

Compositional BD Security

Thomas Bauereiss Andrei Popescu

February 6, 2026

Abstract

Building on a previous AFP entry [8] that formalizes the Bounded-Deducibility Security (BD Security) framework [7], we formalize compositionality and transport theorems for information flow security. These results allow lifting BD Security properties from individual components specified as transition systems, to a composition of systems specified as communicating products of transition systems. The underlying ideas of these results are presented in the papers [7] and [2]. The latter paper also describes a major case study where these results have been used: on verifying the CoSMedis distributed social media platform (itself formalized as an AFP entry [5] that builds on this entry).

Contents

1	Introduction	1
2	Binary compositionality theorem	3
3	Trivial security properties	24
4	Transporting BD Security	25
5	N-ary compositionality theorem	31
6	Combining independent secret sources	72

1 Introduction

Bounded-Deducibility Security (BD Security) [7] is a general framework for stating and proving information flow security, in particular, confidentiality properties. The framework works for any transition system and allows the specification of flexible policies for information flow security by describing the observations, the secrets, a bound on information release (also known as “declassification bound”) and a trigger for information release (also known as “declassification trigger”). The framework been deployed to verify the

confidentiality of (the functional kernels of) several web-based multi-user systems:

- the CoCon conference management system [6, 10] (also in the AFP [9])
- the CoSMed prototype social media platform [1, 3] (also in the AFP [4])
- the CoSMedis distributed extension of CoSMed [2] (also in the AFP [5])

This document presents some results that can help with the BD Security verification of large systems. They have been inspired by the challenges we faced when extending to CoSMedis the properties we had previously verified for CoSMed. The details of how these results were conceived are given in the CoSMedis paper [2], while a more succinct presentation can be found in [7].

The main result is a compositionality theorem, allowing to compose BD security policies for individual components specified as transition systems into a policy for the composition of systems specified as communicating products of transition systems. The theorem guarantees that the compound system obeys the compound policy provided that each component obeys its policy. There is a binary, as well as an N-ary version of the compositionality theorem, whose formalizations are presented in this document in sections with self-explanatory names.

Often, the composed policy does not have the most natural formulation of the desired confidentiality property. To help with reformulating it as a natural property (with the price of perhaps slightly weakening it), we have formalized a BD Security transport theorem. Moreover, we have a theorem that allows combining secret sources to form a stronger BD Security guarantee, which additionally excludes any leak arising from the collusion of the two sources; when this is possible, we call the secret sources *independent*. Finally, we have formalized some cases when BD security holds trivially, which are useful auxiliaries for the more complex results. All these results (for transporting, combining independent secret sources, and establishing security trivially), are again presented in sections with self-explanatory names.

As a matter of terminology and notation, this formalization (similarly to all our AFP formalizations involving BD security) differs from its main reference papers, namely [2] and [7] in that the secrets are called “values” (and consequently the type of secrets is denoted by “value”), and are ranged over by “v” rather than “s”. On the other hand, we use “s” (rather than “ σ ”) to range over states.

2 Binary compositionality theorem

This theory provides the binary version of the compositionality theorem for BD security. It corresponds to Theorem 1 from [2] and to Theorem 5 (the System Compositionality Theorem) from [7].

```
theory Composing-Security
  imports Bounded-Deducibility-Security.BD-Security-TS
begin
```

```
lemma list2-induct[case-names NilNil Cons1 Cons2]:
assumes NN:  $P \ [] \ []$ 
and CN:  $\bigwedge x \ xs \ ys. P \ xs \ ys \implies P \ (x \ # \ xs) \ ys$ 
and NC:  $\bigwedge xs \ y \ ys. P \ xs \ ys \implies P \ xs \ (y \ # \ ys)$ 
shows  $P \ xs \ ys$ 
proof (induction xs)
  case Nil show ?case using NN NC by (induction ys) auto next
  case (Cons x xs) then show ?case using CN by auto
qed
```

```
lemma list22-induct[case-names NilNil ConsNil NilCons ConsCons]:
assumes NN:  $P \ [] \ []$ 
and CN:  $\bigwedge x \ xs. P \ xs \ [] \implies P \ (x \ # \ xs) \ []$ 
and NC:  $\bigwedge y \ ys. P \ [] \ ys \implies P \ [] \ (y \ # \ ys)$ 
and CC:  $\bigwedge x \ xs \ y \ ys.
  P \ xs \ ys \implies
  (\bigwedge ys'. \text{length } ys' \leq \text{Suc } (\text{length } ys) \implies P \ xs \ ys') \implies
  (\bigwedge xs'. \text{length } xs' \leq \text{Suc } (\text{length } xs) \implies P \ xs' \ ys) \implies
  P \ (x \ # \ xs) \ (y \ # \ ys)$ 
shows  $P \ xs \ ys$ 
proof (induction rule: measure-induct2[of  $\lambda xs \ ys. \text{length } xs + \text{length } ys$ , case-names IH])
  case (IH xs ys) with assms show ?case by (cases xs; cases ys) auto
qed
```

```
context BD-Security-TS begin
```

```
declare O-append[simp]
declare V-append[simp]
declare validFrom-Cons[simp]
declare validFrom-append[simp]
```

```
declare list-all-O-map[simp]
declare never-O-Nil[simp]
declare list-all-V-map[simp]
```

```

declare never-V-Nil[simp]

end

locale Abstract-BD-Security-Comp =
  One: Abstract-BD-Security validSystemTraces1 V1 O1 B1 TT1 +
  Two: Abstract-BD-Security validSystemTraces2 V2 O2 B2 TT2 +
  Comp?: Abstract-BD-Security validSystemTraces V O B TT
for
  validSystemTraces1 :: 'traces1 ⇒ bool
and
  V1 :: 'traces1 ⇒ 'values1 and O1 :: 'traces1 ⇒ 'observations1
and
  TT1 :: 'traces1 ⇒ bool
and
  B1 :: 'values1 ⇒ 'values1 ⇒ bool
and

  validSystemTraces2 :: 'traces2 ⇒ bool
and
  V2 :: 'traces2 ⇒ 'values2 and O2 :: 'traces2 ⇒ 'observations2
and
  TT2 :: 'traces2 ⇒ bool
and
  B2 :: 'values2 ⇒ 'values2 ⇒ bool
and

  validSystemTraces :: 'traces ⇒ bool
and
  V :: 'traces ⇒ 'values and O :: 'traces ⇒ 'observations
and
  TT :: 'traces ⇒ bool
and
  B :: 'values ⇒ 'values ⇒ bool
+
fixes
  comp :: 'traces1 ⇒ 'traces2 ⇒ 'traces ⇒ bool
and
  compO :: 'observations1 ⇒ 'observations2 ⇒ 'observations ⇒ bool
and
  compV :: 'values1 ⇒ 'values2 ⇒ 'values ⇒ bool
assumes
validSystemTraces:
 $\bigwedge tr. \text{validSystemTraces } tr \implies$ 
 $(\exists tr1\ tr2. \text{validSystemTraces1 } tr1 \wedge \text{validSystemTraces2 } tr2 \wedge \text{comp } tr1\ tr2\ tr)$ 
and
V-comp:
 $\bigwedge tr1\ tr2\ tr.$ 

```

$validSystemTraces1\ tr1 \implies validSystemTraces2\ tr2 \implies comp\ tr1\ tr2\ tr$
 $\implies compV\ (V1\ tr1)\ (V2\ tr2)\ (V\ tr)$

and

O-comp:
 $\bigwedge\ tr1\ tr2\ tr.$
 $validSystemTraces1\ tr1 \implies validSystemTraces2\ tr2 \implies comp\ tr1\ tr2\ tr$
 $\implies compO\ (O1\ tr1)\ (O2\ tr2)\ (O\ tr)$

and

TT-comp:
 $\bigwedge\ tr1\ tr2\ tr.$
 $validSystemTraces1\ tr1 \implies validSystemTraces2\ tr2 \implies comp\ tr1\ tr2\ tr$
 $\implies TT\ tr \implies TT1\ tr1 \wedge TT2\ tr2$

and

B-comp:
 $\bigwedge\ vl1\ vl2\ vl\ vl'.$
 $compV\ vl1\ vl2\ vl \implies B\ vl\ vl'$
 $\implies \exists\ vl1'\ vl2'.\ compV\ vl1'\ vl2'\ vl' \wedge B1\ vl1\ vl1' \wedge B2\ vl2\ vl2'$

and

O-V-comp:
 $\bigwedge\ tr1\ tr2\ vl\ ol.$
 $validSystemTraces1\ tr1 \implies validSystemTraces2\ tr2 \implies$
 $compV\ (V1\ tr1)\ (V2\ tr2)\ vl \implies compO\ (O1\ tr1)\ (O2\ tr2)\ ol$
 $\implies \exists\ tr.\ validSystemTraces\ tr \wedge O\ tr = ol \wedge V\ tr = vl$

begin

abbreviation *secure* **where** $secure \equiv Comp.secure$

abbreviation *secure1* **where** $secure1 \equiv One.secure$

abbreviation *secure2* **where** $secure2 \equiv Two.secure$

theorem *secure1-secure2-secure:*

assumes $s1: secure1$ **and** $s2: secure2$

shows *secure*

unfolding *secure-def* **proof** *clarify*

fix $tr\ vl\ vl'$

assume $v: validSystemTraces\ tr$ **and** $T: TT\ tr$ **and** $B: B\ (V\ tr)\ vl'$

then obtain $tr1\ tr2$ **where** $v1: validSystemTraces1\ tr1$ **and** $v2: validSystemTraces2\ tr2$

and $tr: comp\ tr1\ tr2\ tr$ **using** *validSystemTraces* **by** *blast*

have $T1: TT1\ tr1$ **and** $T2: TT2\ tr2$ **using** *TT-comp*[*OF* $v1\ v2\ tr\ T$] **by** *auto*

have $Vtr: compV\ (V1\ tr1)\ (V2\ tr2)\ (V\ tr)$ **using** *V-comp*[*OF* $v1\ v2\ tr$].

have $Otr: compO\ (O1\ tr1)\ (O2\ tr2)\ (O\ tr)$ **using** *O-comp*[*OF* $v1\ v2\ tr$].

obtain $vl1'\ vl2'$ **where** $vl': compV\ vl1'\ vl2'\ vl'$ **and**

$B1: B1\ (V1\ tr1)\ vl1'$ **and** $B2: B2\ (V2\ tr2)\ vl2'$ **using** *B-comp*[*OF* $Vtr\ B$] **by** *auto*

obtain $tr1'\ tr2'$ **where** $v1': validSystemTraces1\ tr1'$ **and** $O1: O1\ tr1 = O1\ tr1'$ **and** $vl1': vl1' = V1\ tr1'$

using $s1\ v1\ T1\ B1$ **unfolding** *One.secure-def* **by** *fastforce*

obtain $tr2'\ vl2'$ **where** $v2': validSystemTraces2\ tr2'$ **and** $O2: O2\ tr2 = O2\ tr2'$ **and** $vl2': vl2' = V2\ tr2'$

```

using s2 v2 T2 B2 unfolding Two.secure-def by fastforce
obtain tr' where validSystemTraces tr'  $\wedge$  O tr' = O tr  $\wedge$  V tr' = vl'
using O-V-comp[OF v1' v2' vl'[unfolded vl1' vl2'] Otr[unfolded O1 O2]] by auto
thus  $\exists tr'. \text{validSystemTraces } tr' \wedge O tr' = O tr \wedge V tr' = vl'$  by auto
qed

end

```

```

type-synonym ('state1,'state2) cstate = 'state1  $\times$  'state2
datatype ('state1,'trans1,'state2,'trans2) ctrans = Trans1 'state2 'trans1 | Trans2 'state1 'trans2 | CTrans 'trans1 'trans2
datatype ('obs1,'obs2) cobs = Obs1 'obs1 | Obs2 'obs2 | CObs 'obs1 'obs2
datatype ('value1,'value2) cvalue = isValue1: Value1 'value1 | isValue2: Value2 'value2 | isCValue: CValue 'value1 'value2

```

```

locale BD-Security-TS-Comp =
  One: BD-Security-TS istate1 validTrans1 srcOf1 tgtOf1  $\varphi$ 1 f1  $\gamma$ 1 g1 T1 B1 +
  Two: BD-Security-TS istate2 validTrans2 srcOf2 tgtOf2  $\varphi$ 2 f2  $\gamma$ 2 g2 T2 B2
for
  istate1 :: 'state1 and validTrans1 :: 'trans1  $\Rightarrow$  bool
and
  srcOf1 :: 'trans1  $\Rightarrow$  'state1 and tgtOf1 :: 'trans1  $\Rightarrow$  'state1
and
   $\varphi$ 1 :: 'trans1  $\Rightarrow$  bool and f1 :: 'trans1  $\Rightarrow$  'value1
and
   $\gamma$ 1 :: 'trans1  $\Rightarrow$  bool and g1 :: 'trans1  $\Rightarrow$  'obs1
and
  T1 :: 'trans1  $\Rightarrow$  bool and B1 :: 'value1 list  $\Rightarrow$  'value1 list  $\Rightarrow$  bool
and
  istate2 :: 'state2 and validTrans2 :: 'trans2  $\Rightarrow$  bool
and
  srcOf2 :: 'trans2  $\Rightarrow$  'state2 and tgtOf2 :: 'trans2  $\Rightarrow$  'state2
and
   $\varphi$ 2 :: 'trans2  $\Rightarrow$  bool and f2 :: 'trans2  $\Rightarrow$  'value2
and
   $\gamma$ 2 :: 'trans2  $\Rightarrow$  bool and g2 :: 'trans2  $\Rightarrow$  'obs2
and
  T2 :: 'trans2  $\Rightarrow$  bool and B2 :: 'value2 list  $\Rightarrow$  'value2 list  $\Rightarrow$  bool
+
fixes
  isCom1 :: 'trans1  $\Rightarrow$  bool and isCom2 :: 'trans2  $\Rightarrow$  bool
and
  sync :: 'trans1  $\Rightarrow$  'trans2  $\Rightarrow$  bool
and
  isComV1 :: 'value1  $\Rightarrow$  bool and isComV2 :: 'value2  $\Rightarrow$  bool
and

```

$syncV :: 'value1 \Rightarrow 'value2 \Rightarrow bool$
and
 $isComO1 :: 'obs1 \Rightarrow bool$ **and** $isComO2 :: 'obs2 \Rightarrow bool$
and
 $syncO :: 'obs1 \Rightarrow 'obs2 \Rightarrow bool$

assumes
 $isCom1-isComV1: \bigwedge trn1. validTrans1 trn1 \Longrightarrow One.reach (srcOf1 trn1) \Longrightarrow \varphi1 trn1 \Longrightarrow isCom1 trn1 \longleftrightarrow isComV1 (f1 trn1)$
and
 $isCom1-isComO1: \bigwedge trn1. validTrans1 trn1 \Longrightarrow One.reach (srcOf1 trn1) \Longrightarrow \gamma1 trn1 \Longrightarrow isCom1 trn1 \longleftrightarrow isComO1 (g1 trn1)$
and
 $isCom2-isComV2: \bigwedge trn2. validTrans2 trn2 \Longrightarrow Two.reach (srcOf2 trn2) \Longrightarrow \varphi2 trn2 \Longrightarrow isCom2 trn2 \longleftrightarrow isComV2 (f2 trn2)$
and
 $isCom2-isComO2: \bigwedge trn2. validTrans2 trn2 \Longrightarrow Two.reach (srcOf2 trn2) \Longrightarrow \gamma2 trn2 \Longrightarrow isCom2 trn2 \longleftrightarrow isComO2 (g2 trn2)$
and
 $sync-syncV:$
 $\bigwedge trn1 trn2.$
 $validTrans1 trn1 \Longrightarrow One.reach (srcOf1 trn1) \Longrightarrow$
 $validTrans2 trn2 \Longrightarrow Two.reach (srcOf2 trn2) \Longrightarrow$
 $isCom1 trn1 \Longrightarrow isCom2 trn2 \Longrightarrow \varphi1 trn1 \Longrightarrow \varphi2 trn2 \Longrightarrow$
 $sync trn1 trn2 \Longrightarrow syncV (f1 trn1) (f2 trn2)$

and
 $sync-syncO:$
 $\bigwedge trn1 trn2.$
 $validTrans1 trn1 \Longrightarrow One.reach (srcOf1 trn1) \Longrightarrow$
 $validTrans2 trn2 \Longrightarrow Two.reach (srcOf2 trn2) \Longrightarrow$
 $isCom1 trn1 \Longrightarrow isCom2 trn2 \Longrightarrow \gamma1 trn1 \Longrightarrow \gamma2 trn2 \Longrightarrow$
 $sync trn1 trn2 \Longrightarrow syncO (g1 trn1) (g2 trn2)$

and
 $sync-\varphi1-\varphi2:$
 $\bigwedge trn1 trn2.$
 $validTrans1 trn1 \Longrightarrow One.reach (srcOf1 trn1) \Longrightarrow$
 $validTrans2 trn2 \Longrightarrow Two.reach (srcOf2 trn2) \Longrightarrow$
 $isCom1 trn1 \Longrightarrow isCom2 trn2 \Longrightarrow$
 $sync trn1 trn2 \Longrightarrow \varphi1 trn1 \longleftrightarrow \varphi2 trn2$

and
 $sync-\varphi-\gamma:$
 $\bigwedge trn1 trn2.$
 $validTrans1 trn1 \Longrightarrow One.reach (srcOf1 trn1) \Longrightarrow$
 $validTrans2 trn2 \Longrightarrow Two.reach (srcOf2 trn2) \Longrightarrow$
 $isCom1 trn1 \Longrightarrow isCom2 trn2 \Longrightarrow$
 $\gamma1 trn1 \Longrightarrow \gamma2 trn2 \Longrightarrow$
 $syncO (g1 trn1) (g2 trn2) \Longrightarrow$
 $(\varphi1 trn1 \Longrightarrow \varphi2 trn2 \Longrightarrow syncV (f1 trn1) (f2 trn2))$
 \Longrightarrow

```

    sync trn1 trn2
and
  isCom1- $\gamma$ 1:  $\bigwedge$  trn1. validTrans1 trn1  $\implies$  One.reach (srcOf1 trn1)  $\implies$  isCom1
trn1  $\implies$   $\gamma$ 1 trn1
and
  isCom2- $\gamma$ 2:  $\bigwedge$  trn2. validTrans2 trn2  $\implies$  Two.reach (srcOf2 trn2)  $\implies$  isCom2
trn2  $\implies$   $\gamma$ 2 trn2
and
  isCom2-V2:  $\bigwedge$  trn2. validTrans2 trn2  $\implies$  Two.reach (srcOf2 trn2)  $\implies$   $\varphi$ 2 trn2
 $\implies$  isCom2 trn2
and
  Dummy: istate1 = istate1  $\wedge$  srcOf1 = srcOf1  $\wedge$  tgtOf1 = tgtOf1  $\wedge$  T1 = T1  $\wedge$ 
B1 = B1  $\wedge$ 
    istate2 = istate2  $\wedge$  srcOf2 = srcOf2  $\wedge$  tgtOf2 = tgtOf2  $\wedge$  T2 = T2  $\wedge$  B2
= B2
begin

lemma sync- $\gamma$ 1- $\gamma$ 2:
 $\bigwedge$  trn1 trn2.
  validTrans1 trn1  $\implies$  One.reach (srcOf1 trn1)  $\implies$ 
  validTrans2 trn2  $\implies$  Two.reach (srcOf2 trn2)  $\implies$ 
  isCom1 trn1  $\implies$  isCom2 trn2  $\implies$ 
  sync trn1 trn2  $\implies$   $\gamma$ 1 trn1  $\iff$   $\gamma$ 2 trn2
using isCom1- $\gamma$ 1 isCom2- $\gamma$ 2
by auto

```

definition icstate **where** icstate = (istate1,istate2)

```

fun validTrans :: ('state1, 'trans1, 'state2, 'trans2) ctrans  $\Rightarrow$  bool where
  validTrans (Trans1 s2 trn1) = (validTrans1 trn1  $\wedge$   $\neg$  isCom1 trn1)
| validTrans (Trans2 s1 trn2) = (validTrans2 trn2  $\wedge$   $\neg$  isCom2 trn2)
| validTrans (CTrans trn1 trn2) =
  (validTrans1 trn1  $\wedge$  validTrans2 trn2  $\wedge$  isCom1 trn1  $\wedge$  isCom2 trn2  $\wedge$  sync
trn1 trn2)

```

```

fun srcOf :: ('state1, 'trans1, 'state2, 'trans2) ctrans  $\Rightarrow$  'state1  $\times$  'state2 where
  srcOf (Trans1 s2 trn1) = (srcOf1 trn1, s2)
| srcOf (Trans2 s1 trn2) = (s1, srcOf2 trn2)
| srcOf (CTrans trn1 trn2) = (srcOf1 trn1, srcOf2 trn2)

```

```

fun tgtOf :: ('state1, 'trans1, 'state2, 'trans2) ctrans  $\Rightarrow$  'state1  $\times$  'state2 where
  tgtOf (Trans1 s2 trn1) = (tgtOf1 trn1, s2)
| tgtOf (Trans2 s1 trn2) = (s1, tgtOf2 trn2)
| tgtOf (CTrans trn1 trn2) = (tgtOf1 trn1, tgtOf2 trn2)

```

```

fun  $\varphi$  :: ('state1, 'trans1, 'state2, 'trans2) ctrans  $\Rightarrow$  bool where
   $\varphi$  (Trans1 s2 trn1) =  $\varphi$ 1 trn1
|  $\varphi$  (Trans2 s1 trn2) =  $\varphi$ 2 trn2

```

$|\varphi (CTrans\ trn1\ trn2) = (\varphi1\ trn1 \vee \varphi2\ trn2)$

fun $f :: ('state1, 'trans1, 'state2, 'trans2) ctrans \Rightarrow ('value1, 'value2) cvalue$ **where**
 $f (Trans1\ s2\ trn1) = Value1\ (f1\ trn1)$
 $|f (Trans2\ s1\ trn2) = Value2\ (f2\ trn2)$
 $|f (CTrans\ trn1\ trn2) = CValue\ (f1\ trn1)\ (f2\ trn2)$

fun $\gamma :: ('state1, 'trans1, 'state2, 'trans2) ctrans \Rightarrow bool$ **where**
 $\gamma (Trans1\ s2\ trn1) = \gamma1\ trn1$
 $|\gamma (Trans2\ s1\ trn2) = \gamma2\ trn2$
 $|\gamma (CTrans\ trn1\ trn2) = (\gamma1\ trn1 \vee \gamma2\ trn2)$

fun $g :: ('state1, 'trans1, 'state2, 'trans2) ctrans \Rightarrow ('obs1, 'obs2) cobs$ **where**
 $g (Trans1\ s2\ trn1) = Obs1\ (g1\ trn1)$
 $|g (Trans2\ s1\ trn2) = Obs2\ (g2\ trn2)$
 $|g (CTrans\ trn1\ trn2) = CObs\ (g1\ trn1)\ (g2\ trn2)$

fun $T :: ('state1, 'trans1, 'state2, 'trans2) ctrans \Rightarrow bool$
where
 $T (Trans1\ s2\ trn1) = T1\ trn1$
 $|$
 $T (Trans2\ s1\ trn2) = T2\ trn2$
 $|$
 $T (CTrans\ trn1\ trn2) = (T1\ trn1 \vee T2\ trn2)$

inductive $compV :: 'value1\ list \Rightarrow 'value2\ list \Rightarrow ('value1, 'value2) cvalue\ list \Rightarrow bool$

where

$Nil[intro!,simp]: compV\ []\ []\ []$

$|Step1[intro]:$

$compV\ vl1\ vl2\ vl \Longrightarrow \neg isComV1\ v1$
 $\Longrightarrow compV\ (v1\ \#\ vl1)\ vl2\ (Value1\ v1\ \#\ vl)$

