow secret$ **where** $getSecV s = s$

lemma $getSecV\text{-neq}$: $getSecV s_0 \neq getSecV s_0'$ *<proof>*

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 $isSecO[simp]$: $isSecO s_0$ *<proof>*

lemma $isSecI[simp]$: $\neg isSecO s_1$ *<proof>*

lemma $isSecO'[simp]$: $isSecO s_0'$ *<proof>*

lemma $isSecI'[simp]$: $\neg isSecO s_1'$ *<proof>*

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

<proof>

lemma getAct0:getActO s₀ = getActO s₀' *<proof>*

lemma getObs0:getObsO s₀ \neq getObsO s₀' *<proof>*

lemma getActV0:getActV s₀ = getActV s₀' *<proof>*

lemma getObsV0:getObsV s₀ \neq getObsV s₀' *<proof>*

lemma getAct1:getActV s₁ \neq getActV s₁' *<proof>*

lemma validFromO:Opt.validFromS s₀ [s₀, s₁, s₂] *<proof>*

lemma validFromO':Opt.validFromS s₀' [s₀', s₁', s₂] *<proof>*

lemma tr1-shape-s0-aux:Van.validFromS s₀ tr1 \implies completedFromO s₀ tr1 \implies
tr1 = [s₀, s₁, s₂]

<proof>

lemma tr1-shape-s0:s1 = s₀ \implies Van.validFromS s1 tr1 \implies completedFromO s1
tr1 \implies tr1 = [s₀, s₁, s₂]

<proof>

lemma tr1-shape-s0'-aux:Van.validFromS s₀' tr1 \implies completedFromO s₀' tr1 \implies
tr1 = [s₀', s₁', s₂]

<proof>

lemma tr1-shape-s0':s1 = s₀' \implies Van.validFromS s1 tr1 \implies completedFromO s1

$tr1 \implies tr1 = [s_0', s_1', s_2]$
 $\langle proof \rangle$

proposition $\neg rsecure$
 $\langle proof \rangle$

thm $unwindSD-rsecure$

lemma $tr1\text{-shape-}s0\text{-aux}Opt:Opt.validFromS\ s_0\ tr1 \implies completedFromO\ s_0\ tr1$
 $\implies tr1 = [s_0, s_1, s_2]$
 $\langle proof \rangle$

lemma $tr1\text{-shape-}s0\text{-}Opt:s1 = s_0 \implies Opt.validFromS\ s1\ tr1 \implies completedFromO$
 $s1\ tr1 \implies tr1 = [s_0, s_1, s_2]$
 $\langle proof \rangle$

lemma $tr1\text{-shape-}s0'\text{-aux}Opt:Opt.validFromS\ s_0'\ tr1 \implies completedFromO\ s_0'\ tr1$
 $\implies tr1 = [s_0', s_1', s_2]$
 $\langle proof \rangle$

lemma $tr1\text{-shape-}s0'\text{-}Opt:s1 = s_0' \implies Opt.validFromS\ s1\ tr1 \implies completed-$
 $FromO\ s1\ tr1 \implies tr1 = [s_0', s_1', s_2]$
 $\langle proof \rangle$

lemma $OptS[simp]:Opt.S\ [s_0, s_1, s_2] = [s_0]$ $\langle proof \rangle$
lemma $OptS'[simp]:Opt.S\ [s_0', s_1', s_2] = [s_0']$ $\langle proof \rangle$

lemma $reachO0:reachO\ s_0$ $\langle proof \rangle$
lemma $reachV0:reachV\ s_0$ $\langle proof \rangle$
lemma $reachO0':reachO\ s_0'$ $\langle proof \rangle$
lemma $reachV0':reachV\ s_0'$ $\langle proof \rangle$

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 ≠ Opt.O tr2
∧ sv1 sv2.
  sv1 = s0 ∨ sv1 = s0' ⇒
  corrState sv1 s1 ⇒ sv2 = s0 ∨ sv2 = s0' ⇒ corrState sv2 s4 ⇒ Γ sv1
(Opt.S tr1) sv2 (Opt.S tr2)
unwindSDCond Γ
shows False
  ⟨proof⟩

```

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 (⊥)

```

```

abbreviation noninform (⊥) where ⊥ ≡ undefined

```

```

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

```

```

fun transit::State ⇒ State ⇒ bool(infix →I 55) where
  transit s s' =

```

```

  (if (s = s0 ∧ s' = s1) ∨
      (s = s1 ∧ s' = s2) ∨
      (s = s0' ∧ s' = s1') ∨
      (s = s1' ∧ s' = s2) then True
      else False)

```

```

lemma transit-s0-s1:s0 →I s1 ⟨proof⟩

```

```

lemma transit-s1-s2:s1 →I s2 ⟨proof⟩

```

lemma *transit-s0'-s1':s0' →I s1' <proof>*

lemma *transit-s1'-s2:s1' →I s2 <proof>*

lemma *transit-iff:s →I s' ↔ (s = s0 ∧ s' = s1) ∨
(s = s1 ∧ s' = s2) ∨
(s = s0' ∧ s' = s1') ∨
(s = s1' ∧ s' = s2) <proof>*

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

lemma *final-s2[simp]:final s2 <proof>*

lemma *final-iff: final s ↔ s = s2 <proof>*

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' <proof>*

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 ⟨*proof*⟩

lemma *isSec1*[*simp*]:¬*isSecO* s₁ ⟨*proof*⟩

lemma *isSecO'*[*simp*]:*isSecO* s₀' ⟨*proof*⟩

lemma *isSec1'*[*simp*]:¬*isSecO* s₁' ⟨*proof*⟩

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*

⟨*proof*⟩

lemma *getAct0*:*getActO* s₀ = *getActO* s₀' ⟨*proof*⟩

lemma *getObs0*:*getObsO* s₀ ≠ *getObsO* s₀' ⟨*proof*⟩

lemma *getActV0*:*getActV* s₀ = *getActV* s₀' ⟨*proof*⟩

lemma *getObsV0*:*getObsV* s₀ ≠ *getObsV* s₀' ⟨*proof*⟩

lemma *getAct1*:*getActV* s₁ ≠ *getActV* s₁' ⟨*proof*⟩

lemma *validFromO*:*Opt.lvalidFromS* s₀ [[s₀, s₁, s₂]] ⟨*proof*⟩

lemma *validFromO'*:*Opt.lvalidFromS* s₀' [[s₀', s₁', s₂]] ⟨*proof*⟩

lemma *validTrFinite*: *Van.lvalidFromS* s_0 $tr1 \implies lfinite$ $tr1$
<proof>

lemma *validTrFinite'*: *Van.lvalidFromS* s_0' $tr1 \implies lfinite$ $tr1$
<proof>

lemma *validOptTrFinite*: *Opt.lvalidFromS* s_0 $tr1 \implies lfinite$ $tr1$
<proof>

lemma *validOptTrFinite'*: *Opt.lvalidFromS* s_0' $tr1 \implies lfinite$ $tr1$
<proof>

lemma *enat-reduce*: *enat* $i < enat$ $x \implies i < x$ *<proof>*

find-theorems *lnth*

lemma *tr1-shape-s0-aux*: *Van.lvalidFromS* s_0 $tr1 \implies lcompletedFromO$ s_0 $tr1 \implies tr1 = [[s_0, s_1, s_2]]$
<proof>

lemma *tr1-shape-s0:s1 = s0*: $s1 = s_0 \implies Van.lvalidFromS$ $s1$ $tr1 \implies lcompletedFromO$ $s1$ $tr1 \implies tr1 = [[s_0, s_1, s_2]]$
<proof>

lemma *tr1-shape-s0'-aux*: *Van.lvalidFromS* s_0' $tr1 \implies lcompletedFromO$ s_0' $tr1 \implies tr1 = [[s_0', s_1', s_2]]$
<proof>

lemma *tr1-shape-s0':s1 = s0'*: $s1 = s_0' \implies Van.lvalidFromS$ $s1$ $tr1 \implies lcompletedFromO$ $s1$ $tr1 \implies tr1 = [[s_0', s_1', s_2]]$
<proof>

proposition $\neg lrsecure$
<proof>

lemma *tr1-shape-s0-auxOpt*: *Opt.lvalidFromS* s_0 $tr1 \implies lcompletedFromO$ s_0 $tr1 \implies tr1 = [[s_0, s_1, s_2]]$
<proof>

lemma *tr1-shape-s0-Opt*: $s1 = s_0 \implies \text{Opt.lvalidFromS } s1 \ tr1 \implies \text{lcompletedFromO } s1 \ tr1 \implies \text{tr1} = [[s_0, s_1, s_2]]$
 ⟨proof⟩

lemma *tr1-shape-s0'-auxOpt*: $\text{Opt.lvalidFromS } s_0' \ tr1 \implies \text{lcompletedFromO } s_0' \ tr1 \implies \text{tr1} = [[s_0', s_1', s_2]]$
 ⟨proof⟩

lemma *tr1-shape-s0'-Opt*: $s1 = s_0' \implies \text{Opt.lvalidFromS } s1 \ tr1 \implies \text{lcompletedFromO } s1 \ tr1 \implies \text{tr1} = [[s_0', s_1', s_2]]$
 ⟨proof⟩

lemma *OptS[simp]*: $\text{Opt.S } [s_0, s_1, s_2] = [s_0]$ ⟨proof⟩
lemma *OptS'[simp]*: $\text{Opt.S } [s_0', s_1', s_2] = [s_0']$ ⟨proof⟩

lemma *reachO0*: $\text{reachO } s_0$ ⟨proof⟩
lemma *reachV0*: $\text{reachV } s_0$ ⟨proof⟩
lemma *reachO0'*: $\text{reachO } s_0'$ ⟨proof⟩
lemma *reachV0'*: $\text{reachV } s_0'$ ⟨proof⟩

lemma *SD-incomplete*:

assumes

$s1 = s_0 \vee s1 = s_0'$

$\text{Opt.lvalidFromS } s1 \ tr1$

$\text{lcompletedFromO } s1 \ tr1$

$s4 = s_0 \vee s4 = s_0'$

$\text{Opt.lvalidFromS } s4 \ tr2$

$\text{lcompletedFromO } s4 \ tr2$

$\text{Opt.lA } tr1 = \text{Opt.lA } tr2$

$\text{Opt.lO } tr1 \neq \text{Opt.lO } tr2$

$\bigwedge sv1 \ sv2.$

$sv1 = s_0 \vee sv1 = s_0' \implies$

$\text{corrState } sv1 \ s1 \implies sv2 = s_0 \vee sv2 = s_0' \implies \text{corrState } sv2 \ s4 \implies \Gamma \ sv1$

$(\text{Opt.lS } tr1) \ sv2 \ (\text{Opt.lS } tr2)$

$\text{lunwindSDCond } \Gamma$

shows *False*

⟨proof⟩

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.