$|Step2[intro]:$

$compV\ vl1\ vl2\ vl \Longrightarrow \neg isComV2\ v2$
 $\Longrightarrow compV\ vl1\ (v2\ \#\ vl2)\ (Value2\ v2\ \#\ vl)$

$|Com[intro]:$

$compV\ vl1\ vl2\ vl \Longrightarrow isComV1\ v1 \Longrightarrow isComV2\ v2 \Longrightarrow syncV\ v1\ v2$
 $\Longrightarrow compV\ (v1\ \#\ vl1)\ (v2\ \#\ vl2)\ (CValue\ v1\ v2\ \#\ vl)$

lemma $compV\ cases\ V[consumes\ 3, case\ names\ Nil\ Step1\ Com]:$

assumes $v: Two.validFrom\ s2\ tr2$

and $c: compV\ vl1\ (Two.V\ tr2)\ vl$

and $rs2: Two.reach\ s2$

and $Nil: vl1 = [] \Longrightarrow Two.V\ tr2 = [] \Longrightarrow vl = [] \Longrightarrow P$

and $Step1:$

$\wedge vl1\ vl2\ vll\ v1.$

$vl1 = v1\ \#\ vll1 \Longrightarrow$

$Two.V\ tr2 = vll2 \Longrightarrow$

$vl = Value1\ v1\ \#\ vll \Longrightarrow$

$compV\ vll1\ vll2\ vll \implies \neg isComV1\ v1 \implies P$
and *Com*:
 $\wedge vll1\ vll2\ vll\ v1\ v2.$
 $vl1 = v1 \# vll1 \implies$
 $Two.V\ tr2 = v2 \# vll2 \implies$
 $vl = CValue\ v1\ v2 \# vll \implies$
 $compV\ vll1\ vll2\ vll \implies$
 $isComV1\ v1 \implies isComV2\ v2 \implies syncV\ v1\ v2 \implies P$
shows *P*
using *c proof cases*
case (*Step2 vll2 vll1 v2*)
obtain *tr2a trn2 tr2b* **where** *tr2: tr2 = tr2a @ trn2 # tr2b* **and**
 $\varphi2: \varphi2\ trn2$ **and** $f2: f2\ trn2 = v2$
using $\langle Two.V\ tr2 = v2 \# vll2 \rangle$ **by** (*metis Two.V-eq-Cons append-Cons*)
have *v2: validTrans2 trn2* **using** *tr2 v*
by (*metis Nil-is-append-conv Two.validFrom-def Two.valid-ConsE*
 $Two.valid-append\ list.distinct(2)\ self-append-conv2$)
have *rs2'*: *Two.reach (srcOf2 trn2)* **using** *v rs2* **unfolding** *tr2*
by (*induction tr2a arbitrary: s2*) (*auto intro: Two.reach.Step*)
then have *False* **using** *isCom2-V2[OF v2 rs2' $\varphi2$] $\langle \neg isComV2\ v2 \rangle$*
using $\varphi2\ f2\ isCom2-isComV2\ v2$ **by** *blast*
thus *?thesis* **by** *simp*
qed (*insert assms, auto*)

inductive *compO* :: $'obs1\ list \Rightarrow 'obs2\ list \Rightarrow ('obs1, 'obs2)\ cobs\ list \Rightarrow bool$
where
 $Nil[intro!,simp]: compO\ []\ []\ []$
 $|Step1[intro]:$
 $compO\ ol1\ ol2\ ol \implies \neg isComO1\ o1$
 $\implies compO\ (o1 \# ol1)\ ol2\ (Obs1\ o1 \# ol)$
 $|Step2[intro]:$
 $compO\ ol1\ ol2\ ol \implies \neg isComO2\ o2$
 $\implies compO\ ol1\ (o2 \# ol2)\ (Obs2\ o2 \# ol)$
 $|Com[intro]:$
 $compO\ ol1\ ol2\ ol \implies isComO1\ o1 \implies isComO2\ o2 \implies syncO\ o1\ o2$
 $\implies compO\ (o1 \# ol1)\ (o2 \# ol2)\ (CObs\ o1\ o2 \# ol)$

definition *B* :: $('value1, 'value2)\ cvalue\ list \Rightarrow ('value1, 'value2)\ cvalue\ list \Rightarrow bool$
where
 $B\ vl\ vl' \equiv \forall\ vl1\ vl2. compV\ vl1\ vl2\ vl \longrightarrow$
 $(\exists\ vl1'\ vl2'. compV\ vl1'\ vl2'\ vl' \wedge B1\ vl1\ vl1' \wedge B2\ vl2\ vl2')$

inductive *ccomp* ::
 $'state1 \Rightarrow 'state2 \Rightarrow 'trans1\ trace \Rightarrow 'trans2\ trace \Rightarrow$
 $('state1, 'trans1, 'state2, 'trans2)\ ctrans\ trace \Rightarrow bool$
where
 $Nil[simp,intro!]: ccomp\ s1\ s2\ []\ []\ []$
 $|$

```

Step1[intro]:
ccomp (tgtOf1 trn1) s2 tr1 tr2 tr  $\implies \neg$  isCom1 trn1  $\implies$ 
  ccomp (srcOf1 trn1) s2 (trn1 # tr1) tr2 (Trans1 s2 trn1 # tr)
|
Step2[intro]:
ccomp s1 (tgtOf2 trn2) tr1 tr2 tr  $\implies \neg$  isCom2 trn2  $\implies$ 
  ccomp s1 (srcOf2 trn2) tr1 (trn2 # tr2) (Trans2 s1 trn2 # tr)
|
Com[intro]:
ccomp (tgtOf1 trn1) (tgtOf2 trn2) tr1 tr2 tr  $\implies$ 
  isCom1 trn1  $\implies$  isCom2 trn2  $\implies$  sync trn1 trn2  $\implies$ 
  ccomp (srcOf1 trn1) s2 (trn1 # tr1) (trn2 # tr2) (CTrans trn1 trn2 # tr)

```

definition *comp* where $comp \equiv ccomp\ istate1\ istate2$

end

sublocale *BD-Security-TS-Comp* \subseteq *BD-Security-TS icstate validTrans srcOf tgtOf*
 $\varphi\ f\ \gamma\ g\ T\ B$.

context *BD-Security-TS-Comp*
begin

lemma *valid*:

assumes *valid tr* and $srcOf\ (hd\ tr) = (s1, s2)$

shows

$\exists\ tr1\ tr2$.

One.validFrom s1 tr1 \wedge *Two.validFrom s2 tr2* \wedge
ccomp s1 s2 tr1 tr2 tr

using *assms* **proof**(*induction arbitrary: s1 s2*)

case (*Singl trn*)

show *?case* **proof**(*cases trn*)

case (*Trans1 s22 trn1*)

show *?thesis* **using** *Singl unfolding Trans1*

by (*intro exI[of - [trn1]] exI[of - []]*) *auto*

next

case (*Trans2 s11 trn2*)

show *?thesis* **using** *Singl unfolding Trans2*

by (*intro exI[of - []::'trans1 trace] exI[of - [trn2]]*) *auto*

next

case (*CTrans trn1 trn2*)

show *?thesis* **using** *Singl unfolding CTrans*

by (*intro exI[of - [trn1]] exI[of - [trn2]]*) *auto*

qed

next

case (*Cons trn tr*)

show *?case* **proof**(*cases trn*)

case (*Trans1 s22 trn1*)

```

let ?s1 = tgtOf1 trn1
have s22[simp]: s22 = s2 using ⟨srcOf (hd (trn # tr)) = (s1, s2)⟩
unfolding Trans1 by simp
hence tgtOf trn = (?s1, s2) unfolding Trans1 by simp
hence srcOf (hd tr) = (?s1, s2) using Cons.hyps(2) by auto
from Cons.IH[OF this] obtain tr1 tr2 where
1: One.validFrom ?s1 tr1 ∧ Two.validFrom s2 tr2 ∧
   ccomp ?s1 s2 tr1 tr2 tr by auto
show ?thesis using Cons.premis Cons.hyps 1 unfolding Trans1
by (intro exI[of - trn1 # tr1] exI[of - tr2], cases tr2) auto
next
case (Trans2 s11 trn2)
let ?s2 = tgtOf2 trn2
have s11[simp]: s11 = s1 using ⟨srcOf (hd (trn # tr)) = (s1, s2)⟩
unfolding Trans2 by simp
hence tgtOf trn = (s1, ?s2) unfolding Trans2 by simp
hence srcOf (hd tr) = (s1, ?s2) using Cons.hyps(2) by auto
from Cons.IH[OF this] obtain tr1 tr2 where
1: One.validFrom s1 tr1 ∧ Two.validFrom ?s2 tr2 ∧
   ccomp s1 ?s2 tr1 tr2 tr by auto
show ?thesis using Cons.premis Cons.hyps 1 unfolding Trans2
by (intro exI[of - tr1] exI[of - trn2 # tr2], cases tr1) auto
next
case (CTrans trn1 trn2)
let ?s1 = tgtOf1 trn1 let ?s2 = tgtOf2 trn2
have tgtOf trn = (?s1, ?s2) unfolding CTrans by simp
hence srcOf (hd tr) = (?s1, ?s2) using Cons.hyps(2) by auto
from Cons.IH[OF this] obtain tr1 tr2 where
1: One.validFrom ?s1 tr1 ∧ Two.validFrom ?s2 tr2 ∧
   ccomp ?s1 ?s2 tr1 tr2 tr by auto
show ?thesis using Cons.premis Cons.hyps 1 unfolding CTrans
by (intro exI[of - trn1 # tr1] exI[of - trn2 # tr2], cases tr2) auto
qed
qed

```

lemma *validFrom*:

assumes *validFrom icstate tr*

shows $\exists tr1 tr2. One.validFrom icstate1 tr1 \wedge Two.validFrom icstate2 tr2 \wedge comp\ tr1\ tr2\ tr$

using *assms valid unfolding comp-def icstate-def validFrom-def* **by** (cases tr) *fast-force+*

lemma *reach-reach12*:

assumes *reach s*

obtains *One.reach (fst s)* **and** *Two.reach (snd s)*

using *assms proof* (induction rule: *reach.induct*)

case *Istate*

then show *thesis* **using** *One.reach.Istate Two.reach.Istate* **unfolding** *ic-state-def* **by** *auto*

```

next
  case (Step s trn s^)
  then show thesis
    using One.reach.Step[of fst s - fst s^] Two.reach.Step[of snd s - snd s^]
    by (auto elim: validTrans.elims)
qed

lemma compV-ccomp:
assumes v: One.validFrom s1 tr1 Two.validFrom s2 tr2
and c: ccomp s1 s2 tr1 tr2 tr
and rs1: One.reach s1 and rs2: Two.reach s2
shows compV (One.V tr1) (Two.V tr2) (V tr)
using c v rs1 rs2 proof(induction)
  case (Step1 trn1 s2 tr1 tr2 tr)
  moreover then have One.reach (tgtOf1 trn1)
    using One.reach.Step[of srcOf1 trn1 trn1 tgtOf1 trn1] by auto
  ultimately show ?case by (cases  $\varphi_1$  trn1) (auto simp: isCom1-isComV1)
next
  case (Step2 s1 trn2 tr1 tr2 tr)
  moreover then have Two.reach (tgtOf2 trn2)
    using Two.reach.Step[of srcOf2 trn2 trn2 tgtOf2 trn2] by auto
  ultimately show ?case by (cases  $\varphi_2$  trn2) (auto simp: isCom2-isComV2)
next
  case (Com trn1 trn2 tr1 tr2 tr s2)
  moreover then have One.reach (tgtOf1 trn1) Two.reach (tgtOf2 trn2)
    using One.reach.Step[of srcOf1 trn1 trn1 tgtOf1 trn1]
      Two.reach.Step[of srcOf2 trn2 trn2 tgtOf2 trn2]
    by auto
  ultimately show ?case
    by (cases  $\varphi_1$  trn1; cases  $\varphi_2$  trn2; simp add: isCom1-isComV1 isCom2-isComV2)
      (use sync- $\varphi_1$ - $\varphi_2$  sync-syncV Com in auto)
qed auto

lemma compV:
assumes One.validFrom istate1 tr1 and Two.validFrom istate2 tr2
and comp tr1 tr2 tr
shows compV (One.V tr1) (Two.V tr2) (V tr)
using compV-ccomp assms One.reach.Istate Two.reach.Istate unfolding comp-def
by auto

lemma compO-ccomp:
assumes v: One.validFrom s1 tr1 Two.validFrom s2 tr2
and c: ccomp s1 s2 tr1 tr2 tr
and rs1: One.reach s1 and rs2: Two.reach s2
shows compO (One.O tr1) (Two.O tr2) (O tr)
using c v rs1 rs2 proof(induction)
  case (Step1 trn1 s2 tr1 tr2 tr)
  moreover then have One.reach (tgtOf1 trn1)
    using One.reach.Step[of srcOf1 trn1 trn1 tgtOf1 trn1] by auto

```

```

ultimately show ?case by (cases  $\gamma_1$  trn1) (auto simp: isCom1-isComO1)
next
case (Step2 s1 trn2 tr1 tr2 tr)
moreover then have Two.reach (tgtOf2 trn2)
  using Two.reach.Step[of srcOf2 trn2 trn2 tgtOf2 trn2] by auto
ultimately show ?case by (cases  $\gamma_2$  trn2) (auto simp: isCom2-isComO2)
next
case (Com trn1 trn2 tr1 tr2 tr s2)
moreover then have One.reach (tgtOf1 trn1) Two.reach (tgtOf2 trn2)
  using One.reach.Step[of srcOf1 trn1 trn1 tgtOf1 trn1]
    Two.reach.Step[of srcOf2 trn2 trn2 tgtOf2 trn2]
  by auto
ultimately show ?case
  by (cases  $\gamma_1$  trn1; cases  $\gamma_2$  trn2; simp add: isCom1-isComO1 isCom2-isComO2)
    (use sync- $\gamma_1$ - $\gamma_2$  sync-syncO Com in auto)
qed auto

```

lemma compO:

```

assumes One.validFrom istate1 tr1 and Two.validFrom istate2 tr2
and comp tr1 tr2 tr
shows compO (One.O tr1) (Two.O tr2) (O tr)
using compO-ccomp assms One.reach.Istate Two.reach.Istate unfolding comp-def
by auto

```

lemma T-ccomp:

```

assumes v: One.validFrom s1 tr1 Two.validFrom s2 tr2
and c: ccomp s1 s2 tr1 tr2 tr and n: never T tr
shows never T1 tr1  $\wedge$  never T2 tr2
using c n v by (induction) auto

```

lemma T:

```

assumes One.validFrom istate1 tr1 and Two.validFrom istate2 tr2
and comp tr1 tr2 tr and never T tr
shows never T1 tr1  $\wedge$  never T2 tr2
using T-ccomp assms unfolding comp-def by auto

```

lemma B:

```

assumes compV vl1 vl2 vl and B vl vl'
shows  $\exists vl1' vl2'. compV vl1' vl2' vl' \wedge B1 vl1 vl1' \wedge B2 vl2 vl2'$ 
using assms unfolding B-def by auto

```

lemma pullback-O-V-aux:

```

assumes One.validFrom s1 tr1 Two.validFrom s2 tr2
and One.reach s1 Two.reach s2
and compV (One.V tr1) (Two.V tr2) vl
and compO (One.O tr1) (Two.O tr2) obl
shows  $\exists tr. validFrom (s1, s2) tr \wedge O tr = obl \wedge V tr = vl$ 
using assms proof(induction tr1 tr2 arbitrary: s1 s2 vl obl rule: list22-induct)
  case (NilNil s1 s2 vl obl)

```

```

thus ?case by (intro exI[of - []]) (auto elim: compV.cases compO.cases)
next
  case (ConsNil trn1 tr1 s1 s2 vl obl)
  let ?s1 = tgtOf1 trn1
  have trn1: validTrans1 trn1 and tr1: One.validFrom ?s1 tr1
  and s1: srcOf1 trn1 = s1 One.reach s1 and rs2: Two.reach s2 using ConsNil.premis by auto
  then have rs1': One.reach ?s1 by (intro One.reach.Step[of s1 trn1 ?s1]) auto
  show ?case proof(cases isCom1 trn1)
    case True note com1 = True
    hence  $\gamma_1$ :  $\gamma_1$  trn1 using trn1 isCom1- $\gamma_1$  s1 by auto
    hence isComO1 (g1 trn1) using  $\gamma_1$  com1 s1 isCom1-isComO1 trn1 by blast
    hence False using  $\langle$ compO (One.O (trn1 # tr1)) (Two.O []) obl $\rangle$ 
    using  $\gamma_1$  by (auto elim: compO.cases)
    thus ?thesis by simp
  next
    case False note com1 = False
    show ?thesis proof(cases  $\varphi_1$  trn1)
      case True note  $\varphi_1$  = True
      hence comv1:  $\neg$  isComV1 (f1 trn1) using  $\varphi_1$  com1 isCom1-isComV1 trn1
    s1 by blast
    with  $\langle$ compV (One.V (trn1 # tr1)) (Two.V []) vl $\rangle$   $\varphi_1$ 
    obtain vll where vl: vl = Value1 (f1 trn1) # vll
    and vll: compV (One.V tr1) (Two.V []) vll by (auto elim: compV.cases)
    show ?thesis proof(cases  $\gamma_1$  trn1)
      case True note  $\gamma_1$  = True
      hence  $\neg$  isComO1 (g1 trn1) using  $\gamma_1$  com1 isCom1-isComO1 trn1 s1 by
    blast
      with  $\langle$ compO (One.O (trn1 # tr1)) (Two.O []) obl $\rangle$   $\gamma_1$ 
      obtain obll where obl: obl = Obs1 (g1 trn1) # obll
      and obll: compO (One.O tr1) (Two.O []) obll by (auto elim: compO.cases)
      from ConsNil.IH[OF tr1 - rs1' rs2 vll obll] obtain trr where
        validFrom (?s1, s2) trr and O trr = obll  $\wedge$  V trr = vll by auto
      thus ?thesis
      unfolding obl vl using trn1 com1 s1  $\varphi_1$   $\gamma_1$ 
      by (intro exI[of - Trans1 s2 trn1 # trr]) auto
    next
      case False note  $\gamma_1$  = False
      note obl =  $\langle$ compO (One.O (trn1 # tr1)) (Two.O []) obl $\rangle$ 
      from ConsNil.IH[OF tr1 - rs1' rs2 vll] obl  $\gamma_1$  obtain trr where
        validFrom (?s1, s2) trr and O trr = obl  $\wedge$  V trr = vll by auto
      thus ?thesis
      unfolding obl vl using trn1 com1 s1  $\varphi_1$   $\gamma_1$ 
      by (intro exI[of - Trans1 s2 trn1 # trr]) auto
    qed
  next
    case False note  $\varphi_1$  = False
    note vl =  $\langle$ compV (One.V (trn1 # tr1)) (Two.V []) vl $\rangle$ 
    show ?thesis proof(cases  $\gamma_1$  trn1)

```

```

case True note  $\gamma 1 = True$ 
hence  $\neg isComO1 (g1\ trn1)$  using  $\gamma 1\ com1\ isCom1-isComO1\ trn1\ s1$  by
blast
with  $\langle compO (One.O (trn1 \# tr1)) (Two.O []) \rangle obl$   $\gamma 1$ 
obtain  $obl$  where  $obl = Obs1 (g1\ trn1) \# obl$ 
and  $obl$ :  $compO (One.O\ tr1) (Two.O\ [])\ obl$  by  $(auto\ elim: compO.cases)$ 
from  $ConsNil.IH[OF\ tr1 - rs1' rs2 - obl]$   $vl\ \varphi 1$  obtain  $trr$  where
 $validFrom (?s1, s2)\ trr$  and  $O\ trr = obl \wedge V\ trr = vl$  by  $auto$ 
thus  $?thesis$ 
unfolding  $obl\ vl$  using  $trn1\ com1\ s1\ \varphi 1\ \gamma 1$ 
by  $(intro\ exI[of - Trans1\ s2\ trn1 \# trr])\ auto$ 
next
case False note  $\gamma 1 = False$ 
note  $obl = \langle compO (One.O (trn1 \# tr1)) (Two.O []) \rangle obl$ 
from  $ConsNil.IH[OF\ tr1 - rs1' rs2 -]$   $vl\ \varphi 1\ obl\ \gamma 1$  obtain  $trr$  where
 $validFrom (?s1, s2)\ trr$  and  $O\ trr = obl \wedge V\ trr = vl$  by  $fastforce$ 
thus  $?thesis$ 
unfolding  $obl\ vl$  using  $trn1\ com1\ s1\ \varphi 1\ \gamma 1$ 
by  $(intro\ exI[of - Trans1\ s2\ trn1 \# trr])\ auto$ 
qed
qed
qed
next
case  $(NilCons\ trn2\ tr2\ s1\ s2\ vl\ obl)$ 
let  $?s2 = tgtOf2\ trn2$ 
have  $trn2$ :  $validTrans2\ trn2$  and  $tr2$ :  $Two.validFrom\ ?s2\ tr2$ 
and  $s2$ :  $srcOf2\ trn2 = s2$   $Two.reach\ s2$  and  $rs1$ :  $One.reach\ s1$  using  $Nil-$ 
 $Cons.prem$ s by  $auto$ 
then have  $rs2'$ :  $Two.reach\ ?s2$  by  $(intro\ Two.reach.Step[of\ s2\ trn2\ ?s2])\ auto$ 
show  $?case$  proof( $cases\ isCom2\ trn2$ )
case True note  $com2 = True$ 
hence  $\gamma 2$ :  $\gamma 2\ trn2$  using  $trn2\ isCom2-\gamma 2\ s2$  by  $auto$ 
hence  $isComO2 (g2\ trn2)$  using  $\gamma 2\ com2\ isCom2-isComO2\ trn2\ s2$  by  $blast$ 
hence  $False$  using  $\langle compO (One.O []) (Two.O (trn2 \# tr2)) \rangle obl$ 
using  $\gamma 2$  by  $(auto\ elim: compO.cases)$ 
thus  $?thesis$  by  $simp$ 
next
case False note  $com2 = False$ 
show  $?thesis$  proof( $cases\ \varphi 2\ trn2$ )
case True note  $\varphi 2 = True$ 
hence  $comv1$ :  $\neg isComV2 (f2\ trn2)$  using  $\varphi 2\ com2\ isCom2-isComV2\ trn2$ 
 $s2$  by  $blast$ 
with  $\langle compV (One.V []) (Two.V (trn2 \# tr2)) \rangle vl$   $\varphi 2$ 
obtain  $vll$  where  $vl$ :  $vl = Value2 (f2\ trn2) \# vll$ 
and  $vll$ :  $compV (One.V []) (Two.V\ tr2)\ vll$  by  $(auto\ elim: compV.cases)$ 
show  $?thesis$  proof( $cases\ \gamma 2\ trn2$ )
case True note  $\gamma 2 = True$ 
hence  $\neg isComO2 (g2\ trn2)$  using  $\gamma 2\ com2\ isCom2-isComO2\ trn2\ s2$  by
blast

```

```

with ⟨compO (One.O []) (Two.O (trn2 # tr2)) obl⟩ γ2
obtain obll where obl: obl = Obs2 (g2 trn2) # obll
and obll: compO (One.O []) (Two.O tr2) obll by (auto elim: compO.cases)
from NilCons.IH[OF - tr2 rs1 rs2' vll obll] obtain trr where
validFrom (s1, ?s2) trr and O trr = obll ∧ V trr = vll by auto
thus ?thesis
unfolding obl vl using trn2 com2 s2 φ2 γ2
by (intro exI[of - Trans2 s1 trn2 # trr]) auto
next
case False note γ2 = False
note obl = ⟨compO (One.O []) (Two.O (trn2 # tr2)) obl⟩
from NilCons.IH[OF - tr2 rs1 rs2' vll] obl γ2 obtain trr where
validFrom (s1, ?s2) trr and O trr = obl ∧ V trr = vll by auto
thus ?thesis
unfolding obl vl using trn2 com2 s2 φ2 γ2
by (intro exI[of - Trans2 s1 trn2 # trr]) auto
qed
next
case False note φ2 = False
note vl = ⟨compV (One.V []) (Two.V (trn2 # tr2)) vl⟩
show ?thesis proof(cases γ2 trn2)
case True note γ2 = True
hence ¬ isComO2 (g2 trn2) using γ2 com2 isCom2-isComO2 trn2 s2 by
blast
with ⟨compO (One.O []) (Two.O (trn2 # tr2)) obl⟩ γ2
obtain obll where obl: obl = Obs2 (g2 trn2) # obll
and obll: compO (One.O []) (Two.O tr2) obll by (auto elim: compO.cases)
from NilCons.IH[OF - tr2 rs1 rs2' - obll] vl φ2 obtain trr where
validFrom (s1, ?s2) trr and O trr = obll ∧ V trr = vl by auto
thus ?thesis
unfolding obl vl using trn2 com2 s2 φ2 γ2
by (intro exI[of - Trans2 s1 trn2 # trr]) auto
next
case False note γ2 = False
note obl = ⟨compO (One.O []) (Two.O (trn2 # tr2)) obl⟩
from NilCons.IH[OF - tr2 rs1 rs2' -] vl φ2 obl γ2 obtain trr where
validFrom (s1, ?s2) trr and O trr = obl ∧ V trr = vl by fastforce
thus ?thesis
unfolding obl vl using trn2 com2 s2 φ2 γ2
by (intro exI[of - Trans2 s1 trn2 # trr]) auto
qed
qed
next
case (ConsCons trn1 tr1 trn2 tr2 s1 s2 vl obl)
let ?s1 = tgtOf1 trn1 let ?s2 = tgtOf2 trn2
let ?tr1 = trn1 # tr1 let ?tr2 = trn2 # tr2
have trn1: validTrans1 trn1 and tr1: One.validFrom ?s1 tr1 and s1: srcOf1
trn1 = s1 One.reach s1

```

```

and trn2: validTrans2 trn2 and tr2: Two.validFrom ?s2 tr2 and s2: srcOf2 trn2
= s2 Two.reach s2
using ConsCons.premis by auto
then have rs1': One.reach ?s1 and rs2': Two.reach ?s2
  using One.reach.Step[of s1 trn1 ?s1] Two.reach.Step[of s2 trn2 ?s2] by auto
note vl = ⟨compV (One.V ?tr1) (Two.V ?tr2) vl⟩
note obl = ⟨compO (One.O ?tr1) (Two.O ?tr2) obl⟩
note trr1 = ⟨One.validFrom s1 ?tr1⟩ note trr2 = ⟨Two.validFrom s2 ?tr2⟩
show ?case proof(cases φ1 trn1 ∨ γ1 trn1)
  case False note φγ1 = False
  hence com1: ¬ isCom1 trn1 using isCom1-γ1 trn1 s1 by blast
  from ConsCons.IH(2)[of ?tr2, OF - tr1 trr2 rs1' s2(2)] vl obl φγ1
  obtain trr where validFrom (?s1, s2) trr and O trr = obl ∧ V trr = vl
  by fastforce
  thus ?thesis
  unfolding obl vl using trn1 com1 s1 φγ1
  by (intro exI[of - Trans1 s2 trn1 # trr]) auto
next
  case True note φγ1 = True
  show ?thesis proof(cases φ2 trn2 ∨ γ2 trn2)
    case False note φγ2 = False
    hence com2: ¬ isCom2 trn2 using isCom2-γ2 trn2 s2 by blast
    from ConsCons.IH(3)[of ?tr1, OF - trr1 tr2 s1(2) rs2'] vl obl φγ2
    obtain trr where validFrom (s1, ?s2) trr and O trr = obl ∧ V trr = vl
    by fastforce
    thus ?thesis
    unfolding obl vl using trn2 com2 s2 φγ2
    by (intro exI[of - Trans2 s1 trn2 # trr]) auto
  next
  case True note φγ2 = True
  show ?thesis using obl ConsCons proof cases
    case Nil hence γ12: ¬ γ1 trn1 ∧ ¬ γ2 trn2 by auto
    hence obl: compO (One.O tr1) (Two.O ?tr2) obl
      compO (One.O tr1) (Two.O tr2) obl
    using obl by auto
    have φ12: φ1 trn1 ∧ φ2 trn2 using φγ1 φγ2 γ12 by auto
    show ?thesis using trr2 vl s2(2) proof(cases rule: compV-cases-V)
      case Nil hence False using φ12 by auto
      thus ?thesis by simp
    next
    case (Step1 vll1 vll2 vll v1)
    hence f1: f1 trn1 = v1 and vll1: One.V tr1 = vll1 using φ12 by auto
    hence vll: compV (One.V tr1) (Two.V ?tr2) vll using Step1 by auto
    from ConsCons.IH(2)[OF - tr1 trr2 rs1' s2(2) vll obl(1)]
    obtain trr where validFrom (?s1, s2) trr and O trr = obl ∧ V trr = vll
    by auto
    thus ?thesis using trn1 Step1 f1 φ12 γ12 isCom2-V2 isCom2-γ2 trn2 s2
    by (intro exI[of - Trans1 s2 trn1 # trr]) auto
  next

```

case (*Com vll1 vll2 vll v1 v2*)
hence *f1: f1 trn1 = v1 and vll1: One.V tr1 = vll1*
and *f2: f2 trn2 = v2 and vll2: Two.V tr2 = vll2*
using $\varphi12$ **by** *auto*
hence *vll: compV (One.V tr1) (Two.V tr2) vll using Com by auto*
from *ConsCons.IH(1)[OF tr1 tr2 rs1' rs2' vll obl(2)]*
obtain *trr where validFrom (?s1, ?s2) trr and O trr = obl \wedge V trr =*
vll
by *auto*
thus *?thesis using trn1 Step1 f1 $\varphi12$ $\gamma12$ isCom2-V2 isCom2- $\gamma2$ trn2 s2*
by (*intro exI[of - Trans1 s2 trn1 # trr]*) *auto*
qed
next
case (*Step1 obll1 obll ob1*) **note** *Step1O = Step1*
show *?thesis proof(cases $\gamma1$ trn1)*
case *True note $\gamma1 = True$*
hence *g1: g1 trn1 = ob1 and obll1 = One.O tr1 using Step1 by auto*
hence *obll: compO (One.O tr1) (Two.O ?tr2) obll using Step1 by auto*
have *com1: \neg isCom1 trn1 using Step1O $\gamma1$ g1 isCom1-isComO1 trn1*
s1 by blast
show *?thesis using trr2 vl s2(2) proof(cases rule: compV-cases-V)*
case *Nil*
hence $\varphi12$: *$\neg \varphi1$ trn1 $\wedge \neg \varphi2$ trn2 by auto*
hence *vl: compV (One.V tr1) (Two.V ?tr2) vl using vl by auto*
from *ConsCons.IH(2)[OF - tr1 trr2 rs1' s2(2) vl obll]*
obtain *trr where validFrom (?s1, s2) trr and O trr = obll \wedge V trr =*
vl
by *auto*
thus *?thesis using trn1 Step1O g1 $\varphi12$ $\gamma1$ trn1 s1 isCom1-isComO1*
by (*intro exI[of - Trans1 s2 trn1 # trr]*) *auto*
next
case (*Step1 vll1 vll2 vll v1*) **note** *Step1V = Step1*
show *?thesis proof(cases $\varphi1$ trn1)*
case *False note $\varphi1 = False$*
hence *vl: compV (One.V tr1) (Two.V ?tr2) vl using vl by auto*
from *ConsCons.IH(2)[OF - tr1 trr2 rs1' s2(2) vl obll]*
obtain *trr where validFrom (?s1, s2) trr and O trr = obll \wedge V trr =*
= vl
by *auto*
thus *?thesis using trn1 Step1O g1 $\varphi1$ $\gamma1$ trn1 s1 isCom1-isComO1*
by (*intro exI[of - Trans1 s2 trn1 # trr]*) *auto*
next
case *True note $\varphi1 = True$*
hence *f1: f1 trn1 = v1 and vll1 = One.V tr1 using Step1V by auto*
hence *vll: compV (One.V tr1) (Two.V ?tr2) vll using Step1V com1*
by auto
from *ConsCons.IH(2)[OF - tr1 trr2 rs1' s2(2) vll obll]*
obtain *trr where validFrom (?s1, s2) trr and O trr = obll \wedge V trr =*
= vll

```

      by auto
      thus ?thesis using trn1 Step1O Step1V f1  $\varphi$ 1 g1  $\varphi$ 1  $\gamma$ 1 trn1 s1
isCom1-isComO1
      by (intro exI[of - Trans1 s2 trn1 # trr]) auto
      qed
    next
      case (Com vll1 vll2 vll v1 v2)
      hence  $\varphi$ 1:  $\neg \varphi$ 1 trn1 using com1 isCom1-isComV1[OF trn1] s1 by
auto
      hence vl: compV (One.V tr1) (Two.V ?tr2) vl using vl by auto
      from ConsCons.IH(2)[OF - tr1 trr2 rs1' s2(2) vl obl]
      obtain trr where validFrom (?s1, s2) trr and O trr = obl  $\wedge$  V trr =
vl
      by auto
      thus ?thesis using trn1 Step1O g1  $\varphi$ 1  $\gamma$ 1 trn1 s1 isCom1-isComO1
      by (intro exI[of - Trans1 s2 trn1 # trr]) auto
      qed
    next
      case False note  $\gamma$ 1 = False
      hence obl: compO (One.O tr1) (Two.O ?tr2) obl using obl by simp
      hence  $\varphi$ 1:  $\varphi$ 1 trn1 and com1:  $\neg$  isCom1 trn1 using  $\varphi$  $\gamma$ 1  $\gamma$ 1 isCom1- $\gamma$ 1
trn1 s1 by auto
      show ?thesis using trr2 vl s2(2) proof(cases rule: compV-cases-V)
        case Nil hence False using  $\varphi$ 1 by auto
        thus ?thesis by simp
      next
      case Com hence False using  $\varphi$ 1 com1 trn1 using isCom1-isComV1 s1
by auto
      thus ?thesis by simp
    next
      case (Step1 vll1 vll2 vll v1) note Step1V = Step1
      hence f1: f1 trn1 = v1 and vll1 = One.V tr1 using  $\varphi$ 1 by auto
      hence vll: compV (One.V tr1) (Two.V ?tr2) vll using Step1V com1
by auto
      from ConsCons.IH(2)[OF - tr1 trr2 rs1' s2(2) vll obl]
      obtain trr where validFrom (?s1, s2) trr and O trr = obl  $\wedge$  V trr =
vll
      by auto
      thus ?thesis using trn1 Step1O Step1V f1  $\varphi$ 1  $\varphi$ 1  $\gamma$ 1 trn1 s1 is-
Com1-isComV1
      by (intro exI[of - Trans1 s2 trn1 # trr]) auto
      qed
    qed
  next
    case (Step2 obl2 obl ob2) note Step2O = Step2
    hence com2:  $\neg$  isCom2 trn2 using isCom2- $\gamma$ 2[OF trn2] isCom2-isComO2[OF
trn2] s2 by auto
    hence  $\varphi$ 2:  $\neg \varphi$ 2 trn2 using isCom2-V2[OF trn2] s2 by auto
    hence vl: compV (One.V ?tr1) (Two.V tr2) vl using vl by simp

```

```

    have  $\gamma_2$ :  $\gamma_2$  trn2 using  $\varphi\gamma_2$   $\varphi_2$  by simp
    hence g2:  $g_2$  trn2 = ob2 and obl2 = Two.O tr2 using Step2 by auto
    hence obl: compO (One.O ?tr1) (Two.O tr2) obl using Step2 by auto
    from ConsCons.IH(3)[OF - trr1 tr2 s1(2) rs2' vl obl]
    obtain trr where validFrom (s1, ?s2) trr and O trr = obl  $\wedge$  V trr = vl
  by auto
    thus ?thesis using trn1 Step2O g2  $\varphi_2$   $\gamma_2$  trn2 s2 isCom2-isComO2
    by (intro exI[of - Trans2 s1 trn2 # trr]) auto
  next
    case (Com obll1 obll2 obl ob1 ob2) note ComO = Com
    show ?thesis
    proof(cases  $\gamma_1$  trn1)
      case True note  $\gamma_1$  = True
      hence com1: isCom1 trn1 using isCom1-isComO1[OF trn1] s1 ComO
    by auto
      show ?thesis proof(cases  $\gamma_2$  trn2)
        case True note  $\gamma_2$  = True
        hence com2: isCom2 trn2 using isCom2-isComO2[OF trn2] s2 ComO
      by auto
    have obl: compO (One.O tr1) (Two.O tr2) obl using obl ComO  $\gamma_1$   $\gamma_2$ 
  by auto
    have g1:  $g_1$  trn1 = ob1 and obll1 = One.O tr1 and
      g2:  $g_2$  trn2 = ob2 and obll2 = Two.O tr2
    using  $\gamma_1$   $\gamma_2$  ComO by auto
    have rs1: One.reach (srcOf1 trn1) and rs2: Two.reach (srcOf2 trn2)
      using s1 s2 by auto
    have sync: sync trn1 trn2 proof(rule sync- $\varphi$ - $\gamma$ [OF trn1 rs1 trn2 rs2
com1 com2])
      show syncO (g1 trn1) (g2 trn2) using Com  $\gamma_1$   $\gamma_2$  by auto
    next
      assume  $\varphi_{12}$ :  $\varphi_1$  trn1  $\varphi_2$  trn2
      hence comV: isComV1 (f1 trn1)  $\wedge$  isComV2 (f2 trn2)
      using com1 com2 isCom1-isComV1 isCom2-isComV2 trn1 trn2 rs1 rs2
    by blast
      show syncV (f1 trn1) (f2 trn2) using vl  $\varphi_{12}$  comV by cases auto
    qed(insert  $\gamma_1$   $\gamma_2$ , auto)
    show ?thesis
    proof(cases  $\varphi_1$  trn1)
      case True
      hence  $\varphi_{12}$ :  $\varphi_1$  trn1  $\wedge$   $\varphi_2$  trn2 using sync- $\varphi_1$ - $\varphi_2$ [OF trn1 rs1 trn2
rs2 com1 com2 sync] by simp
      show ?thesis using trr2 vl s2(2) proof(cases rule: compV-cases-V)
        case Nil hence False using  $\varphi_{12}$  by auto
        thus ?thesis by simp
      next
        case Step1 hence False using  $\varphi_{12}$  com1 isCom1-isComV1[OF trn1]
    s1 by auto
      thus ?thesis by simp
    next

```

```

      case (Com vll1 vll2 vll v1 v2) note ComV = Com
      hence f1: f1 trn1 = v1 and vll1 = One.V tr1 and
            f2: f2 trn2 = v2 and vll2 = Two.V tr2
      using  $\varphi12$  by auto
      hence vll: compV (One.V tr1) (Two.V tr2) vll using ComV com1
com2  $\varphi12$  by auto
      from ConsCons.IH(1)[OF tr1 tr2 rs1' rs2' vll obl]
      obtain trr where validFrom (?s1, ?s2) trr and O trr = obl  $\wedge$  V
trr = vll
      by auto
      thus ?thesis using trn1 trn2 ComO ComV f1 f2  $\varphi12$  g1 g2  $\gamma1$   $\gamma2$ 
com1 com2 sync s1 s2
      by (intro exI[of - CTrans trn1 trn2 # trr]) auto
      qed
    next
      case False
      hence  $\varphi12$ :  $\neg \varphi1$  trn1  $\wedge \neg \varphi2$  trn2 using sync- $\varphi1$ - $\varphi2$ [OF trn1 rs1
trn2 rs2 com1 com2 sync] by simp
      hence vl: compV (One.V tr1) (Two.V tr2) vl using vl by simp
      from ConsCons.IH(1)[OF tr1 tr2 rs1' rs2' vl obl]
      obtain trr where validFrom (?s1, ?s2) trr and O trr = obl  $\wedge$  V trr
= vl by auto
      thus ?thesis using trn1 trn2 ComO  $\varphi12$  g1 g2  $\gamma1$   $\gamma2$  com1 com2 sync
s1 s2
      by (intro exI[of - CTrans trn1 trn2 # trr]) auto
      qed
    next
      case False
      hence  $\varphi2$ :  $\varphi2$  trn2 and com2:  $\neg$  isCom2 trn2 using  $\varphi\gamma2$  isCom2- $\gamma2$ 
trn2 s2 by auto
      have False using trr2 vl s2(2)  $\varphi2$  com2 isCom2-V2[OF trn2] s2 by
(cases rule: compV-cases-V) auto
      thus ?thesis by simp
      qed
    next
      case False note  $\gamma1 = False$ 
      hence obl: compO (One.O tr1) (Two.O ?tr2) obl using obl by simp
      have  $\varphi1$ :  $\varphi1$  trn1 and com1:  $\neg$  isCom1 trn1 using  $\gamma1$   $\varphi\gamma1$  isCom1- $\gamma1$ 
trn1 s1 by auto
      show ?thesis using trr2 vl s2(2) proof(cases rule: compV-cases-V)
      case Nil hence False using  $\varphi1$  by auto
      thus ?thesis by simp
      next
      case Com hence False using com1  $\varphi1$  isCom1-isComV1[OF trn1] s1
by auto
      thus ?thesis by simp
      next
      case (Step1 vll1 vll2 vll v1) note Step1V = Step1
      hence f1: f1 trn1 = v1 and vll1 = One.V tr1 using  $\varphi1$  by auto

```

hence vll : $compV$ ($One.V$ $tr1$) ($Two.V$ $?tr2$) vll **using** $Step1V$ $com1$
by $auto$
 from $ConsCons.IH(2)[OF - tr1$ $trr2$ $rs1'$ $s2(2)$ vll $obl]$
obtain trr **where** $validFrom$ ($?s1, s2$) trr **and** O $trr = obl \wedge V$ $trr =$
 vll
by $auto$
thus $?thesis$ **using** $trn1$ $Step1V$ $f1$ $\varphi1$ $\gamma1$ $trn1$ $s1$ $isCom1-isComO1$ $com1$
by ($intro$ $exI[of - Trans1$ $s2$ $trn1$ $\#$ $trr]$) $auto$
qed
qed
qed
qed
qed

lemma $pullback-O-V$:

assumes $One.validFrom$ $istate1$ $tr1$ **and** $Two.validFrom$ $istate2$ $tr2$
and $compV$ ($One.V$ $tr1$) ($Two.V$ $tr2$) vl
and $compO$ ($One.O$ $tr1$) ($Two.O$ $tr2$) ol
shows $\exists tr.$ $validFrom$ $icstate$ $tr \wedge O$ $tr = ol \wedge V$ $tr = vl$
using $assms$ $pullback-O-V-aux$ $One.reach.Istate$ $Two.reach.Istate$ **unfolding** $ic-$
 $state-def$ **by** $auto$

end

sublocale $BD-Security-TS-Comp \subseteq K? : Abstract-BD-Security-Comp$ **where**
 $validSystemTraces1 = One.validFrom$ $istate1$ **and** $V1 = One.V$ **and** $O1 = One.O$
and $TT1 = never$ $T1$ **and** $B1 = B1$ **and**
 $validSystemTraces2 = Two.validFrom$ $istate2$ **and** $V2 = Two.V$ **and** $O2 =$
 $Two.O$
and $TT2 = never$ $T2$ **and** $B2 = B2$ **and**
 $validSystemTraces = validFrom$ $icstate$ **and** $V = V$ **and** $O = O$
and $TT = never$ T **and** $B = B$ **and**
 $comp = comp$ **and** $compO = compO$ **and** $compV = compV$
apply $standard$
subgoal **using** $validFrom$ **by** $fastforce$
subgoal **using** $compV$ **by** $fastforce$
subgoal **using** $compO$ **by** $fastforce$
subgoal **using** T **by** $fastforce$
subgoal **using** B **by** $fastforce$
subgoal **using** $pullback-O-V$ **by** $fastforce$
done

context $BD-Security-TS-Comp$ **begin**

theorem $secure1 \implies secure2 \implies secure$
using $secure1-secure2-secure$.

end

end

3 Trivial security properties

Here we formalize some cases when BD Security holds trivially.

theory *Trivial-Security*
imports *Bounded-Deducibility-Security.Abstract-BD-Security*
begin

definition $B-id :: 'value \Rightarrow 'value \Rightarrow bool$
where $B-id\ vl\ vl1 \equiv (vl1 = vl)$

context *Abstract-BD-Security*
begin

lemma *B-id-secure:*
assumes $\bigwedge tr\ vl\ vl1. B (V\ tr)\ vl1 \implies validSystemTrace\ tr \implies B-id (V\ tr)\ vl1$
shows $secure$
using *assms unfolding secure-def B-id-def by auto*

lemma *O-const-secure:*
assumes $\bigwedge tr. validSystemTrace\ tr \implies O\ tr = ol$
and $\bigwedge tr\ vl\ vl1. B (V\ tr)\ vl1 \implies validSystemTrace\ tr \implies (\exists tr1. validSystemTrace\ tr1 \wedge V\ tr1 = vl1)$
shows $secure$
unfolding *secure-def* **proof** (*intro allI impI, elim conjE*)
fix $tr\ vl\ vl1$
assume $B\ vl\ vl1$ **and** $validSystemTrace\ tr$ **and** $V\ tr = vl$
moreover then obtain $tr1$ **where** $validSystemTrace\ tr1 \wedge V\ tr1 = vl1$ **using**
assms(2) **by auto**
ultimately show $\exists tr1. validSystemTrace\ tr1 \wedge O\ tr1 = O\ tr \wedge V\ tr1 = vl1$
using *assms(1)* **by auto**
qed

definition *OV-compatible* $:: 'observations \Rightarrow 'values \Rightarrow bool$ **where**
 $OV-compatible\ obs\ vl \equiv (\exists tr. O\ tr = obs \wedge V\ tr = vl)$

definition *V-compatible* $:: 'values \Rightarrow 'values \Rightarrow bool$ **where**

V -compatible $vl\ vl1 \equiv (\forall obs. OV\text{-compatible}\ obs\ vl \longrightarrow OV\text{-compatible}\ obs\ vl1)$

definition $validObs :: 'observations \Rightarrow bool$ **where**
 $validObs\ obs \equiv (\exists tr. validSystemTrace\ tr \wedge O\ tr = obs)$

definition $validVal :: 'values \Rightarrow bool$ **where**
 $validVal\ vl \equiv (\exists tr. validSystemTrace\ tr \wedge V\ tr = vl)$

lemma $OV\text{-total-secure}$:

assumes $OV: \bigwedge obs\ vl. validObs\ obs \Longrightarrow validVal\ vl \Longrightarrow OV\text{-compatible}\ obs\ vl$
 $\Longrightarrow (\exists tr. validSystemTrace\ tr \wedge O\ tr = obs \wedge V\ tr = vl)$

and $BV: \bigwedge vl\ vl1. B\ vl\ vl1 \Longrightarrow validVal\ vl \Longrightarrow V\text{-compatible}\ vl\ vl1 \wedge validVal\ vl1$
shows $secure$

unfolding $secure\text{-def}$ **proof** (*intro allI impI, elim conjE*)

fix $tr\ vl\ vl1$

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

then have $validObs\ (O\ tr)$ **and** $validVal\ (V\ tr)$ **and** $OV\text{-compatible}\ (O\ tr)\ (V\ tr)$

unfolding $validObs\text{-def}\ validVal\text{-def}\ OV\text{-compatible}\text{-def}$ **by** $blast+$

moreover then have $validVal\ vl1$ **and** $OV\text{-compatible}\ (O\ tr)\ vl1$

using $B\ BV$ **unfolding** $V\text{-compatible}\text{-def}\ vl$ **by** $blast+$

ultimately show $\exists tr1. validSystemTrace\ tr1 \wedge O\ tr1 = O\ tr \wedge V\ tr1 = vl1$

using OV **by** $blast$

qed

lemma $unconstrained\text{-secure}$:

assumes $\bigwedge tr. validSystemTrace\ tr$

and $BV: \bigwedge vl\ vl1. B\ vl\ vl1 \Longrightarrow validVal\ vl \Longrightarrow V\text{-compatible}\ vl\ vl1 \wedge validVal\ vl1$

shows $secure$

using $assms$ **by** (*intro* $OV\text{-total-secure}$) (*auto simp*: $OV\text{-compatible}\text{-def}$)

end

end

4 Transporting BD Security

This theory proves a transport theorem for BD security: from a stronger to a weaker security model. It corresponds to Theorem 2 from [2] and to Theorem 6 (the Transport Theorem) from [7].

theory $Transporting\text{-Security}$

imports $Bounded\text{-Deducibility}\text{-Security}\text{-BD}\text{-Security}\text{-TS}$

begin

locale $Abstract\text{-BD}\text{-Security}\text{-Trans} =$

Orig: $Abstract\text{-BD}\text{-Security}\ validSystemTrace\ V\ O\ B\ TT$

+ *Prime*: $Abstract\text{-BD}\text{-Security}\ validSystemTrace'\ V'\ O'\ B'\ TT'$

for

```

    validSystemTrace :: 'traces  $\Rightarrow$  bool
  and
    V :: 'traces  $\Rightarrow$  'values
  and
    O :: 'traces  $\Rightarrow$  'observations
  and
    B :: 'values  $\Rightarrow$  'values  $\Rightarrow$  bool
  and
    TT :: 'traces  $\Rightarrow$  bool
  and
    validSystemTrace' :: 'traces'  $\Rightarrow$  bool
  and
    V' :: 'traces'  $\Rightarrow$  'values'
  and
    O' :: 'traces'  $\Rightarrow$  'observations'
  and
    B' :: 'values'  $\Rightarrow$  'values'  $\Rightarrow$  bool
  and
    TT' :: 'traces'  $\Rightarrow$  bool
+
  fixes
    translateTrace :: 'traces  $\Rightarrow$  'traces'
  and
    translateObs :: 'observations  $\Rightarrow$  'observations'
  and
    translateVal :: 'values  $\Rightarrow$  'values'
  assumes
    vST-vST': validSystemTrace tr  $\Longrightarrow$  validSystemTrace' (translateTrace tr)
  and
    vST'-vST: validSystemTrace' tr'  $\Longrightarrow$  ( $\exists$  tr. validSystemTrace tr  $\wedge$  translateTrace
tr = tr')
  and
    V'-V: validSystemTrace tr  $\Longrightarrow$  V' (translateTrace tr) = translateVal (V tr)
  and
    O'-O: validSystemTrace tr  $\Longrightarrow$  O' (translateTrace tr) = translateObs (O tr)
  and
    B'-B: B' vl' vl1'  $\Longrightarrow$  validSystemTrace tr  $\Longrightarrow$  TT tr  $\Longrightarrow$  translateVal (V tr) =
vl'
 $\Longrightarrow$  ( $\exists$  vl1. translateVal vl1 = vl1'  $\wedge$  B (V tr) vl1)
  and
    TT'-TT: TT' (translateTrace tr)  $\Longrightarrow$  validSystemTrace tr  $\Longrightarrow$  TT tr
begin

lemma translate-secure:
  assumes Orig.secure
  shows Prime.secure
  unfolding Prime.secure-def proof (intro allI impI, elim conjE)
    fix tr' vl' vl1'
    assume tr': validSystemTrace' tr' and TT': TT' tr' and B': B' vl' vl1' and vl':

```

```

V' tr' = vl'
from tr' obtain tr where tr: validSystemTrace tr translateTrace tr = tr'
  using vST'-vST by auto
  moreover have TT tr using TT' tr TT'-TT by auto
  moreover then obtain vl1 where B (V tr) vl1 and vl1: translateVal vl1 =
vl1'
  using tr B' B'-B[of vl' vl1' tr] vl' V'-V by auto
  ultimately obtain tr1 where validSystemTrace tr1 O tr1 = O tr V tr1 = vl1
  using assms unfolding Orig.secure-def by auto
  then show  $\exists tr1'. \text{validSystemTrace}' tr1' \wedge O' tr1' = O' tr' \wedge V' tr1' = vl1'$ 
  using vST-vST' O'-O V'-V tr vl1 by (intro exI[of - translateTrace tr1]) auto
qed

end

```

```

locale BD-Security-TS-Trans =
  Orig: BD-Security-TS istate validTrans srcOf tgtOf  $\varphi$  f  $\gamma$  g T B
+ Prime?: BD-Security-TS istate' validTrans' srcOf' tgtOf'  $\varphi'$  f'  $\gamma'$  g' T' B'
for istate :: 'state and validTrans :: 'trans  $\Rightarrow$  bool
and srcOf :: 'trans  $\Rightarrow$  'state and tgtOf :: 'trans  $\Rightarrow$  'state
and  $\varphi$  :: 'trans  $\Rightarrow$  bool and f :: 'trans  $\Rightarrow$  'val
and  $\gamma$  :: 'trans  $\Rightarrow$  bool and g :: 'trans  $\Rightarrow$  'obs
and T :: 'trans  $\Rightarrow$  bool and B :: 'val list  $\Rightarrow$  'val list  $\Rightarrow$  bool
and istate' :: 'state' and validTrans' :: 'trans'  $\Rightarrow$  bool
and srcOf' :: 'trans'  $\Rightarrow$  'state' and tgtOf' :: 'trans'  $\Rightarrow$  'state'
and  $\varphi'$  :: 'trans'  $\Rightarrow$  bool and f' :: 'trans'  $\Rightarrow$  'val'
and  $\gamma'$  :: 'trans'  $\Rightarrow$  bool and g' :: 'trans'  $\Rightarrow$  'obs'
and T' :: 'trans'  $\Rightarrow$  bool and B' :: 'val' list  $\Rightarrow$  'val' list  $\Rightarrow$  bool
+
fixes
  translateState :: 'state  $\Rightarrow$  'state'
and
  translateTrans :: 'trans  $\Rightarrow$  'trans'
and
  translateObs :: 'obs  $\Rightarrow$  'obs' option
and
  translateVal :: 'val  $\Rightarrow$  'val' option
assumes
  vT-vT': validTrans trn  $\Longrightarrow$  Orig.reach (srcOf trn)  $\Longrightarrow$  validTrans' (translateTrans
trn)
and
  vT'-vT: validTrans' trn'  $\Longrightarrow$  srcOf' trn' = translateState s  $\Longrightarrow$  Orig.reach s  $\Longrightarrow$ 
( $\exists$  trn. validTrans trn  $\wedge$  srcOf trn = s  $\wedge$  translateTrans trn = trn')
and
  srcOf'-srcOf: validTrans trn  $\Longrightarrow$  Orig.reach (srcOf trn)  $\Longrightarrow$  srcOf' (translateTrans
trn) = translateState (srcOf trn)
and
  tgtOf'-tgtOf: validTrans trn  $\Longrightarrow$  Orig.reach (srcOf trn)  $\Longrightarrow$  tgtOf' (translateTrans
trn) = translateState (tgtOf trn)

```

and
 $istate'-istate: istate' = translateState\ istate$
and
 $\gamma'-\gamma: validTrans\ trn \implies Orig.reach\ (srcOf\ trn) \implies \gamma'\ (translateTrans\ trn) \implies \gamma\ trn \wedge translateObs\ (g\ trn) = Some\ (g'\ (translateTrans\ trn))$
and
 $\gamma'-\gamma': validTrans\ trn \implies Orig.reach\ (srcOf\ trn) \implies \gamma\ trn \implies \gamma'\ (translateTrans\ trn) \vee translateObs\ (g\ trn) = None$
and
 $\varphi'-\varphi: validTrans\ trn \implies Orig.reach\ (srcOf\ trn) \implies \varphi'\ (translateTrans\ trn) \implies \varphi\ trn \wedge translateVal\ (f\ trn) = Some\ (f'\ (translateTrans\ trn))$
and
 $\varphi'-\varphi': validTrans\ trn \implies Orig.reach\ (srcOf\ trn) \implies \varphi\ trn \implies \varphi'\ (translateTrans\ trn) \vee translateVal\ (f\ trn) = None$
and
 $T-T': T\ trn \implies validTrans\ trn \implies Orig.reach\ (srcOf\ trn) \implies T'\ (translateTrans\ trn)$
and
 $B'-B: B'\ vl'\ vl1' \implies Orig.validFrom\ istate\ tr \implies never\ T\ tr \implies these\ (map\ translateVal\ (Orig.V\ tr)) = vl'$
 $\implies (\exists\ vl1.\ these\ (map\ translateVal\ vl1) = vl1' \wedge B\ (Orig.V\ tr)\ vl1)$
begin

definition $translateTrace :: 'trans\ list \Rightarrow 'trans'\ list$
where $translateTrace = map\ translateTrans$

definition $translateO :: 'obs\ list \Rightarrow 'obs'\ list$
where $translateO\ ol = these\ (map\ translateObs\ ol)$

definition $translateV :: 'val\ list \Rightarrow 'val'\ list$
where $translateV\ vl = these\ (map\ translateVal\ vl)$

lemma $validFrom-validFrom'$:
assumes $Orig.validFrom\ s\ tr$
and $Orig.reach\ s$
shows $Prime.validFrom\ (translateState\ s)\ (translateTrace\ tr)$
using $assms\ unfolding\ translateTrace-def$
proof ($induction\ tr\ arbitrary: s$)
case ($Cons\ trn\ tr\ s$)
then **have** $tr: Orig.validFrom\ (tgtOf\ trn)\ tr$ **and** $s': Orig.reach\ (tgtOf\ trn)$
unfolding $Orig.validFrom-Cons$ **by** ($auto\ intro: Orig.reach.Step$)
from $Cons.IH[OF\ this]\ Cons.premis$ **show** $?case$ **using** $vT-vT'$
by ($auto\ simp: Orig.validFrom-Cons\ Prime.validFrom-Cons\ srcOf'-srcOf\ tgtOf'-tgtOf$)
qed $auto$

lemma $validFrom'-validFrom$:
assumes $Prime.validFrom\ s'\ tr'$
and $s' = translateState\ s$

and $Orig.reach\ s$
obtains tr **where** $Orig.validFrom\ s\ tr$ **and** $tr' = translateTrace\ tr$
using $assms\ unfolding\ translateTrace-def$
proof (*induction* tr' *arbitrary*: $s'\ s$)
 case ($Cons\ trn'\ tr'\ s'\ s$)
 obtain trn **where** $trn: validTrans\ trn\ srcOf\ trn = s\ trn' = translateTrans\ trn$
 using $vT'-vT[of\ trn'\ s]\ Cons.premis\ unfolding\ Prime.validFrom-Cons$ **by**
 auto
 show *thesis* **proof** (*rule* $Cons.IH$)
 show $Prime.validFrom\ (tgtOf'\ trn')\ tr'$ **using** $Cons.premis\ unfolding\ Prime.validFrom-Cons$
by *auto*
 show $tgtOf'\ trn' = translateState\ (tgtOf\ trn)$ **using** $trn\ Cons.premis(4)$
tgtOf'-tgtOf **by** *auto*
 show $Orig.reach\ (tgtOf\ trn)$ **using** $trn\ Cons.premis(4)$
 by (*auto* *intro*: $Orig.reach.Step[of\ s\ trn\ tgtOf\ trn]$)
 next
 fix tr
 assume $Orig.validFrom\ (tgtOf\ trn)\ tr\ tr' = map\ translateTrans\ tr$
 then show *thesis* **using** $trn\ Cons.premis$
 by (*intro* $Cons.premis(1)[of\ trn\ \# tr]$) (*auto* *simp*: $Orig.validFrom-Cons$)
 qed
qed *auto*

lemma $V'-V$:
assumes $Orig.validFrom\ s\ tr$
and $Orig.reach\ s$
shows $Prime.V\ (translateTrace\ tr) = translateV\ (Orig.V\ tr)$
using $assms\ unfolding\ translateTrace-def\ translateV-def$
proof (*induction* tr *arbitrary*: s)
 case ($Cons\ trn\ tr\ s$)
 then have $validTrans\ trn\ srcOf\ trn = s\ Orig.validFrom\ (tgtOf\ trn)\ tr\ Orig.reach$
 ($tgtOf\ trn$)
 unfolding $Orig.validFrom-Cons$ **by** (*auto* *intro*: $Orig.reach.Step[of\ s\ trn\ tgtOf\ trn]$)
 then show *?case* **using** $\varphi'-\varphi[of\ trn]\ \varphi-\varphi'[of\ trn]\ Cons(3)\ Cons.IH$
 by (*cases* $\varphi\ trn$; *cases* $\varphi'\ (translateTrans\ trn)$) *auto*
 qed *auto*

lemma $O'-O$:
assumes $Orig.validFrom\ s\ tr$
and $Orig.reach\ s$
shows $Prime.O\ (translateTrace\ tr) = translateO\ (Orig.O\ tr)$
using $assms\ unfolding\ translateTrace-def\ translateO-def$
proof (*induction* tr *arbitrary*: s)
 case ($Cons\ trn\ tr\ s$)
 then have $validTrans\ trn\ srcOf\ trn = s\ Orig.validFrom\ (tgtOf\ trn)\ tr\ Orig.reach$
 ($tgtOf\ trn$)
 unfolding $Orig.validFrom-Cons$ **by** (*auto* *intro*: $Orig.reach.Step[of\ s\ trn\ tgtOf\ trn]$)

then show *?case* **using** $\gamma'-\gamma$ [of *trn*] $\gamma-\gamma'$ [of *trn*] *Cons(3)* *Cons.IH*
by (*cases* γ *trn*; *cases* γ' (*translateTrans trn*)) *auto*
qed *auto*

lemma *TT'-TT*:
assumes *never T'* (*translateTrace tr*)
and *Orig.validFrom s tr*
and *Orig.reach s*
shows *never T tr*
using *assms* **unfolding** *translateTrace-def*
proof (*induction tr arbitrary: s*)
case (*Cons trn tr s*)
moreover then have *never T tr* **and** *validTrans trn* **and** *srcOf trn = s*
using *Orig.reach.Step*[of *s trn tgtOf trn*] **unfolding** *Orig.validFrom-Cons* **by**
auto
ultimately show *?case* **using** *T-T'* **by** *auto*
qed *auto*

sublocale *Abstract-BD-Security-Trans*
where *validSystemTrace = Orig.validFrom istate* **and** *O = Orig.O* **and** *V = Orig.V* **and** *TT = never T*
and *validSystemTrace' = Prime.validFrom istate'* **and** *O' = Prime.O* **and** *V' = Prime.V* **and** *TT' = never T'*
and *translateTrace = translateTrace* **and** *translateObs = translateO* **and** *translateVal = translateV*
proof
fix *tr*
assume *Orig.validFrom istate tr*
then show *Prime.validFrom istate' (translateTrace tr)*
using *Orig.reach.Istate* **unfolding** *istate'-istate* **by** (*intro validFrom-validFrom'*)
next
fix *tr'*
assume *Prime.validFrom istate' tr'*
then show $\exists tr. \text{Orig.validFrom istate } tr \wedge \text{translateTrace } tr = tr'$
using *istate'-istate Orig.reach.Istate* **by** (*auto elim: validFrom'-validFrom*)
next
fix *tr*
assume *Orig.validFrom istate tr*
then show *Prime.V (translateTrace tr) = translateV (Orig.V tr)*
using *V'-V Orig.reach.Istate* **by** *blast*
next
fix *tr*
assume *Orig.validFrom istate tr*
then show *Prime.O (translateTrace tr) = translateO (Orig.O tr)*
using *O'-O Orig.reach.Istate* **by** *blast*
next
fix *vl' vl1' tr*
assume *B' vl' vl1'* **and** *Orig.validFrom istate tr* **and** *translateV (Orig.V tr) = vl'*

```

and never T tr
then show  $\exists vl1. \text{translateV } vl1 = vl1' \wedge B (\text{Orig.V } tr) vl1$ 
using B'-B unfolding translateV-def by blast
next
fix tr
assume never T' (translateTrace tr) and Orig.validFrom istate tr
then show never T tr using TT'-TT Orig.reach.Istate by blast
qed

```

```

theorem Orig.secure  $\implies$  Prime.secure using translate-secure .

```

```

end

```

```

locale BD-Security-TS-Weaken-Observations =
  Orig: BD-Security-TS where g = g for g :: 'trans  $\Rightarrow$  'obs
  + fixes translateObs :: 'obs  $\Rightarrow$  'obs' option
begin

```

```

definition  $\gamma' :: 'trans \Rightarrow bool$ 
where  $\gamma' \text{ trn} \equiv \gamma \text{ trn} \wedge \text{translateObs } (g \text{ trn}) \neq \text{None}$ 

```

```

definition  $g' :: 'trans \Rightarrow 'obs'$ 
where  $g' \text{ trn} \equiv \text{the } (\text{translateObs } (g \text{ trn}))$ 

```

```

sublocale Prime?: BD-Security-TS istate validTrans srcOf tgtOf  $\varphi$  f  $\gamma'$   $g'$  T B .

```

```

sublocale BD-Security-TS-Trans istate validTrans srcOf tgtOf  $\varphi$  f  $\gamma$  g T B
  istate validTrans srcOf tgtOf  $\varphi$  f  $\gamma'$   $g'$  T B
  id id translateObs Some
by (unfold-locales) (auto simp:  $\gamma'$ -def  $g'$ -def)

```

```

theorem Orig.secure  $\implies$  Prime.secure using translate-secure .

```

```

end

```

```

end

```

5 N-ary compositionality theorem

This theory provides the n-ary version of the compositionality theorem for BD security. It corresponds to Theorem 3 from [2] and to Theorem 7 (the System Compositionality Theorem, n-ary case) from [7].

```

theory Composing-Security-Network
imports Trivial-Security Transporting-Security Composing-Security
begin

```

Definition of n-ary system composition:

```

type-synonym ('nodeid, 'state) nstate = 'nodeid ⇒ 'state
datatype ('nodeid, 'state, 'trans) ntrans =
  LTrans ('nodeid, 'state) nstate 'nodeid 'trans
| CTrans ('nodeid, 'state) nstate 'nodeid 'trans 'nodeid 'trans
datatype ('nodeid, 'obs) nobs = LObs 'nodeid 'obs | CObs 'nodeid 'obs 'nodeid
'obs
datatype ('nodeid, 'val) nvalue = LVal 'nodeid 'val | CVal 'nodeid 'val 'nodeid
'val
datatype com = Send | Recv | Internal

locale TS-Network =
fixes
  ystate :: ('nodeid, 'state) nstate and validTrans :: 'nodeid ⇒ 'trans ⇒ bool
and
  srcOf :: 'nodeid ⇒ 'trans ⇒ 'state and tgtOf :: 'nodeid ⇒ 'trans ⇒ 'state
and
  nodes :: 'nodeid set
and
  comOf :: 'nodeid ⇒ 'trans ⇒ com
and
  tgtNodeOf :: 'nodeid ⇒ 'trans ⇒ 'nodeid
and
  sync :: 'nodeid ⇒ 'trans ⇒ 'nodeid ⇒ 'trans ⇒ bool
assumes
  finite-nodes: finite nodes
and
  isCom-tgtNodeOf:  $\bigwedge nid\ trn.$ 
     $\llbracket \text{validTrans } nid\ trn; \text{comOf } nid\ trn = \text{Send} \vee \text{comOf } nid\ trn = \text{Recv};$ 
     $\text{Transition-System.reach } (ystate\ nid)\ (\text{validTrans } nid)\ (\text{srcOf } nid)\ (\text{tgtOf } nid)$ 
     $(\text{srcOf } nid\ trn)\rrbracket$ 
     $\implies \text{tgtNodeOf } nid\ trn \neq nid$ 
begin

abbreviation isCom :: 'nodeid ⇒ 'trans ⇒ bool
where isCom nid trn  $\equiv (\text{comOf } nid\ trn = \text{Send} \vee \text{comOf } nid\ trn = \text{Recv}) \wedge$ 
  tgtNodeOf nid trn  $\in$  nodes

abbreviation lreach :: 'nodeid ⇒ 'state ⇒ bool
where lreach nid s  $\equiv \text{Transition-System.reach } (ystate\ nid)\ (\text{validTrans } nid)\ (\text{srcOf } nid)$ 
  (tgtOf nid) s

```

Two types of valid transitions in the network:

- Local transitions of network nodes, i.e. transitions that are not communicating (with another node in the network. There might be external communication transitions with the outside world. These are kept as local transitions, and turn into synchronized communication transitions when the target node joins the network during the inductive proofs later on.)

- Communication transitions between two network nodes; these are allowed if they are synchronized.

```

fun nValidTrans :: ('nodeid, 'state, 'trans) ntrans  $\Rightarrow$  bool where
  Local: nValidTrans (LTrans s nid trn) =
    (validTrans nid trn  $\wedge$  srcOf nid trn = s nid  $\wedge$  nid  $\in$  nodes  $\wedge$   $\neg$ isCom nid trn)
| Comm: nValidTrans (CTrans s nid1 trn1 nid2 trn2) =
    (validTrans nid1 trn1  $\wedge$  srcOf nid1 trn1 = s nid1  $\wedge$  comOf nid1 trn1 = Send
 $\wedge$  tgtNodeOf nid1 trn1 = nid2  $\wedge$ 
    validTrans nid2 trn2  $\wedge$  srcOf nid2 trn2 = s nid2  $\wedge$  comOf nid2 trn2 = Recv
 $\wedge$  tgtNodeOf nid2 trn2 = nid1  $\wedge$ 
    nid1  $\in$  nodes  $\wedge$  nid2  $\in$  nodes  $\wedge$  nid1  $\neq$  nid2  $\wedge$ 
    sync nid1 trn1 nid2 trn2)

```

```

fun nSrcOf :: ('nodeid, 'state, 'trans) ntrans  $\Rightarrow$  ('nodeid, 'state) nstate where
  nSrcOf (LTrans s nid trn) = s
| nSrcOf (CTrans s nid1 trn1 nid2 trn2) = s

```

```

fun nTgtOf :: ('nodeid, 'state, 'trans) ntrans  $\Rightarrow$  ('nodeid, 'state) nstate where
  nTgtOf (LTrans s nid trn) = s(nid := tgtOf nid trn)
| nTgtOf (CTrans s nid1 trn1 nid2 trn2) = s(nid1 := tgtOf nid1 trn1, nid2 :=
tgtOf nid2 trn2)

```

sublocale Transition-System istate nValidTrans nSrcOf nTgtOf .

```

fun nSrcOfTrFrom where
  nSrcOfTrFrom s [] = s
| nSrcOfTrFrom s (trn # tr) = nSrcOf trn

```

lemma nSrcOfTrFrom-nSrcOf-hd:
 $tr \neq [] \Rightarrow nSrcOfTrFrom s tr = nSrcOf (hd tr)$
by (cases tr) auto

```

fun nTgtOfTrFrom where
  nTgtOfTrFrom s [] = s
| nTgtOfTrFrom s (trn # tr) = nTgtOfTrFrom (nTgtOf trn) tr

```

lemma nTgtOfTrFrom-nTgtOf-last:
 $tr \neq [] \Rightarrow nTgtOfTrFrom s tr = nTgtOf (last tr)$
by (induction s tr rule: nTgtOfTrFrom.induct) auto

lemma reach-lreach:

assumes reach s

obtains lreach nid (s nid)

proof –

interpret Node: Transition-System istate nid validTrans nid srcOf nid tgtOf nid

.

from assms that **show** thesis

proof induction

```

    case Istate then show thesis using Node.reach.Istate by auto
next
case (Step s trn s')
  show thesis proof (rule Step.IH)
  assume Node.reach (s nid)
  then show thesis using Step.hyps Node.reach.Step[of s nid - s' nid]
  by (intro Step.prem, cases trn) (auto)
qed
qed
qed

```

Alternative characterization of valid network traces as composition of valid node traces.

```

inductive comp :: ('nodeid, 'state) nstate  $\Rightarrow$  ('nodeid, 'state, 'trans) ntrans list  $\Rightarrow$ 
bool
where
  Nil: comp s []
| Local:  $\bigwedge s$  trn s' tr nid.
   $\llbracket$ comp s tr; tgtOf nid trn = s nid; s' = s(nid := srcOf nid trn); nid  $\in$  nodes;
 $\neg$ isCom nid trn $\rrbracket$ 
 $\implies$  comp s' (LTrans s' nid trn # tr)
| Comm:  $\bigwedge s$  trn1 trn2 s' tr nid1 nid2.
   $\llbracket$ comp s tr; tgtOf nid1 trn1 = s nid1; tgtOf nid2 trn2 = s nid2;
s' = s(nid1 := srcOf nid1 trn1, nid2 := srcOf nid2 trn2);
nid1  $\in$  nodes; nid2  $\in$  nodes; nid1  $\neq$  nid2;
comOf nid1 trn1 = Send; tgtNodeOf nid1 trn1 = nid2;
comOf nid2 trn2 = Recv; tgtNodeOf nid2 trn2 = nid1;
sync nid1 trn1 nid2 trn2 $\rrbracket$ 
 $\implies$  comp s' (CTrans s' nid1 trn1 nid2 trn2 # tr)

```

```

abbreviation lValidFrom :: 'nodeid  $\Rightarrow$  'state  $\Rightarrow$  'trans list  $\Rightarrow$  bool where
lValidFrom nid  $\equiv$  Transition-System.validFrom (validTrans nid) (srcOf nid) (tgtOf
nid)

```

fun decomp **where**

```

  decomp (LTrans s nid' trn' # tr) nid = (if nid' = nid then trn' # decomp tr nid
else decomp tr nid)
| decomp (CTrans s nid1 trn1 nid2 trn2 # tr) nid = (if nid1 = nid then trn1 #
decomp tr nid else
                                                                    (if nid2 = nid then trn2 # decomp tr nid
else
                                                                    decomp tr nid))
| decomp [] nid = []

```

lemma decomp-append: decomp (tr1 @ tr2) nid = decomp tr1 nid @ decomp tr2 nid

proof (induction tr1)

```

  case (Cons trn tr1) then show ?case by (cases trn) auto
qed auto

```

```

lemma validFrom-comp: validFrom s tr  $\implies$  comp s tr
proof (induction tr arbitrary: s)
  case Nil show ?case by (intro comp.Nil)
next
  case (Cons trn tr s)
  then have IH: comp (nTgtOf trn) tr by (auto simp: validFrom-Cons)
  then show ?case using Cons.prems by (cases trn) (auto simp: validFrom-Cons)
intro: comp.intros
qed

```

```

lemma validFrom-lValidFrom:
assumes validFrom s tr
shows lValidFrom nid (s nid) (decomp tr nid)
proof –
  interpret Node: Transition-System istate nid validTrans nid srcOf nid tgtOf nid
  .
  from assms show ?thesis proof (induction tr arbitrary: s)
    case (Cons trn tr)
      have lValidFrom nid (nTgtOf trn nid) (decomp tr nid)
        using Cons.prems by (intro Cons.IH) (auto simp: validFrom-Cons)
      then show ?case using Cons.prems by (cases trn) (auto simp: validFrom-Cons)
Node.validFrom-Cons
    qed auto
qed

```

```

lemma comp-validFrom:
assumes comp s tr and  $\bigwedge$ nid. lValidFrom nid (s nid) (decomp tr nid)
shows validFrom s tr
using assms proof induction
  case (Local s trn s' tr nid)
    interpret Node: Transition-System istate nid validTrans nid srcOf nid tgtOf nid
    .
    have Node.validFrom (s' nid) (decomp (LTrans s' nid trn # tr) nid) using Local
by blast
    then have nValidTrans (LTrans s' nid trn) using Local by (auto simp: Node.validFrom-Cons)
    moreover have validFrom s tr proof (intro Local.IH)
      fix nid'
      have lValidFrom nid' (s' nid') (decomp (LTrans s' nid trn # tr) nid') using
Local(7) .
      then show lValidFrom nid' (s nid') (decomp tr nid') using Local(2,3)
        by (cases nid' = nid) (auto split: if-splits simp: Node.validFrom-Cons)
    qed
    ultimately show ?case using Local(2,3) unfolding validFrom-Cons by auto
next
  case (Comm s trn1 trn2 s' tr nid1 nid2)
    interpret Node1: Transition-System istate nid1 validTrans nid1 srcOf nid1 tgtOf
nid1 .
    interpret Node2: Transition-System istate nid2 validTrans nid2 srcOf nid2 tgtOf

```

```

nid2 .
  have Node1.validFrom (s' nid1) (decomp (CTrans s' nid1 trn1 nid2 trn2 # tr)
nid1)
  and Node2.validFrom (s' nid2) (decomp (CTrans s' nid1 trn1 nid2 trn2 # tr)
nid2) using Comm by blast+
  then have nValidTrans (CTrans s' nid1 trn1 nid2 trn2) using Comm
  by (auto simp: Node1.validFrom-Cons Node2.validFrom-Cons)
  moreover have validFrom s tr proof (intro Comm.IH)
  fix nid'
  have lValidFrom nid' (s' nid') (decomp (CTrans s' nid1 trn1 nid2 trn2 # tr)
nid') using Comm(14) .
  then show lValidFrom nid' (s nid') (decomp tr nid')
  using Comm(2,3,4) Node1.validFrom-Cons Node2.validFrom-Cons
  by (cases nid' = nid1 ∨ nid' = nid2) (auto split: if-splits)
qed
ultimately show ?case using Comm(2,3,4) unfolding validFrom-Cons by
auto
qed auto

```

lemma *validFrom-iff-comp*:
 $validFrom\ s\ tr \longleftrightarrow comp\ s\ tr \wedge (\forall\ nid.\ lValidFrom\ nid\ (s\ nid)\ (decomp\ tr\ nid))$
using *validFrom-comp validFrom-lValidFrom comp-validFrom* **by** *blast*

end

locale *Empty-TS-Network* = *TS-Network* **where** *nodes* = {}
begin

lemma *nValidTransE*: $nValidTrans\ trn \implies P$ **by** (cases *trn*) *auto*

lemma *validE*: $valid\ tr \implies P$ **by** (induction rule: *valid.induct*) (auto elim: *nValidTransE*)

lemma *validFrom-iff-Nil*: $validFrom\ s\ tr \longleftrightarrow tr = []$ **unfolding** *validFrom-def* **by** (auto elim: *validE*)

lemma *reach-istate*: $reach\ s \implies s = istate$ **by** (induction rule: *reach.induct*) (auto elim: *nValidTransE*)

end

Definition of n-ary security property composition:

locale *BD-Security-TS-Network* = *TS-Network* *istate validTrans srcOf tgtOf nodes comOf tgtNodeOf sync*

for

istate :: ('nodeid, 'state) *nstate* **and** *validTrans* :: 'nodeid \Rightarrow 'trans \Rightarrow bool

and

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

and

nodes :: 'nodeid set

and

```

    comOf :: 'nodeid ⇒ 'trans ⇒ com
and
    tgtNodeOf :: 'nodeid ⇒ 'trans ⇒ 'nodeid
and
    sync :: 'nodeid ⇒ 'trans ⇒ 'nodeid ⇒ 'trans ⇒ bool
+
fixes
    φ :: 'nodeid ⇒ 'trans ⇒ bool and f :: 'nodeid ⇒ 'trans ⇒ 'val
and
    γ :: 'nodeid ⇒ 'trans ⇒ bool and g :: 'nodeid ⇒ 'trans ⇒ 'obs
and
    T :: 'nodeid ⇒ 'trans ⇒ bool and B :: 'nodeid ⇒ 'val list ⇒ 'val list ⇒ bool
and
    comOfV :: 'nodeid ⇒ 'val ⇒ com
and
    tgtNodeOfV :: 'nodeid ⇒ 'val ⇒ 'nodeid
and
    syncV :: 'nodeid ⇒ 'val ⇒ 'nodeid ⇒ 'val ⇒ bool
and
    comOfO :: 'nodeid ⇒ 'obs ⇒ com
and
    tgtNodeOfO :: 'nodeid ⇒ 'obs ⇒ 'nodeid
and
    syncO :: 'nodeid ⇒ 'obs ⇒ 'nodeid ⇒ 'obs ⇒ bool

and
    source :: 'nodeid

assumes
    comOfV-comOf[simp]:
    ∧nid trn. [[validTrans nid trn; lreach nid (srcOf nid trn); φ nid trn]] ⇒ comOfV
    nid (f nid trn) = comOf nid trn
and
    tgtNodeOfV-tgtNodeOf[simp]:
    ∧nid trn. [[validTrans nid trn; lreach nid (srcOf nid trn); φ nid trn; comOf nid
    trn = Send ∨ comOf nid trn = Recv]]
    ⇒ tgtNodeOfV nid (f nid trn) = tgtNodeOf nid trn
and
    comOfO-comOf[simp]:
    ∧nid trn. [[validTrans nid trn; lreach nid (srcOf nid trn); γ nid trn]] ⇒ comOfO
    nid (g nid trn) = comOf nid trn
and
    tgtNodeOfO-tgtNodeOf[simp]:
    ∧nid trn. [[validTrans nid trn; lreach nid (srcOf nid trn); γ nid trn; comOf nid
    trn = Send ∨ comOf nid trn = Recv]]
    ⇒ tgtNodeOfO nid (g nid trn) = tgtNodeOf nid trn
and
    sync-syncV:
    ∧nid1 trn1 nid2 trn2.

```

$$\begin{aligned}
& \text{validTrans } nid1 \text{ trn1} \implies \text{lreach } nid1 \text{ (srcOf } nid1 \text{ trn1)} \implies \\
& \text{validTrans } nid2 \text{ trn2} \implies \text{lreach } nid2 \text{ (srcOf } nid2 \text{ trn2)} \implies \\
& \text{comOf } nid1 \text{ trn1} = \text{Send} \implies \text{tgtNodeOf } nid1 \text{ trn1} = nid2 \implies \\
& \text{comOf } nid2 \text{ trn2} = \text{Recv} \implies \text{tgtNodeOf } nid2 \text{ trn2} = nid1 \implies \\
& \varphi \text{ } nid1 \text{ trn1} \implies \varphi \text{ } nid2 \text{ trn2} \implies \\
& \text{sync } nid1 \text{ trn1 } nid2 \text{ trn2} \implies \text{syncV } nid1 \text{ (f } nid1 \text{ trn1)} \text{ } nid2 \text{ (f } nid2 \text{ trn2)}
\end{aligned}$$

and

sync-syncO:

$\bigwedge nid1 \text{ trn1 } nid2 \text{ trn2.}$

$$\begin{aligned}
& \text{validTrans } nid1 \text{ trn1} \implies \text{lreach } nid1 \text{ (srcOf } nid1 \text{ trn1)} \implies \\
& \text{validTrans } nid2 \text{ trn2} \implies \text{lreach } nid2 \text{ (srcOf } nid2 \text{ trn2)} \implies \\
& \text{comOf } nid1 \text{ trn1} = \text{Send} \implies \text{tgtNodeOf } nid1 \text{ trn1} = nid2 \implies \\
& \text{comOf } nid2 \text{ trn2} = \text{Recv} \implies \text{tgtNodeOf } nid2 \text{ trn2} = nid1 \implies \\
& \gamma \text{ } nid1 \text{ trn1} \implies \gamma \text{ } nid2 \text{ trn2} \implies \\
& \text{sync } nid1 \text{ trn1 } nid2 \text{ trn2} \implies \text{syncO } nid1 \text{ (g } nid1 \text{ trn1)} \text{ } nid2 \text{ (g } nid2 \text{ trn2)}
\end{aligned}$$

and

sync- φ 1- φ 2:

$\bigwedge nid1 \text{ trn1 } nid2 \text{ trn2.}$

$$\begin{aligned}
& \text{validTrans } nid1 \text{ trn1} \implies \text{lreach } nid1 \text{ (srcOf } nid1 \text{ trn1)} \implies \\
& \text{validTrans } nid2 \text{ trn2} \implies \text{lreach } nid2 \text{ (srcOf } nid2 \text{ trn2)} \implies \\
& \text{comOf } nid1 \text{ trn1} = \text{Send} \implies \text{tgtNodeOf } nid1 \text{ trn1} = nid2 \implies \\
& \text{comOf } nid2 \text{ trn2} = \text{Recv} \implies \text{tgtNodeOf } nid2 \text{ trn2} = nid1 \implies \\
& \text{sync } nid1 \text{ trn1 } nid2 \text{ trn2} \implies \varphi \text{ } nid1 \text{ trn1} \longleftrightarrow \varphi \text{ } nid2 \text{ trn2}
\end{aligned}$$

and

sync- φ - γ :

$\bigwedge nid1 \text{ trn1 } nid2 \text{ trn2.}$

$$\begin{aligned}
& \text{validTrans } nid1 \text{ trn1} \implies \text{lreach } nid1 \text{ (srcOf } nid1 \text{ trn1)} \implies \\
& \text{validTrans } nid2 \text{ trn2} \implies \text{lreach } nid2 \text{ (srcOf } nid2 \text{ trn2)} \implies \\
& \text{comOf } nid1 \text{ trn1} = \text{Send} \implies \text{tgtNodeOf } nid1 \text{ trn1} = nid2 \implies \\
& \text{comOf } nid2 \text{ trn2} = \text{Recv} \implies \text{tgtNodeOf } nid2 \text{ trn2} = nid1 \implies \\
& \gamma \text{ } nid1 \text{ trn1} \implies \gamma \text{ } nid2 \text{ trn2} \implies \\
& \text{syncO } nid1 \text{ (g } nid1 \text{ trn1)} \text{ } nid2 \text{ (g } nid2 \text{ trn2)} \implies \\
& (\varphi \text{ } nid1 \text{ trn1} \implies \varphi \text{ } nid2 \text{ trn2} \implies \text{syncV } nid1 \text{ (f } nid1 \text{ trn1)} \text{ } nid2 \text{ (f } nid2 \text{ trn2)}) \\
& \implies \\
& \text{sync } nid1 \text{ trn1 } nid2 \text{ trn2}
\end{aligned}$$

and

isCom- γ : $\bigwedge nid \text{ trn. } \text{validTrans } nid \text{ trn} \implies \text{lreach } nid \text{ (srcOf } nid \text{ trn)} \implies \text{comOf } nid \text{ trn} = \text{Send} \vee \text{comOf } nid \text{ trn} = \text{Recv} \implies \gamma \text{ } nid \text{ trn}$

and

φ -source: $\bigwedge nid \text{ trn. } \llbracket \text{validTrans } nid \text{ trn; lreach } nid \text{ (srcOf } nid \text{ trn); } \varphi \text{ } nid \text{ trn; } nid \neq \text{source; } nid \in \text{nodes} \rrbracket$

$\implies \text{isCom } nid \text{ trn} \wedge \text{tgtNodeOf } nid \text{ trn} = \text{source} \wedge \text{source} \in$

nodes

begin

abbreviation *isComO* $nid \text{ obs} \equiv (\text{comOfO } nid \text{ obs} = \text{Send} \vee \text{comOfO } nid \text{ obs} = \text{Recv}) \wedge \text{tgtNodeOfO } nid \text{ obs} \in \text{nodes}$

abbreviation *isComV* $nid \text{ val} \equiv (\text{comOfV } nid \text{ val} = \text{Send} \vee \text{comOfV } nid \text{ val} = \text{Recv}) \wedge \text{tgtNodeOfV } nid \text{ val} \in \text{nodes}$

```

fun nφ :: ('nodeid, 'state, 'trans) ntrans ⇒ bool where
  nφ (LTrans s nid trn) = φ nid trn
| nφ (CTrans s nid1 trn1 nid2 trn2) = (φ nid1 trn1 ∨ φ nid2 trn2)

fun nf :: ('nodeid, 'state, 'trans) ntrans ⇒ ('nodeid, 'val) nvalue where
  nf (LTrans s nid trn) = LVal nid (f nid trn)
| nf (CTrans s nid1 trn1 nid2 trn2) = CVal nid1 (f nid1 trn1) nid2 (f nid2 trn2)

fun nγ :: ('nodeid, 'state, 'trans) ntrans ⇒ bool where
  nγ (LTrans s nid trn) = γ nid trn
| nγ (CTrans s nid1 trn1 nid2 trn2) = (γ nid1 trn1 ∨ γ nid2 trn2)

fun ng :: ('nodeid, 'state, 'trans) ntrans ⇒ ('nodeid, 'obs) nobs where
  ng (LTrans s nid trn) = LObs nid (g nid trn)
| ng (CTrans s nid1 trn1 nid2 trn2) = CObs nid1 (g nid1 trn1) nid2 (g nid2 trn2)

fun nT :: ('nodeid, 'state, 'trans) ntrans ⇒ bool where
  nT (LTrans s nid trn) = T nid trn
| nT (CTrans s nid1 trn1 nid2 trn2) = (T nid1 trn1 ∨ T nid2 trn2)

fun decompV :: ('nodeid, 'val) nvalue list ⇒ 'nodeid ⇒ 'val list where
  decompV (LVal nid' v # vl) nid = (if nid' = nid then v # decompV vl nid else
  decompV vl nid)
| decompV (CVal nid1 v1 nid2 v2 # vl) nid = (if nid1 = nid then v1 # decompV
  vl nid else
  (if nid2 = nid then v2 # decompV vl nid else
  decompV vl nid))
| decompV [] nid = []

fun nValidV :: ('nodeid, 'val) nvalue ⇒ bool where
  nValidV (LVal nid v) = (nid ∈ nodes ∧ ¬isComV nid v)
| nValidV (CVal nid1 v1 nid2 v2) =
  (nid1 ∈ nodes ∧ nid2 ∈ nodes ∧ nid1 ≠ nid2 ∧ syncV nid1 v1 nid2 v2 ∧
  comOfV nid1 v1 = Send ∧ tgtNodeOfV nid1 v1 = nid2 ∧ comOfV nid2 v2 =
  Recv ∧ tgtNodeOfV nid2 v2 = nid1)

fun decompO :: ('nodeid, 'obs) nobs list ⇒ 'nodeid ⇒ 'obs list where
  decompO (LObs nid' obs # obsl) nid = (if nid' = nid then obs # decompO obsl
  nid else decompO obsl nid)
| decompO (CObs nid1 obs1 nid2 obs2 # obsl) nid = (if nid1 = nid then obs1 #
  decompO obsl nid else
  (if nid2 = nid then obs2 # decompO obsl
  nid else
  decompO obsl nid))
| decompO [] nid = []

```

definition $nB :: ('nodeid, 'val) nvalue\ list \Rightarrow ('nodeid, 'val) nvalue\ list \Rightarrow bool$
where
 $nB\ vl\ vl' \equiv (\forall\ nid \in nodes. B\ nid\ (decompV\ vl\ nid)\ (decompV\ vl'\ nid)) \wedge$
 $(list-all\ nValidV\ vl \longrightarrow list-all\ nValidV\ vl')$

fun $subDecompV :: ('nodeid, 'val) nvalue\ list \Rightarrow 'nodeid\ set \Rightarrow ('nodeid, 'val) nvalue\ list$ **where**
 $subDecompV\ (LVal\ nid'\ v\ \#\ vl)\ nds =$
 $(if\ nid' \in nds\ then\ LVal\ nid'\ v\ \#\ subDecompV\ vl\ nds\ else\ subDecompV\ vl\ nds)$
 $| subDecompV\ (CVal\ nid1\ v1\ nid2\ v2\ \#\ vl)\ nds =$
 $(if\ nid1 \in nds \wedge nid2 \in nds\ then\ CVal\ nid1\ v1\ nid2\ v2\ \#\ subDecompV\ vl\ nds$
 $else$
 $(if\ nid1 \in nds\ then\ LVal\ nid1\ v1\ \#\ subDecompV\ vl\ nds\ else$
 $(if\ nid2 \in nds\ then\ LVal\ nid2\ v2\ \#\ subDecompV\ vl\ nds\ else$
 $subDecompV\ vl\ nds)))$
 $| subDecompV\ []\ nds = []$

lemma $decompV-subDecompV[simp]: nid \in nds \Longrightarrow decompV\ (subDecompV\ vl\ nds)\ nid = decompV\ vl\ nid$

proof $(induction\ vl)$

case $(Cons\ v\ vl)$ **then show** $?case\ by\ (cases\ v)\ (auto\ split:\ if-splits)$

qed $auto$

sublocale $BD-Security-TS\ istate\ nValidTrans\ nSrcOf\ nTgtOf\ n\varphi\ n\psi\ n\gamma\ n\eta\ nT\ nB$
 $.$

abbreviation $lV :: 'nodeid \Rightarrow 'trans\ list \Rightarrow 'val\ list$ **where**
 $lV\ nid \equiv BD-Security-TS.V\ (\varphi\ nid)\ (f\ nid)$

abbreviation $lO :: 'nodeid \Rightarrow 'trans\ list \Rightarrow 'obs\ list$ **where**
 $lO\ nid \equiv BD-Security-TS.O\ (\gamma\ nid)\ (g\ nid)$

abbreviation $lTT :: 'nodeid \Rightarrow 'trans\ list \Rightarrow bool$ **where**
 $lTT\ nid \equiv never\ (T\ nid)$

abbreviation $lsecure :: 'nodeid \Rightarrow bool$ **where**
 $lsecure\ nid \equiv Abstract-BD-Security.secure\ (lValidFrom\ nid\ (istate\ nid))\ (lV\ nid)$
 $(lO\ nid)\ (B\ nid)\ (lTT\ nid)$

lemma $decompV-decomp:$

assumes $validFrom\ s\ tr$

and $reach\ s$

shows $decompV\ (V\ tr)\ nid = lV\ nid\ (decomp\ tr\ nid)$

proof $-$

```

interpret Node: BD-Security-TS istate nid validTrans nid srcOf nid tgtOf nid
       $\varphi$  nid f nid  $\gamma$  nid g nid T nid B nid .
from assms show ?thesis proof (induction tr arbitrary: s)
  case (Cons trn tr s)
    then have tr: decompV (V tr) nid = Node.V (decomp tr nid)
      by (intro Cons.IH[of nTgtOf trn]) (auto intro: reach.Step)
    show ?case proof (cases trn)
      case (LTrans s' nid' trn') with Cons.prems tr show ?thesis by (cases n $\varphi$ 
trn) auto
    next
      case (CTrans s' nid1 trn1 nid2 trn2)
        then have lreach nid1 (s' nid1) and lreach nid2 (s' nid2)
          using Cons.prems by (auto elim: reach-lreach)
        then have  $\varphi$  nid1 trn1  $\longleftrightarrow$   $\varphi$  nid2 trn2
          using Cons.prems CTrans by (intro sync- $\varphi$ 1- $\varphi$ 2) auto
        then show ?thesis using Cons.prems CTrans tr Node.V-Cons-unfold by
(cases n $\varphi$  trn) auto
      qed
    qed auto
  qed

```

lemma *decompO-decomp*:

assumes validFrom s tr

and reach s

shows decompO (O tr) nid = lO nid (decomp tr nid)

proof –

```

interpret Node: BD-Security-TS istate nid validTrans nid srcOf nid tgtOf nid
       $\varphi$  nid f nid  $\gamma$  nid g nid T nid B nid .
from assms show ?thesis proof (induction tr arbitrary: s)
  case (Cons trn tr s)
    then have tr: decompO (O tr) nid = Node.O (decomp tr nid)
      by (intro Cons.IH[of nTgtOf trn]) (auto intro: reach.Step)
    show ?case proof (cases trn)
      case (LTrans s' nid' trn') with Cons.prems tr show ?thesis by (cases n $\gamma$ 
trn) auto
    next
      case (CTrans s' nid1 trn1 nid2 trn2)
        then have lreach nid1 (s' nid1) and lreach nid2 (s' nid2)
          using Cons.prems by (auto elim: reach-lreach)
        then have  $\gamma$  nid1 trn1 and  $\gamma$  nid2 trn2
          using Cons.prems CTrans by (auto intro: isCom- $\gamma$ )
        then show ?thesis using Cons.prems CTrans tr Node.O-Cons-unfold by
(cases n $\gamma$  trn) auto
      qed
    qed auto
  qed

```

lemma *nTT-TT*: never nT tr \implies never (T nid) (decomp tr nid)

proof (induction tr)

case (*Cons trn tr*) **then show** *?case* **by** (*cases trn*) *auto*
qed *auto*

lemma *validFrom-nValidV*:

assumes *validFrom s tr*

and *reach s*

shows *list-all nValidV (V tr)*

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

case (*Cons trn tr s*)

have *tr: list-all nValidV (V tr)* **using** *Cons.IH[of nTgtOf trn]* *Cons.prem*s
by (*auto intro: reach.Step*)

then show *?case* **proof** (*cases trn*)

case (*LTrans s' nid' trn'*)

moreover then have *lreach nid' (s' nid')* **using** *Cons.prem*s **by** (*auto elim: reach-lreach*)

ultimately show *?thesis* **using** *Cons.prem*s *tr* **by** (*cases nφ trn*) *auto*

next

case (*CTrans s' nid1 trn1 nid2 trn2*)

moreover then have *lreach nid1 (s' nid1)* **and** *lreach nid2 (s' nid2)*

using *Cons.prem*s **by** (*auto elim: reach-lreach*)

moreover then have $\varphi \text{ nid1 trn1} \longleftrightarrow \varphi \text{ nid2 trn2}$

using *Cons.prem*s *CTrans* **by** (*intro sync-φ1-φ2*) *auto*

ultimately show *?thesis* **using** *Cons.prem*s *tr* **by** (*cases nφ trn*) (*auto intro: sync-sync V*)

qed

qed *auto*

end

An empty network is trivially secure. This is useful as a base case in proofs.

locale *BD-Security-Empty-TS-Network* = *BD-Security-TS-Network* **where** *nodes*

= {}

begin

sublocale *Empty-TS-Network* ..

lemma *nValidVE*: *nValidV v* \implies *P* **by** (*cases v*) *auto*

lemma *list-all-nValidV-Nil*: *list-all nValidV vl* \implies *vl = []* **by** (*cases vl*) (*auto elim: nValidVE*)

lemma *trivially-secure*: *secure*

by (*intro B-id-secure*) (*auto iff: validFrom-iff-Nil simp: nB-def B-id-def elim: list-all-nValidV-Nil*)

end

Another useful base case: a singleton network with just the secret source node.

locale *BD-Security-Singleton-Source-Network* = *BD-Security-TS-Network* **where** *nodes* = {*source*}

```

begin

sublocale Node: BD-Security-TS istate source validTrans source srcOf source tgtOf
source
       $\varphi$  source  $f$  source  $\gamma$  source  $g$  source  $T$  source  $B$  source .

lemma [simp]: decompV (map (LVal source) vl) source = vl
by (induction vl) auto

lemma [simp]: list-all nValidV vl'  $\implies$  map (LVal source) (decompV vl' source) =
vl'
proof (induction vl')
  case (Cons v vl') then show ?case by (cases v) auto
qed auto

lemma Node-validFrom-nValidV:
  Node.validFrom s tr  $\implies$  Node.reach s  $\implies$  list-all nValidV (map (LVal source)
(Node.V tr))
proof (induction tr arbitrary: s)
  case (Cons trn tr)
    then have Node.reach (tgtOf source trn) using Node.reach.Step[of s trn tgtOf
source trn] by auto
    then show ?case using Cons.premis Cons.IH[of tgtOf source trn]
using isCom-tgtNodeOf by (cases  $\varphi$  source trn) auto
qed auto

sublocale Trans?: BD-Security-TS-Trans
  where istate = istate source and validTrans = validTrans source and srcOf =
srcOf source and tgtOf = tgtOf source
  and  $\varphi$  =  $\varphi$  source and  $f$  =  $f$  source and  $\gamma$  =  $\gamma$  source and  $g$  =  $g$  source and  $T$ 
=  $T$  source and  $B$  =  $B$  source
  and istate' = istate and validTrans' = nValidTrans and srcOf' = nSrcOf and
tgtOf' = nTgtOf
  and  $\varphi'$  =  $n\varphi$  and  $f'$  =  $nf$  and  $\gamma'$  =  $n\gamma$  and  $g'$  =  $ng$  and  $T'$  =  $nT$  and  $B'$  =
 $nB$ 
  and translateState =  $\lambda s$ . istate(source := s)
  and translateTrans =  $\lambda trn$ . LTrans (istate(source := srcOf source trn)) source
trn
  and translateObs =  $\lambda obs$ . Some (LObs source obs)
  and translateVal = Some o LVal source
using isCom-tgtNodeOf
proof (unfold-locales, goal-cases)
  case (2 trn' s) then show ?case by (cases trn') auto next
  case (11 vl' vl1' tr)
    then show ?case using Node.reach.Istate
    by (intro exI[of - decompV vl1' source]) (auto simp: nB-def intro: Node-validFrom-nValidV)
qed auto

end

```

Setup for changing the set of nodes in a network, e.g. adding a new one. We re-check unique secret polarization, while the other assumptions about the observation and secret infrastructure are inherited from the original setup.

```

locale BD-Security-TS-Network-Change-Nodes = Orig: BD-Security-TS-Network
+
fixes nodes'
assumes finite-nodes': finite nodes'
and  $\varphi$ -source':
   $\bigwedge \text{nid trn. } \llbracket \text{validTrans nid trn; } \text{Orig.lreach nid (srcOf nid trn); } \varphi \text{ nid trn; nid} \neq \text{source; nid} \in \text{nodes}' \rrbracket$ 
   $\implies \text{Orig.isCom nid trn} \wedge \text{tgtNodeOf nid trn} = \text{source} \wedge \text{source} \in \text{nodes}'$ 
begin

```

```

sublocale BD-Security-TS-Network where nodes = nodes'
proof (unfold-locales, goal-cases)
  case 1 show ?case using finite-nodes' . next
  case 2 then show ?case using Orig.isCom-tgtNodeOf by auto next
  case 3 then show ?case by auto next
  case 4 then show ?case by auto next
  case 5 then show ?case by auto next
  case 6 then show ?case by auto next
  case 7 then show ?case using Orig.sync-syncV by auto next
  case 8 then show ?case using Orig.sync-syncO by auto next
  case 9 then show ?case using Orig.sync- $\varphi$ 1- $\varphi$ 2 by auto next
  case 10 then show ?case using Orig.sync- $\varphi$ - $\gamma$  by auto next
  case 11 then show ?case using Orig.isCom- $\gamma$  by auto next
  case 12 then show ?case using  $\varphi$ -source' by auto
qed
end

```

Adding a new node to a network that is not the secret source:

```

locale BD-Security-TS-Network-New-Node-NoSource = Sub: BD-Security-TS-Network
where istate = istate and nodes = nodes and f = f and g = g
for istate :: 'nodeid  $\Rightarrow$  'state and nodes :: 'nodeid set
and f :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'val and g :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'obs

```

```

+
fixes NID :: 'nodeid
assumes new-node: NID  $\notin$  nodes
and no-source: NID  $\neq$  source
and  $\varphi$ -NID-source:
   $\bigwedge \text{trn. } \llbracket \text{validTrans NID trn; Sub.lreach NID (srcOf NID trn); } \varphi \text{ NID trn} \rrbracket$ 
   $\implies \text{Sub.isCom NID trn} \wedge \text{tgtNodeOf NID trn} = \text{source} \wedge \text{source} \in \text{nodes}$ 
begin

```

```

sublocale Node: BD-Security-TS istate NID validTrans NID srcOf NID tgtOf NID
   $\varphi$  NID f NID  $\gamma$  NID g NID T NID B NID .

```

sublocale *BD-Security-TS-Network-Change-Nodes* **where** $nodes' = insert\ NID\ nodes$

using φ -*NID-source* *Sub.* φ -*source* *Sub.finite-nodes*
by (*unfold-locales*) *auto*

fun *isCom1* :: ('*nodeid*, '*state*, '*trans*) *ntrans* \Rightarrow *bool* **where**
isCom1 (*LTrans s nid trn*) = ($nid \in nodes \wedge isCom\ nid\ trn \wedge tgtNodeOf\ nid\ trn = NID$)
| *isCom1* - = *False*

definition *isCom2* *trn* = ($\exists\ nid. nid \in nodes \wedge isCom\ NID\ trn \wedge tgtNodeOf\ NID\ trn = nid$)

fun *Sync* :: ('*nodeid*, '*state*, '*trans*) *ntrans* \Rightarrow '*trans* \Rightarrow *bool* **where**
Sync (*LTrans s nid trn*) *trn'* = ($tgtNodeOf\ nid\ trn = NID \wedge tgtNodeOf\ NID\ trn' = nid \wedge$
 $((sync\ nid\ trn\ NID\ trn' \wedge comOf\ nid\ trn = Send \wedge comOf\ NID\ trn' = Recv)$
 $\vee (sync\ NID\ trn'\ nid\ trn \wedge comOf\ NID\ trn' = Send \wedge comOf\ nid\ trn = Recv)))$
| *Sync* - - = *False*

fun *isComV1* :: ('*nodeid*, '*val*) *nvalue* \Rightarrow *bool* **where**
isComV1 (*LVal nid v*) = ($nid \in nodes \wedge isComV\ nid\ v \wedge tgtNodeOfV\ nid\ v = NID$)
| *isComV1* - = *False*

definition *isComV2* *v* = ($\exists\ nid. nid \in nodes \wedge isComV\ NID\ v \wedge tgtNodeOfV\ NID\ v = nid$)

fun *SyncV* :: ('*nodeid*, '*val*) *nvalue* \Rightarrow '*val* \Rightarrow *bool* **where**
SyncV (*LVal nid v1*) *v2* = ($tgtNodeOfV\ nid\ v1 = NID \wedge tgtNodeOfV\ NID\ v2 = nid \wedge$
 $((syncV\ nid\ v1\ NID\ v2 \wedge comOfV\ nid\ v1 = Send \wedge comOfV\ NID\ v2 = Recv)$
 $\vee (syncV\ NID\ v2\ nid\ v1 \wedge comOfV\ NID\ v2 = Send \wedge comOfV\ nid\ v1 = Recv)))$
| *SyncV* - - = *False*

fun *CmpV* :: ('*nodeid*, '*val*) *nvalue* \Rightarrow '*val* \Rightarrow ('*nodeid*, '*val*) *nvalue* **where**
CmpV (*LVal nid v1*) *v2* = (*if* $comOfV\ nid\ v1 = Send$ *then* *CVal* *nid v1 NID v2* *else* *CVal* *NID v2 nid v1*)
| *CmpV* *cv v2* = *cv*

fun *isComO1* :: ('*nodeid*, '*obs*) *nobs* \Rightarrow *bool* **where**
isComO1 (*LObs nid obs*) = ($nid \in nodes \wedge isComO\ nid\ obs \wedge tgtNodeOfO\ nid\ obs = NID$)
| *isComO1* - = *False*

definition $isComO2\ obs = (\exists\ nid.\ nid \in nodes \wedge isComO\ NID\ obs \wedge tgtNodeOfO\ NID\ obs = nid)$

fun $SyncO :: ('nodeid, 'obs)\ nobs \Rightarrow 'obs \Rightarrow bool$ **where**
 $SyncO\ (LObs\ nid\ obs1)\ obs2 = (tgtNodeOfO\ nid\ obs1 = NID \wedge tgtNodeOfO\ NID\ obs2 = nid \wedge$
 $((syncO\ nid\ obs1\ NID\ obs2 \wedge comOfO\ nid\ obs1 = Send$
 $\wedge comOfO\ NID\ obs2 = Recv)$
 $\vee (syncO\ NID\ obs2\ nid\ obs1 \wedge comOfO\ NID\ obs2 =$
 $Send \wedge comOfO\ nid\ obs1 = Recv)))$
 $| SyncO\ - - = False$

We prove security using the binary composition theorem, composing the existing network with the new node.

sublocale $Comp: BD\text{-}Security\text{-}TS\text{-}Comp\ istate\ Sub.nValidTrans\ Sub.nSrcOf\ Sub.nTgtOf\ Sub.n\varphi\ Sub.nf\ Sub.n\gamma\ Sub.ng\ Sub.nT\ Sub.nB$
 $istate\ NID\ validTrans\ NID\ srcOf\ NID\ tgtOf\ NID\ \varphi\ NID\ f\ NID\ \gamma\ NID\ g\ NID\ T$
 $NID\ B\ NID$
 $isCom1\ isCom2\ Sync\ isComV1\ isComV2\ SyncV\ isComO1\ isComO2\ SyncO$

proof

fix $trn1$

assume $trn1: Sub.nValidTrans\ trn1\ Sub.reach\ (Sub.nSrcOf\ trn1)\ Sub.n\varphi\ trn1$

then show $isCom1\ trn1 = isComV1\ (Sub.nf\ trn1)$

proof $(cases\ trn1)$

case $(LTrans\ s\ nid\ trn)$

with $trn1$ **have** $lreach\ nid\ (srcOf\ nid\ trn)$ **by** $(auto\ elim!: Sub.reach\ lreach)$

with $trn1$ **show** $?thesis$ **using** $LTrans$ **by** $auto$

qed $auto$

next

fix $trn1$

assume $trn1: Sub.nValidTrans\ trn1\ Sub.reach\ (Sub.nSrcOf\ trn1)\ Sub.n\gamma\ trn1$

then show $isCom1\ trn1 = isComO1\ (Sub.ng\ trn1)$

proof $(cases\ trn1)$

case $(LTrans\ s\ nid\ trn)$

with $trn1$ **have** $lreach\ nid\ (srcOf\ nid\ trn)$ **by** $(auto\ elim!: Sub.reach\ lreach)$

with $trn1$ **show** $?thesis$ **using** $LTrans$ **by** $auto$

qed $auto$

next

fix $trn2$

assume $trn2: validTrans\ NID\ trn2\ Node.reach\ (srcOf\ NID\ trn2)\ \varphi\ NID\ trn2$

then show $isCom2\ trn2 = isComV2\ (f\ NID\ trn2)$

unfolding $isCom2\text{-}def\ isComV2\text{-}def$ **by** $auto$

next

fix $trn2$

assume $trn2: validTrans\ NID\ trn2\ Node.reach\ (srcOf\ NID\ trn2)\ \gamma\ NID\ trn2$

then show $isCom2\ trn2 = isComO2\ (g\ NID\ trn2)$

unfolding $isCom2\text{-}def\ isComO2\text{-}def$ **by** $auto$

next

fix $trn1\ trn2$

```

    assume trn12: Sub.nValidTrans trn1 Sub.reach (Sub.nSrcOf trn1) validTrans
NID trn2
    Node.reach (srcOf NID trn2) isCom1 trn1 isCom2 trn2 Sub.nφ trn1 φ NID
trn2 Sync trn1 trn2
    then show SyncV (Sub.nf trn1) (f NID trn2) proof (cases trn1)
      case (LTrans s nid trn)
        with trn12 have lreach nid (srcOf nid trn) by (auto elim: Sub.reach-lreach)
        with trn12 show ?thesis using LTrans by (auto intro: Sub.sync-syncV)
      qed auto
    next
    fix trn1 trn2
    assume trn12: Sub.nValidTrans trn1 Sub.reach (Sub.nSrcOf trn1) validTrans
NID trn2
    Node.reach (srcOf NID trn2) isCom1 trn1 isCom2 trn2 Sub.nγ trn1 γ NID
trn2 Sync trn1 trn2
    then show SyncO (Sub.ng trn1) (g NID trn2) proof (cases trn1)
      case (LTrans s nid trn)
        with trn12 have lreach nid (srcOf nid trn) by (auto elim: Sub.reach-lreach)
        with trn12 show ?thesis using LTrans by (auto intro: Sub.sync-syncO)
      qed auto
    next
    fix trn1 trn2
    assume trn12: Sub.nValidTrans trn1 Sub.reach (Sub.nSrcOf trn1) validTrans
NID trn2
    Node.reach (srcOf NID trn2) isCom1 trn1 isCom2 trn2 Sync trn1 trn2
    then show Sub.nφ trn1 = φ NID trn2 proof (cases trn1)
      case (LTrans s nid trn)
        with trn12 show ?thesis using Sub.sync-φ1-φ2[of nid trn NID trn2] Sub.sync-φ1-φ2[of
NID trn2 nid trn]
        by (auto elim: Sub.reach-lreach)
      qed auto
    next
    fix trn1 trn2
    assume trn12: Sub.nValidTrans trn1 Sub.reach (Sub.nSrcOf trn1) validTrans
NID trn2
    Node.reach (srcOf NID trn2) isCom1 trn1 isCom2 trn2
    Sub.nγ trn1 γ NID trn2 SyncO (Sub.ng trn1) (g NID trn2)
    Sub.nφ trn1  $\implies$  φ NID trn2  $\implies$  SyncV (Sub.nf trn1) (f NID trn2)
    then show Sync trn1 trn2 proof (cases trn1)
      case (LTrans s nid trn)
        with trn12 have lreach nid (srcOf nid trn) by (auto elim: Sub.reach-lreach)
        with trn12 show ?thesis using LTrans by (auto intro: Sub.sync-φ-γ)
      qed auto
    next
    fix trn1
    assume trn1: Sub.nValidTrans trn1 Sub.reach (Sub.nSrcOf trn1) isCom1 trn1
    then show Sub.nγ trn1 proof (cases trn1)
      case (LTrans s nid trn)
        with trn1 show ?thesis using Sub.isCom-γ[of nid trn] by (auto elim:

```

```

Sub.reach-lreach)
qed auto
next
fix trn2
assume validTrans NID trn2 Node.reach (srcOf NID trn2) isCom2 trn2
then show  $\gamma$  NID trn2 unfolding isCom2-def by (auto intro: Sub.isCom- $\gamma$ )
next
fix trn2
assume validTrans NID trn2 Node.reach (srcOf NID trn2)  $\varphi$  NID trn2
then show isCom2 trn2 using  $\varphi$ -NID-source unfolding isCom2-def by auto
qed auto

```

We then translate the canonical security property obtained from the binary compositionality result back to the original observation and secret infrastructure using the transport theorem.

```

fun translateState :: (('nodeid  $\Rightarrow$  'state)  $\times$  'state)  $\Rightarrow$  ('nodeid  $\Rightarrow$  'state) where
  translateState (sSub, sNode) = (sSub(NID := sNode))

fun translateTrans :: (('nodeid  $\Rightarrow$  'state, ('nodeid, 'state, 'trans) ntrans, 'state,
'trans) ctrans  $\Rightarrow$  ('nodeid, 'state, 'trans) ntrans) where
  translateTrans (Trans1 sNode (LTrans s nid trn)) = LTrans (s(NID := sNode))
  nid trn
| translateTrans (Trans1 sNode (CTrans s nid1 trn1 nid2 trn2)) = CTrans (s(NID
:= sNode)) nid1 trn1 nid2 trn2
| translateTrans (Trans2 sSub trn) = LTrans (sSub(NID := srcOf NID trn)) NID
trn
| translateTrans (ctrans.CTrans (LTrans s nid trn) trnNode) =
  (if comOf nid trn = Send
   then CTrans (s(NID := srcOf NID trnNode)) nid trn NID trnNode
   else CTrans (s(NID := srcOf NID trnNode)) NID trnNode nid trn)
| translateTrans - = undefined

fun translateObs :: (('nodeid, 'obs) nob, 'obs) cobs  $\Rightarrow$  ('nodeid, 'obs) nob) where
  translateObs (Obs1 obs) = obs
| translateObs (Obs2 obs) = (LObs NID obs)
| translateObs (cobs.CObs (LObs nid1 obs1) obs2) =
  (if comOfO nid1 obs1 = Send then CObs nid1 obs1 NID obs2 else CObs NID
obs2 nid1 obs1)
| translateObs - = undefined

fun translateVal :: (('nodeid, 'val) nvalue, 'val) cvalue  $\Rightarrow$  ('nodeid, 'val) nvalue
where
  translateVal (Value1 v) = v
| translateVal (Value2 v) = (LVal NID v)
| translateVal (cvalue.CValue (LVal nid1 v1) v2) =
  (if comOfV nid1 v1 = Send then CVal nid1 v1 NID v2 else CVal NID v2 nid1
v1)
| translateVal - = undefined

```

```

fun invTranslateVal :: ('nodeid, 'val) nvalue  $\Rightarrow$  (('nodeid, 'val) nvalue, 'val) cvalue
where
  invTranslateVal (LVal nid v) = (if nid = NID then Value2 v else Value1 (LVal
  nid v))
| invTranslateVal (CVal nid1 v1 nid2 v2) =
  (if nid1  $\in$  nodes  $\wedge$  nid2  $\in$  nodes then Value1 (CVal nid1 v1 nid2 v2)
  else (if nid1 = NID then CValue (LVal nid2 v2) v1
  else CValue (LVal nid1 v1) v2))

```

```

lemma translateVal-invTranslateVal[simp]: nValidV v  $\Longrightarrow$  (translateVal (invTranslateVal
v)) = v
by (elim nValidV.elims) auto

```

```

lemma map-translateVal-invTranslateVal[simp]:
  list-all nValidV vl  $\Longrightarrow$  map (translateVal o invTranslateVal) vl = vl
by (induction vl) auto

```

```

fun compValidV :: (('nodeid, 'val) nvalue, 'val) cvalue  $\Rightarrow$  bool where
  compValidV (Value1 (LVal nid v)) = (Sub.nValidV (LVal nid v)  $\wedge$  (isComV nid
v  $\longrightarrow$  tgtNodeOfV nid v  $\neq$  NID))
| compValidV (Value1 (CVal nid1 v1 nid2 v2)) = Sub.nValidV (CVal nid1 v1 nid2
v2)
| compValidV (Value2 v2) = nValidV (LVal NID v2)
| compValidV (CValue (CVal nid1 v1 nid2 v2) v) = False
| compValidV (CValue (LVal nid1 v1) v2) = (nValidV (CVal nid1 v1 NID v2)  $\vee$ 
nValidV (CVal NID v2 nid1 v1))

```

```

lemma nValidV-compValidV: nValidV v  $\Longrightarrow$  compValidV (invTranslateVal v)
by (cases v) auto

```

```

lemma list-all-nValidV-compValidV: list-all nValidV vl  $\Longrightarrow$  list-all compValidV
(map invTranslateVal vl)
by (induction vl) (auto intro: nValidV-compValidV)

```

```

lemma compValidV-nValidV: compValidV v  $\Longrightarrow$  nValidV (translateVal v)
by (cases v rule: compValidV.cases) auto

```

```

lemma list-all-compValidV-nValidV: list-all compValidV vl  $\Longrightarrow$  list-all nValidV
(map translateVal vl)
by (induction vl) (auto intro: compValidV-nValidV)

```

```

lemma nValidV-subDecompV: list-all nValidV vl  $\Longrightarrow$  list-all Sub.nValidV (subDecompV
vl nodes)

```

```

proof (induction vl)
  case (Cons v vl) then show ?case by (cases v) auto
qed auto

```

```

lemma validTrans-compValidV:
assumes Comp.validTrans trn and Comp.reach (Comp.srcOf trn) and Comp. $\varphi$ 

```

```

trn
shows compValidV (Comp.f trn)
proof (cases trn)
  case (Trans1 sNode trnSub)
    show ?thesis proof (cases trnSub)
      case (LTrans s nid1 trn1)
        then have lreach nid1 (s nid1)
          using Trans1 assms Comp.reach-reach12[of Comp.srcOf trn]
          by (auto elim!: Sub.reach-lreach)
        then show ?thesis using LTrans Trans1 assms by auto
      next
        case (CTrans s nid1 trn1 nid2 trn2)
          then have lreach nid1 (s nid1) and lreach nid2 (s nid2)
            using Trans1 assms Comp.reach-reach12[of Comp.srcOf trn]
            by (auto elim!: Sub.reach-lreach)
          then show ?thesis using CTrans Trans1 assms
            using sync-syncV[of nid1 trn1 nid2 trn2] sync-φ1-φ2[of nid1 trn1 nid2
trn2] by auto
          qed
      next
        case (Trans2 sSub trnNode)
          then have lreach NID (srcOf NID trnNode) using assms Comp.reach-reach12[of
Comp.srcOf trn] by auto
          with assms Trans2 show ?thesis using φ-NID-source by (auto simp: is-
Com2-def)
          next
            case (CTrans trnSub trnNode)
              then obtain sSub nid1 trn1 where trnSub = LTrans sSub nid1 trn1 using
assms
              by (cases trnSub) auto
              moreover then have lreach nid1 (sSub nid1) and lreach NID (srcOf NID
trnNode)
                using assms Comp.reach-reach12[of Comp.srcOf trn] CTrans by (auto elim!:
Sub.reach-lreach)
              ultimately show ?thesis using assms CTrans
                using sync-syncV[of nid1 trn1 NID trnNode] sync-φ1-φ2[of nid1 trn1 NID
trnNode]
                using sync-syncV[of NID trnNode nid1 trn1] sync-φ1-φ2[of NID trnNode
nid1 trn1]
                by (cases comOf NID trnNode = Send) auto
              qed
            qed

lemma validFrom-compValidV: Comp.reach s  $\implies$  Comp.validFrom s tr  $\implies$  list-all
compValidV (Comp.V tr)
proof (induction tr arbitrary: s)
  case (Cons trn tr)
    then have Comp.reach (Comp.tgtOf trn) using Comp.reach.Step[of s trn
Comp.tgtOf trn] by auto
    then show ?case using Cons.premis Cons.IH[of Comp.tgtOf trn] validTrans-compValidV

```

by (cases *Comp.φ trn*) auto
qed auto

lemma *validFrom-istate-compValidV*: *Comp.validFrom Comp.icstate tr* \implies *list-all compValidV (Comp.V tr)*
using *validFrom-compValidV Comp.reach.Istate* **by** *blast*

lemma *compV-decompV*:
assumes *list-all compValidV vl*
shows *Comp.compV vl1 vl2 vl*
 \longleftrightarrow *vl1 = subDecompV (map translateVal vl) nodes* \wedge *vl2 = decompV (map translateVal vl) NID*

proof

assume *Comp.compV vl1 vl2 vl*

then show *vl1 = subDecompV (map translateVal vl) nodes* \wedge *vl2 = decompV (map translateVal vl) NID*

using *assms new-node*

proof (*induction rule: Comp.compV.induct*)

case (*Step1 vl1 vl2 vl v1*) then show ?case **by** (*cases v1*) auto **next**

case (*Com vl1 vl2 vl v1 v2*) then show ?case **by** (*cases v1*) auto

qed auto

next

assume *vl1 = subDecompV (map translateVal vl) nodes* \wedge *vl2 = decompV (map translateVal vl) NID*

moreover have *Comp.compV (subDecompV (map translateVal vl) nodes) (decompV (map translateVal vl) NID) vl*

using *assms new-node*

proof (*induction vl*)

case (*Cons v vl*)

then show ?case **proof** (*cases v*)

case (*Value1 v1*) **with** *Cons* **show** ?thesis **by** (*cases v1*) auto

next

case (*Value2 v2*)

then have \neg *isComV2 v2* **using** *Cons unfolding isComV2-def* **by** auto

then show ?thesis **using** *Cons Value2* **by** auto

next

case (*CValue cv v2*)

then show ?thesis **using** *Cons.prem* **proof** (*cases cv*)

case (*LVal nid1 v1*)

then have *isComV2 v2* **using** *LVal CValue Cons unfolding isComV2-def*

by auto

then have *Comp.compV (LVal nid1 v1 # subDecompV (map translateVal vl) nodes)*

(*v2 # decompV (map translateVal vl) NID*)

(*CValue (LVal nid1 v1) v2 # vl*)

using *LVal CValue Cons* **by** (*intro Comp.compV.Com*) auto

then show ?thesis **using** *LVal CValue Cons* **by** auto

qed auto

qed

```

qed auto
ultimately show Comp.compV vl1 vl2 vl by auto
qed

```

```

sublocale Trans?: BD-Security-TS-Trans Comp.icstate Comp.validTrans Comp.srcOf
Comp.tgtOf

```

```

Comp.φ Comp.f Comp.γ Comp.g Comp.T Comp.B
istate nValidTrans nSrcOf nTgtOf nφ nf nγ ng nT nB
translateState translateTrans Some o translateObs Some o translateVal

```

```

proof

```

```

fix trn

```

```

assume trn: Comp.validTrans trn Comp.reach (Comp.srcOf trn)

```

```

then show nValidTrans (translateTrans trn) using new-node

```

```

proof (cases trn)

```

```

case (Trans2 sSub trnNode)

```

```

with trn have Node.reach (srcOf NID trnNode) by (auto elim: Comp.reach-reach12)

```

```

with trn Trans2 show ?thesis using Sub.isCom-tgtNodeOf[of NID -] by

```

```

(auto simp: isCom2-def)

```

```

next

```

```

case (CTrans trnSub trnNode)

```

```

with trn obtain sSub nid trn1 where trnSub: trnSub = LTrans sSub nid

```

```

trn1

```

```

by (auto elim: Sync.elims)

```

```

then have lreach nid (srcOf nid trn1) and lreach NID (srcOf NID trnNode)

```

```

using CTrans trn by (auto elim!: Comp.reach-reach12 Sub.reach-lreach)

```

```

then show ?thesis using CTrans trn trnSub by (auto elim: Sub.nValidTrans.elims)

```

```

qed (auto elim!: Sub.nValidTrans.elims split: if-split-asm)

```

```

next

```

```

fix trn' s

```

```

assume trn': nValidTrans trn' nSrcOf trn' = translateState s Comp.reach s

```

```

then obtain trn where Comp.validTrans trn Comp.srcOf trn = s translateTrans
trn = trn'

```

```

proof (cases trn')

```

```

case (LTrans s' nid trn)

```

```

show thesis proof cases

```

```

assume nid = NID

```

```

then show thesis using trn' LTrans

```

```

by (cases s; intro that[of Trans2 (fst s) trn]) (auto simp: isCom2-def)

```

```

next

```

```

assume nid ≠ NID

```

```

then show thesis using trn' LTrans

```

```

by (cases s; intro that[of Trans1 (snd s) (LTrans (fst s) nid trn)]) auto

```

```

qed

```

```

next

```

```

case (CTrans s' nid1 trn1 nid2 trn2)

```

```

show thesis proof cases

```

```

assume nid1 = NID ∨ nid2 = NID

```

```

then show thesis proof
  assume  $nid1 = NID$ 
  then show thesis using  $trn'$  CTrans new-node
    by (cases s; intro that[of ctrans.CTrans (LTrans (fst s) nid2 trn2) trn1]]
      (auto simp: isCom2-def)
  next
    assume  $nid2 = NID$ 
    then show thesis using  $trn'$  CTrans new-node
      by (cases s; intro that[of ctrans.CTrans (LTrans (fst s) nid1 trn1) trn2]]
        (auto simp: isCom2-def)
    qed
  next
    assume  $\neg (nid1 = NID \vee nid2 = NID)$ 
    then show thesis using  $trn'$  CTrans
      by (cases s; intro that[of Trans1 (snd s) (CTrans (fst s) nid1 trn1 nid2
trn2)]] auto)
    qed
  qed
then show  $\exists trn. Comp.validTrans\ trn \wedge Comp.srcOf\ trn = s \wedge translateTrans$ 
 $trn = trn'$  by auto
next
  fix  $trn$ 
  assume  $trn: Comp.validTrans\ trn\ Comp.reach\ (Comp.srcOf\ trn)$ 
  show  $nSrcOf\ (translateTrans\ trn) = translateState\ (Comp.srcOf\ trn)$ 
    using  $trn$  by (cases trn rule: translateTrans.cases) auto
  show  $nTgtOf\ (translateTrans\ trn) = translateState\ (Comp.tgtOf\ trn)$ 
    using  $trn\ new-node$  by (cases trn rule: translateTrans.cases) (auto intro:
fun-upd-twist)
  next
    show  $istate = translateState\ Comp.icstate$  unfolding Comp.icstate-def by auto
  next
    fix  $trn$ 
    assume  $trn: Comp.validTrans\ trn\ Comp.reach\ (Comp.srcOf\ trn)\ n\gamma\ (translateTrans$ 
 $trn)$ 
    then show  $Comp.\gamma\ trn \wedge (Some\ o\ translateObs)\ (Comp.g\ trn) = Some\ (ng$ 
 $(translateTrans\ trn))$ 
    proof (cases trn rule: translateTrans.cases)
      case ( $4\ sSub\ nid\ trnSub\ trnNode$ )
        with  $trn$  have  $lreach\ nid\ (srcOf\ nid\ trnSub)$  and  $lreach\ NID\ (srcOf\ NID$ 
 $trnNode)$ 
        by (auto elim!: Comp.reach-reach12 Sub.reach-lreach)
        with  $trn\ 4$  show ?thesis using isCom- $\gamma$ [of  $nid\ trnSub$ ] isCom- $\gamma$ [of  $NID$ 
 $trnNode$ ] by auto
        qed auto
      next
        fix  $trn$ 
        assume  $trn: Comp.validTrans\ trn\ Comp.reach\ (Comp.srcOf\ trn)\ Comp.\gamma\ trn$ 
        then show  $n\gamma\ (translateTrans\ trn) \vee (Some\ o\ translateObs)\ (Comp.g\ trn) =$ 
 $None$ 

```

```

proof (cases trn rule: translateTrans.cases)
  case (4 sSub nid trnSub trnNode)
    with trn have lreach nid (srcOf nid trnSub) and lreach NID (srcOf NID
trnNode)
    by (auto elim!: Comp.reach-reach12 Sub.reach-lreach)
    with trn 4 show ?thesis by auto
  qed auto
next
  fix trn
  assume trn: Comp.validTrans trn Comp.reach (Comp.srcOf trn) nφ (translateTrans
trn)
  then show Comp.φ trn ∧ (Some o translateVal) (Comp.f trn) = Some (nf
(translateTrans trn))
  proof (cases trn rule: translateTrans.cases)
    case (4 sSub nid trnSub trnNode)
      with trn have lreach nid (srcOf nid trnSub) and lreach NID (srcOf NID
trnNode)
      by (auto elim!: Comp.reach-reach12 Sub.reach-lreach)
      with trn 4 show ?thesis
      using sync-φ1-φ2[of nid trnSub NID trnNode] sync-φ1-φ2[of NID trnNode
nid trnSub] by auto
    qed auto
  next
  fix trn
  assume trn: Comp.validTrans trn Comp.reach (Comp.srcOf trn) Comp.φ trn
  then show nφ (translateTrans trn) ∨ (Some o translateVal) (Comp.f trn) =
None
  proof (cases trn rule: translateTrans.cases)
    case (4 sSub nid trnSub trnNode)
      with trn have lreach nid (srcOf nid trnSub) and lreach NID (srcOf NID
trnNode)
      by (auto elim!: Comp.reach-reach12 Sub.reach-lreach)
      with trn 4 show ?thesis by auto
    qed auto
  next
  fix trn
  assume Comp.T trn Comp.validTrans trn Comp.reach (Comp.srcOf trn)
  then show nT (translateTrans trn) by (cases trn rule: translateTrans.cases)
auto
  next
  fix vl' vl1' tr
  let ?vl1 = map invTranslateVal vl1'
  assume nB: nB vl' vl1' and tr: Comp.validFrom Comp.icstate tr
  and vl': these (map (Some o translateVal) (Comp.V tr)) = vl'
  moreover then have list-all compValidV (Comp.V tr) and list-all compValidV
?vl1
  by (auto intro: validFrom-istate-compValidV list-all-nValidV-compValidV list-all-compValidV-nValidV
simp: nB-def)
  ultimately have Comp.B (Comp.V tr) ?vl1 and list-all nValidV vl1'

```

```

unfolding nB-def Comp.B-def Sub.nB-def
by (auto simp: comp V-decomp V intro: n ValidV-subDecomp V list-all-comp ValidV-n ValidV)
then show  $\exists vl1. these (map (Some o translateVal) vl1) = vl1' \wedge Comp.B$ 
(Comp. V tr) vl1
by (intro exI[of - ?vl1], auto)
(metis list.map-comp map-translateVal-invTranslateVal these-map-Some)
qed

```

Security for the composition of the network with the new node:

```

lemma secure-new-node:
assumes Sub.secure and lsecure NID
shows secure
using assms by (auto intro: Trans.translate-secure Comp.secure1-secure2-secure)

end

```

Composing two sub-networks:

```

locale BD-Security-TS-Cut-Network = BD-Security-TS-Network
+
fixes nodesLeft and nodesRight
assumes
nodesLeftRight-disjoint: nodesLeft  $\cap$  nodesRight = {}
and
nodes-nodesLeftRight: nodes = nodesLeft  $\cup$  nodesRight
and
no-source-right: source  $\notin$  nodesRight
begin

```

```

lemma finite-nodesLeft: finite nodesLeft using finite-nodes nodes-nodesLeftRight
by auto

```

```

lemma finite-nodesRight: finite nodesRight using finite-nodes nodes-nodesLeftRight
by auto

```

```

sublocale Left: BD-Security-TS-Network-Change-Nodes where nodes' = nodesLeft
using finite-nodesLeft no-source-right  $\varphi$ -source nodes-nodesLeftRight
by (unfold-locales) auto

```

If the sub-network (potentially) containing the secret source is secure and all the nodes in the other sub-network are locally secure, then the composition is secure.

The proof proceeds by finite set induction on the set of non-source nodes, using the above infrastructure for adding new nodes to a network.

```

lemma merged-secure:
assumes Left.secure
and  $\forall nid \in nodesRight. lsecure\ nid$ 
shows secure
using finite-nodesRight assms no-source-right nodesLeftRight-disjoint  $\varphi$ -source un-
folding nodes-nodesLeftRight

```

```

proof (induction rule: finite-induct)
  case (insert nid nodesMerged)
    interpret Left': BD-Security-TS-Network-Change-Nodes where nodes' = nodesLeft
     $\cup$  nodesMerged
      using finite-nodes nodes-nodesLeftRight insert by (unfold-locales) auto
      interpret Node: BD-Security-TS istate nid validTrans nid srcOf nid tgtOf nid
         $\varphi$  nid f nid  $\gamma$  nid g nid T nid B nid .
      have secure1: Left'.secure and secure2: Node.secure using insert by auto
      interpret Comp: BD-Security-TS-Network-New-Node-NoSource
        where nodes = nodesLeft  $\cup$  nodesMerged and NID = nid
        using insert.premis insert.hyps by unfold-locales auto
      show ?case using secure1 secure2 using Comp.secure-new-node by auto
    qed auto

end

```

```

context BD-Security-TS-Network
begin

```

Putting it all together:

```

theorem network-secure:
assumes  $\forall$  nid  $\in$  nodes. lsecure nid
shows secure
proof (cases source  $\in$  nodes)
  case True
    interpret BD-Security-TS-Cut-Network where nodesLeft = {source} and
    nodesRight = nodes - {source}
    using True by unfold-locales auto
    interpret Source-BD: BD-Security-Singleton-Source-Network by intro-locales

    show secure using assms Source-BD.translate-secure True by (intro merged-secure)
    auto
  next
    case False
    interpret BD-Security-TS-Cut-Network where nodesLeft = {} and nodesRight
    = nodes
    using False by unfold-locales auto
    interpret Empty-BD: BD-Security-Empty-TS-Network by intro-locales

    show secure using assms Empty-BD.trivially-secure by (intro merged-secure)
  qed

end

```

Translating composite secrets using a function *mergeSec*:

```

datatype ('nodeid, 'sec, 'msec) merged-sec = LMSec 'nodeid 'sec | CMSec 'msec

```

```

locale BD-Security-TS-Network-MergeSec =
  Net?: BD-Security-TS-Network istate validTrans srcOf tgtOf nodes comOf tgtN-

```

```

odeOf sync  $\varphi$  f
for istate :: 'nodeid  $\Rightarrow$  'state
and validTrans :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  bool
and srcOf :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'state
and tgtOf :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'state
and nodes :: 'nodeid set
and comOf :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  com
and tgtNodeOf :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'nodeid
and sync :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  bool
and  $\varphi$  :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  bool
and f :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'sec +
fixes mergeSec :: 'nodeid  $\Rightarrow$  'sec  $\Rightarrow$  'nodeid  $\Rightarrow$  'sec  $\Rightarrow$  'msec
begin

inductive compSec :: ('nodeid  $\Rightarrow$  'sec list)  $\Rightarrow$  ('nodeid, 'sec, 'msec) merged-sec list
 $\Rightarrow$  bool
where
  Nil: compSec ( $\lambda$ -. []) []
  | Local:  $\llbracket$  compSec sls sl; isComV nid s  $\longrightarrow$  tgtNodeOfV nid s  $\notin$  nodes; nid  $\in$  nodes  $\rrbracket$ 
 $\implies$  compSec (sls(nid := s # sls nid)) (LMSec nid s # sl)
  | Comm:  $\llbracket$  compSec sls sl; nid1  $\in$  nodes; nid2  $\in$  nodes; nid1  $\neq$  nid2;
comOfV nid1 s1 = Send; tgtNodeOfV nid1 s1 = nid2;
comOfV nid2 s2 = Recv; tgtNodeOfV nid2 s2 = nid1;
syncV nid1 s1 nid2 s2  $\rrbracket$ 
 $\implies$  compSec (sls(nid1 := s1 # sls nid1, nid2 := s2 # sls nid2))
(CMSec (mergeSec nid1 s1 nid2 s2) # sl)

definition nB :: ('nodeid, 'sec, 'msec) merged-sec list  $\Rightarrow$  ('nodeid, 'sec, 'msec)
merged-sec list  $\Rightarrow$  bool where
nB sl sl'  $\equiv$   $\forall$  sls. compSec sls sl  $\longrightarrow$ 
( $\exists$  sls'. compSec sls' sl'  $\wedge$  ( $\forall$  nid  $\in$  nodes. B nid (sls nid) (sls' nid)))

fun nf :: ('nodeid, 'state, 'trans) ntrans  $\Rightarrow$  ('nodeid, 'sec, 'msec) merged-sec where
nf (LTrans s nid trn) = LMSec nid (f nid trn)
| nf (CTrans s nid1 trn1 nid2 trn2) = CMSec (mergeSec nid1 (f nid1 trn1) nid2
(f nid2 trn2))

sublocale BD-Security-TS istate nValidTrans nSrcOf nTgtOf n $\varphi$  nf n $\gamma$  ng nT nB
.

fun translateSec :: ('nodeid, 'sec) nvalue  $\Rightarrow$  ('nodeid, 'sec, 'msec) merged-sec where
translateSec (LVal nid s) = LMSec nid s
| translateSec (CVal nid1 s1 nid2 s2) = CMSec (mergeSec nid1 s1 nid2 s2)

lemma decompV-Cons-LVal: decompV (LVal nid s # sl) = (decompV sl)(nid :=
s # decompV sl nid)
by auto

```

lemma *decompV-Cons-CVal*:
assumes $nid1 \neq nid2$
shows $decompV (CVal\ nid1\ s1\ nid2\ s2\ \#\ sl) = (decompV\ sl)(nid1 := s1\ \# decompV\ sl\ nid1, nid2 := s2\ \# decompV\ sl\ nid2)$
using *assms* **by** *auto*

lemma *nValidV-compSec*:
assumes *list-all nValidV sl*
shows $compSec (decompV\ sl) (map\ translateSec\ sl)$
using *assms* **proof** (*induction sl*)
 case Nil **then show** *?case* **using** *compSec.Nil* **by** *auto*
next
 case (Cons s sl)
 then have $sl: compSec (decompV\ sl) (map\ translateSec\ sl)$ **by** *auto*
 show *?case* **proof** (*cases s*)
 case (LVal nid1 s1)
 moreover with *Cons sl* **have** $compSec ((decompV\ sl)(nid1 := s1\ \# decompV\ sl\ nid1)) (LMSec\ nid1\ s1\ \# map\ translateSec\ sl)$
 by (*intro compSec.Local*) *auto*
 ultimately show *?thesis unfolding LVal decompV-Cons-LVal[symmetric]* **by**
 (*auto simp del: decompV.simps*)
 next
 case (CVal nid1 s1 nid2 s2)
 moreover with *Cons sl* **have** $compSec ((decompV\ sl)(nid1 := s1\ \# decompV\ sl\ nid1, nid2 := s2\ \# decompV\ sl\ nid2)) (CMSec (mergeSec\ nid1\ s1\ nid2\ s2)\ \# map\ translateSec\ sl)$
 by (*intro compSec.Comm*) *auto*
 moreover have $n: nid1 \neq nid2$ **using** *CVal Cons* **by** *auto*
 ultimately show *?thesis unfolding CVal decompV-Cons-CVal[OF n, symmetric]* **by** (*auto simp del: decompV.simps*)
 qed
qed

lemma *compSecE*:
assumes *compSec sls sl*
obtains sl' **where** $decompV\ sl' = sls$ **and** $map\ translateSec\ sl' = sl$ **and** *list-all nValidV sl'*
using *assms* **proof** (*induction*)
 case Nil **from** *this[of []]* **show** *thesis* **by** *auto*
next
 case (Local sls sl nid s)
 show *thesis* **proof** (*rule Local.IH*)
 fix sl'
 assume $decompV\ sl' = sls$ **and** $map\ translateSec\ sl' = sl$ **and** *list-all nValidV sl'*
 with *Local.hyps* **show** *thesis* **by** (*intro Local.prem[s of LVal nid s # sl']*) *auto*
 qed
next

```

case (Comm sls sl nid1 nid2 s1 s2)
show thesis proof (rule Comm.IH)
  fix sl'
  assume decompV sl' = sls and map translateSec sl' = sl and list-all nValidV
sl'
  with Comm.hyps show thesis by (intro Comm.prem[of CVal nid1 s1 nid2 s2
# sl']) auto
  qed
qed

interpretation Trans: BD-Security-TS-Trans istate nValidTrans nSrcOf nTgtOf
nφ Net.nf nγ ng nT Net.nB
istate nValidTrans nSrcOf nTgtOf nφ nf nγ ng
nT nB
id id Some Some o translateSec

proof (unfold-locals, goal-cases)
  case (8 trn)
  then show ?case by (cases trn) auto
next
  case (11 sl' sl1' tr)
  then have list-all nValidV (Net.V tr) by (auto intro: validFrom-nValidV reach.Istate)
  then have compSec (decompV (Net.V tr)) (map translateSec (Net.V tr)) by
(intro nValidV-compSec)
  then obtain sls1 where compSec sls1 sl1' and  $\forall \textit{nid} \in \textit{nodes}. B \textit{nid} (\textit{decompV}
(Net.V tr) nid) (sls1 nid)$ 
  using  $\langle \textit{nB sl' sl1'} \rangle \langle \textit{these (map (Some \circ translateSec) (Net.V tr)) = sl'} \rangle$ 
unfolding nB-def by auto
  moreover then obtain sl1 where decompV sl1 = sls1 and list-all nValidV sl1
and map translateSec sl1 = sl1' by (elim compSecE)
  ultimately show ?case unfolding Net.nB-def by auto
qed auto

theorem network-secure:
assumes  $\forall \textit{nid} \in \textit{nodes}. lsecure \textit{nid}$ 
shows secure
using assms Net.network-secure Trans.translate-secure by blast

end

```

```

context BD-Security-TS-Network
begin

```

In order to formalize a result about preserving the notion of secrets of the source node upon composition, we define a notion of synchronization of secrets of the source and another node.

```

inductive srcSyncV :: 'nodeid  $\Rightarrow$  'val list  $\Rightarrow$  'val list  $\Rightarrow$  bool
for nid :: 'nodeid
where
  Nil: srcSyncV nid [] []

```

| *Other*: $\llbracket \text{srcSyncV } nid \text{ vlSrc vlNode}; \neg \text{isComV } source \ v \vee \text{tgtNodeOfV } source \ v \neq nid \rrbracket$
 $\implies \text{srcSyncV } nid \ (v \# \text{vlSrc}) \ \text{vlNode}$
 | *Send*: $\llbracket \text{srcSyncV } nid \ \text{vlSrc } \text{vlNode}; \text{comOfV } source \ vSrc = \text{Send}; \text{comOfV } nid \ vNode = \text{Recv};$
 $\text{tgtNodeOfV } source \ vSrc = nid; \text{tgtNodeOfV } nid \ vNode = source;$
 $\text{syncV } source \ vSrc \ nid \ vNode \rrbracket \implies \text{srcSyncV } nid \ (vSrc \# \text{vlSrc}) \ (vNode \# \text{vlNode})$
 | *Recv*: $\llbracket \text{srcSyncV } nid \ \text{vlSrc } \text{vlNode}; \text{comOfV } source \ vSrc = \text{Recv}; \text{comOfV } nid \ vNode = \text{Send};$
 $\text{tgtNodeOfV } source \ vSrc = nid; \text{tgtNodeOfV } nid \ vNode = source;$
 $\text{syncV } nid \ vNode \ source \ vSrc \rrbracket \implies \text{srcSyncV } nid \ (vSrc \# \text{vlSrc}) \ (vNode \# \text{vlNode})$

Sanity check that this is equivalent to a more general notion of binary secret synchronisation applied to source secrets and target secrets, where the latter do not contain internal secrets (in line with the assumption of unique secret polarization).

inductive $\text{binSyncV} :: 'nodeid \Rightarrow 'nodeid \Rightarrow 'val \ \text{list} \Rightarrow 'val \ \text{list} \Rightarrow \text{bool}$
for $nid1 \ nid2 :: 'nodeid$

where

$\text{Nil}: \text{binSyncV } nid1 \ nid2 \ [] \ []$
 | *Int1*: $\llbracket \text{binSyncV } nid1 \ nid2 \ \text{vl1 } \text{vl2}; \neg \text{isComV } nid1 \ v \vee \text{tgtNodeOfV } nid1 \ v \neq nid2 \rrbracket$
 $\implies \text{binSyncV } nid1 \ nid2 \ (v \# \text{vl1}) \ \text{vl2}$
 | *Int2*: $\llbracket \text{binSyncV } nid1 \ nid2 \ \text{vl1 } \text{vl2}; \neg \text{isComV } nid2 \ v \vee \text{tgtNodeOfV } nid2 \ v \neq nid1 \rrbracket$
 $\implies \text{binSyncV } nid1 \ nid2 \ \text{vl1} \ (v \# \text{vl2})$
 | *Send*: $\llbracket \text{binSyncV } nid1 \ nid2 \ \text{vl1 } \text{vl2}; \text{comOfV } nid1 \ v1 = \text{Send}; \text{comOfV } nid2 \ v2 = \text{Recv};$
 $\text{tgtNodeOfV } nid1 \ v1 = nid2; \text{tgtNodeOfV } nid2 \ v2 = nid1;$
 $\text{syncV } nid1 \ v1 \ nid2 \ v2 \rrbracket \implies \text{binSyncV } nid1 \ nid2 \ (v1 \# \text{vl1}) \ (v2 \# \text{vl2})$
 | *Recv*: $\llbracket \text{binSyncV } nid1 \ nid2 \ \text{vl1 } \text{vl2}; \text{comOfV } nid1 \ v1 = \text{Recv}; \text{comOfV } nid2 \ v2 = \text{Send};$
 $\text{tgtNodeOfV } nid1 \ v1 = nid2; \text{tgtNodeOfV } nid2 \ v2 = nid1;$
 $\text{syncV } nid2 \ v2 \ nid1 \ v1 \rrbracket \implies \text{binSyncV } nid1 \ nid2 \ (v1 \# \text{vl1}) \ (v2 \# \text{vl2})$

lemma srcSyncV-binSyncV :

assumes $source \in \text{nodes}$ **and** $nid2 \in \text{nodes}$

shows $\text{srcSyncV } nid2 \ \text{vl1 } \text{vl2} \longleftrightarrow (\text{binSyncV } source \ nid2 \ \text{vl1 } \text{vl2} \wedge \text{list-all } (\lambda v. \text{isComV } nid2 \ v \wedge \text{tgtNodeOfV } nid2 \ v = source) \ \text{vl2})$

(**is** $?l \longleftrightarrow ?r$)

proof

assume $?l$

then show $?r$ **using** assms **by** (*induction rule: srcSyncV.induct*) (*auto intro: binSyncV.intros*)

next

assume $?r$

then have $\text{binSyncV } source \ nid2 \ \text{vl1 } \text{vl2}$

and $\text{list-all } (\lambda v. \text{isComV } nid2 \ v \wedge \text{tgtNodeOfV } nid2 \ v = source) \ \text{vl2}$ **by** *auto*

then show ?l **by** (induction rule: binSyncV.induct) (auto intro: srcSyncV.intros)
qed

end

We can obtain a security property for the network w.r.t. the original declassification bound of the secret issuer node, if that bound is suitably reflected in the bounds of all the other nodes, i.e. the bounds of the receiving nodes do not declassify any more confidential information than is already declassified by the bound of the secret issuer node.

locale *BD-Security-TS-Network-Preserve-Source-Security* = *Net?*: *BD-Security-TS-Network*
+

assumes *source-in-nodes*: *source* \in *nodes*

and *source-secure*: *lsecure source*

and *B-source-in-B-sinks*: \bigwedge *nid tr vl' vl1*.

\llbracket *B source (lV source tr) vl1; srcSyncV nid (lV source tr) vl'*;

lValidFrom source (istate source) tr; never (T source) tr;

nid \in *nodes*; *nid* \neq *source* \rrbracket

$\implies (\exists vl1'. B\ nid\ vl'\ vl1' \wedge srcSyncV\ nid\ vl1\ vl1')$

begin

abbreviation *nodes'* \equiv *nodes* - {*source*}

fun *nf'* **where**

nf' (*LTrans s nid trn*) = *f source trn*

| *nf'* (*CTrans s nid1 trn1 nid2 trn2*) = (if *nid1* = *source* then *f source trn1* else *f source trn2*)

fun *translateVal* **where**

translateVal (LVal nid v) = *v*

| *translateVal (CVal nid1 v1 nid2 v2)* = (if *nid1* = *source* then *v1* else *v2*)

definition *isProjectionOf* **where**

isProjectionOf p vl = (\forall *nid* \in *nodes'*. *srcSyncV nid vl (p nid)*)

lemma *nValidV-tgtNodeOf*:

assumes *list-all nValidV vl'*

shows *list-all* (λv . *isComV source v* \longrightarrow *tgtNodeOfV source v* \neq *source*) (*decompV vl' source*)

using *assms* **proof** (*induction vl'*)

case (*Cons v vl'*) **then show** ?case **by** (*cases v*) *auto*

qed *auto*

lemma *lValidFrom-source-tgtNodeOfV*:

assumes *lValidFrom source s tr*

and *lreach source s*

shows *list-all* (λv . *isComV source v* \longrightarrow *tgtNodeOfV source v* \neq *source*) (*lV source tr*)

(**is** ?goal *tr*)

```

proof –
  interpret Node: BD-Security-TS istate source validTrans source srcOf source
  tgtOf source
     $\varphi$  source f source  $\gamma$  source g source T source B source .
  from assms show ?thesis proof (induction tr arbitrary: s)
    case (Cons trn tr s)
      have ?goal tr using Cons.prems by (intro Cons.IH[of tgtOf source trn]) (auto
  intro: Node.reach.Step)
      then show ?case using Cons.prems isCom-tgtNodeOf by (cases  $\varphi$  source
  trn) auto
      qed auto
    qed

lemma merge-projection:
assumes isProjectionOf p vl
and list-all ( $\lambda v. \text{isComV source } v \longrightarrow \text{tgtNodeOfV source } v \neq \text{source}$ ) vl
obtains vl' where  $\forall \text{nid} \in \text{nodes}'. \text{decompV } vl' \text{ nid} = p \text{ nid}$ 
  and  $\text{decompV } vl' \text{ source} = vl$ 
  and  $\text{map translateVal } vl' = vl$ 
  and list-all nValidV vl'
using assms proof (induction vl arbitrary: p)
  case (Nil p)
    from Nil.prems(2) show thesis
    by (intro Nil.prems(1)[of []]) (auto simp: isProjectionOf-def elim!: ballE
  srcSyncV.cases)
  next
    case (Cons v vl p)
      show thesis proof (cases isComV source v  $\wedge$  tgtNodeOfV source v  $\in$  nodes)
        case False
          show thesis proof (rule Cons.IH[of p])
            show isProjectionOf p vl
              using Cons(3) False unfolding isProjectionOf-def by (auto elim:
  srcSyncV.cases)
            show list-all ( $\lambda v. \text{isComV source } v \longrightarrow \text{tgtNodeOfV source } v \neq \text{source}$ ) vl
              using Cons(4) by auto
          next
            fix vl'
              assume  $\forall \text{nid} \in \text{nodes}'. \text{decompV } vl' \text{ nid} = p \text{ nid}$  and  $\text{decompV } vl' \text{ source}$ 
  = vl
                and  $\text{map translateVal } vl' = vl$  and list-all nValidV vl'
              then show thesis
                using False source-in-nodes by (intro Cons(2)[of LVal source v # vl'])
            auto
          qed
        next
          case True
            let ?tgt = tgtNodeOfV source v
            from True Cons(4) have nodes': ?tgt  $\in$  nodes' by auto
            with Cons(3) obtain vn vln

```

```

where p: p ?tgt = vn # vln and cmp: srcSyncV ?tgt (v # vl) (vn # vln)
  using True unfolding isProjectionOf-def
  by (cases p ?tgt) (auto elim!: ballE elim: srcSyncV.cases)
show thesis proof (rule Cons.IH[of p(?tgt := vln)])
  show isProjectionOf (p(?tgt := vln)) vl
    using Cons(3) True cmp unfolding isProjectionOf-def by (auto elim:
srcSyncV.cases)
  show list-all (λv. isComV source v → tgtNodeOfV source v ≠ source) vl
    using Cons(4) by auto
next
fix vl'
assume vl': ∀ nid ∈ nodes'. decompV vl' nid = (p(?tgt := vln)) nid
  decompV vl' source = vl map translateVal vl' = vl list-all nValidV
vl'
  show thesis proof cases
    assume comOfV source v = Send
    then show thesis using vl' p source-in-nodes True nodes' cmp
      by (intro Cons(2)[of CVal source v ?tgt vn # vl']) (auto elim:
srcSyncV.cases)
    next
    assume comOfV source v ≠ Send
    then show thesis using vl' p source-in-nodes True nodes' cmp
      by (intro Cons(2)[of CVal ?tgt vn source v # vl']) (auto elim:
srcSyncV.cases)
  qed
qed
qed
qed

lemma translateVal-decompV:
assumes validFrom s tr
and reach s
shows map translateVal (V tr) = decompV (V tr) source
using assms proof (induction tr arbitrary: s)
case (Cons trn tr s)
  then have tr: validFrom (nTgtOf trn) tr and r: reach (nTgtOf trn)
  unfolding validFrom-Cons by (auto intro: reach.Step[of s trn])
  show ?case proof (cases trn)
  case (LTrans s' nid trn')
    moreover then have φ nid trn' → nid = source
    using Cons.prem1 Net.φ-source[of nid trn'] reach-lreach by auto
    ultimately show ?thesis using Cons.IH[OF tr r] by (cases nφ trn) auto
  next
  case (CTrans s' nid1 trn1 nid2 trn2)
    moreover then have nφ trn → (nid1 = source ∨ nid2 = source)
    using Cons.prem1 Net.φ-source[of nid1 trn1] Net.φ-source[of nid2 trn2]
reach-lreach
    by (cases nid1 ≠ source) auto
    ultimately show ?thesis using Cons.IH[OF tr r] by (cases nφ trn) auto

```

qed
qed auto

lemma *srcSyncV-decompV*:
assumes *tr*: *validFrom s tr*
and *s*: *reach s*
and *nid* \in *nodes* **and** *nid* \neq *source*
shows *srcSyncV nid (decompV (V tr) source) (decompV (V tr) nid)*
using *tr s* **proof** (*induction tr arbitrary: s*)
 case (*Cons trn tr s*)
 then have *trn*: *nValidTrans trn* **and** *tr*: *validFrom (nTgtOf trn) tr* **and** *r*:
reach (nTgtOf trn)
 unfolding *validFrom-Cons* **by** (*auto intro: reach.Step[of s trn]*)
 show *?case* **proof** (*cases trn*)
 case (*LTrans s' nid' trn'*)
 show *?thesis* **proof** (*cases φ nid' trn'*)
 case *True*
 then have *nid' = source*
 using *Cons.premis Net. φ -source[of nid' trn'] reach-lreach LTrans* **by**
auto
 then show *?thesis* **using** *Cons.IH[OF tr r] Cons.premis LTrans assms(3,4)*
True
 by (*auto intro!: srcSyncV.Other elim: reach-lreach[of s nid']*)
 next
 case *False*
 then show *?thesis* **using** *Cons.IH[OF tr r] LTrans* **by** *auto*
 qed
 next
 case (*CTrans s' nid1 trn1 nid2 trn2*)
 have *r1*: *lreach nid1 (s' nid1)* **and** *r2*: *lreach nid2 (s' nid2)*
 using *CTrans reach-lreach Cons.premis* **by** *auto*
 show *?thesis* **proof** (*cases*)
 assume φ : *n φ trn*
 then have *nid1 = source \vee nid2 = source*
 using *CTrans Cons.premis Net. φ -source[of nid1 trn1] Net. φ -source[of*
nid2 trn2] *reach-lreach*
 by (*cases nid1 \neq source*) *auto*
 moreover have φ *nid1 trn1* **and** φ *nid2 trn2* **using** *CTrans trn r1 r2 φ*
sync- φ 1- φ 2 **by** *auto*
 moreover then have *comOfV nid1 (f nid1 trn1) = comOf nid1 trn1*
 and *isCom nid1 trn1 \longrightarrow tgtNodeOfV nid1 (f nid1 trn1) =*
tgtNodeOf nid1 trn1
 and *comOfV nid2 (f nid2 trn2) = comOf nid2 trn2*
 and *isCom nid2 trn2 \longrightarrow tgtNodeOfV nid2 (f nid2 trn2) =*
tgtNodeOf nid2 trn2
 and *syncV nid1 (f nid1 trn1) nid2 (f nid2 trn2)*
 using *CTrans trn r1 r2* **by** (*auto intro: sync-syncV*)
 ultimately show *?thesis*
 using *Cons.IH[OF tr r] trn assms(3,4) CTrans*

```

    using srcSyncV.Send[OF Cons.IH[OF tr r], of f nid1 trn1 f nid2 trn2]
    using srcSyncV.Recv[OF Cons.IH[OF tr r], of f nid2 trn2 f nid1 trn1]
    using srcSyncV.Other[OF Cons.IH[OF tr r], of f nid1 trn1]
    using srcSyncV.Other[OF Cons.IH[OF tr r], of f nid2 trn2]
    by auto
  next
    assume  $\neg n\varphi$  trn
    with Cons.IH[OF tr r] show ?thesis by auto
  qed
qed (auto intro: srcSyncV.Nil)

sublocale BD-Security-TS-Trans istate nValidTrans nSrcOf nTgtOf n $\varphi$  nf n $\gamma$  ng
nT nB
      istate nValidTrans nSrcOf nTgtOf n $\varphi$  nf' n $\gamma$  ng nT B
source
      id id Some Some o translateVal
proof unfold-locales
  fix trn
  assume trn: nValidTrans trn and r: reach (nSrcOf trn) and  $\varphi$ : n $\varphi$  (id trn)
  show n $\varphi$  trn  $\wedge$  (Some  $\circ$  translateVal) (nf trn) = Some (nf' (id trn))
  proof (cases trn)
    case (LTrans s nid trn')
    moreover then have nid = source using trn r  $\varphi$  Net. $\varphi$ -source[of nid trn']
    reach-lreach by auto
    ultimately show ?thesis using  $\varphi$  by auto
  next
    case (CTrans s nid1 trn1 nid2 trn2)
    moreover then have nid1 = source  $\vee$  nid2 = source
    using trn r  $\varphi$  Net. $\varphi$ -source[of nid1 trn1] Net. $\varphi$ -source[of nid2 trn2]
    reach-lreach[OF r]
    by (cases nid1  $\neq$  source) auto
    ultimately show ?thesis using  $\varphi$  by auto
  qed
next
  fix vl' vl1' tr
  interpret Source: Transition-System istate source validTrans source srcOf source
  tgtOf source .
  assume B source vl' vl1' and tr: validFrom istate tr and nT: never nT tr
  and vl': these (map (Some  $\circ$  translateVal) (V tr)) = vl'
  then have B: B source (decompV (V tr) source) vl1'
  using reach.Istate by (auto simp: translateVal-decompV)
  then obtain tr1 where tr1: lValidFrom source (istate source) tr1 and lV source
  tr1 = vl1'
  using source-secure validFrom-lValidFrom[OF tr, of source] decompV-decomp[OF
  tr reach.Istate] nT
  unfolding Abstract-BD-Security.secure-def by (auto intro: nTT-TT)
  then have  $\forall$  nid  $\in$  nodes'. srcSyncV nid (decompV (V tr) source) (decompV (V

```

```

\lambda v. \text{isComV source } v \longrightarrow \text{tgtNodeOfV source } v \neq
source) vl1'
  using B tr vl' reach.Istate srcSyncV-decompV nValidV-tgtNodeOf validFrom-nValidV
  using lValidFrom-source-tgtNodeOfV[OF tr1 Source.reach.Istate]
  by (auto simp: translateVal-decompV)
  then have  $\exists p. \forall \text{nid} \in \text{nodes}'. B \text{nid} (\text{decompV } (V \text{tr}) \text{nid}) (p \text{nid}) \wedge \text{srcSyncV}$ 
nid vl1' (p nid)
  using B B-source-in-B-sinks decompV-decomp[OF tr reach.Istate] validFrom-lValidFrom[OF
tr, of source]
  using nT nTT-TT by (intro bchoice) auto
  then obtain p where isProjectionOf p vl1' and B':  $\forall \text{nid} \in \text{nodes}'. B \text{nid}$ 
(decompV (V tr) nid) (p nid)
  unfolding isProjectionOf-def by auto
  then obtain vl1 where p:  $\forall \text{nid} \in \text{nodes}'. \text{decompV } vl1 \text{nid} = p \text{nid}$  and vl1':
decompV vl1 source = vl1'
    and map translateVal vl1 = vl1' and list-all nValidV vl1
  using tgt-vl1' by (elim merge-projection) auto
  moreover have  $\forall \text{nid} \in \text{nodes}. B \text{nid} (\text{decompV } (V \text{tr}) \text{nid}) (\text{decompV } vl1 \text{nid})$ 
proof
  fix nid
  assume nid  $\in \text{nodes}$ 
  then show B nid (decompV (V tr) nid) (decompV vl1 nid)
    using B vl1' B' p by (cases nid = source) auto
  qed
  ultimately show  $\exists vl1. \text{these } (\text{map } (\text{Some} \circ \text{translateVal}) vl1) = vl1' \wedge nB (V$ 
tr) vl1
    using B tr vl' reach.Istate
    by (intro exI[of - vl1]) (auto simp: nB-def)
  qed auto

theorem preserve-source-secure:
  assumes  $\forall \text{nid} \in \text{nodes}'. \text{lsecure } \text{nid}$ 
  shows secure
  using assms source-secure
  by (intro translate-secure network-secure ballI) auto

end

```

We can simplify the check that the bound of the source node is reflected in those of the other nodes with the help of a function mapping secrets communicated by the source node to those of the target nodes.

```

locale BD-Security-TS-Network-getTgtV = BD-Security-TS-Network +
  fixes getTgtV
  assumes getTgtV-Send:  $\text{comOfV source } v\text{Src} = \text{Send} \implies \text{tgtNodeOfV source } v\text{Src}$ 
= nid  $\implies \text{nid} \neq \text{source} \implies (\text{syncV source } v\text{Src } \text{nid } vn \longleftrightarrow vn = \text{getTgtV } v\text{Src})$ 
 $\wedge \text{tgtNodeOfV } \text{nid} (\text{getTgtV } v\text{Src}) = \text{source} \wedge \text{comOfV } \text{nid} (\text{getTgtV } v\text{Src}) = \text{Recv}$ 
  and getTgtV-Recv:  $\text{comOfV source } v\text{Src} = \text{Recv} \implies \text{tgtNodeOfV source } v\text{Src} =$ 
nid  $\implies \text{nid} \neq \text{source} \implies (\text{syncV } \text{nid } vn \text{ source } v\text{Src} \longleftrightarrow vn = \text{getTgtV } v\text{Src}) \wedge$ 

```

$tgtNodeOfV\ nid\ (getTgtV\ vSrc) = source \wedge comOfV\ nid\ (getTgtV\ vSrc) = Send$
begin

abbreviation $projectSrcV\ nid\ vlSrc$
 $\equiv map\ getTgtV\ (filter\ (\lambda v. isComV\ source\ v \wedge tgtNodeOfV\ source\ v = nid)\ vlSrc)$

lemma $srcSyncV-projectSrcV$:
assumes $nid \in nodes - \{source\}$
shows $srcSyncV\ nid\ vlSrc\ vln \longleftrightarrow vln = projectSrcV\ nid\ vlSrc$

proof

assume $srcSyncV\ nid\ vlSrc\ vln$

then show $vln = projectSrcV\ nid\ vlSrc$ **using** $assms\ getTgtV-Send\ getTgtV-Recv$

by $induction\ auto$

next

assume $vln = projectSrcV\ nid\ vlSrc$

moreover have $srcSyncV\ nid\ vlSrc\ (projectSrcV\ nid\ vlSrc)$

using $assms\ getTgtV-Send\ getTgtV-Recv$ **by** $(induction\ vlSrc)\ (auto\ intro:\ srcSyncV.intros)$

ultimately show $srcSyncV\ nid\ vlSrc\ vln$ **by** $simp$

qed

end

locale $BD-Security-TS-Network-Preserve-Source-Security-getTgtV = Net?: BD-Security-TS-Network-getTgtV$
 $+$

assumes $source-in-nodes: source \in nodes$

and $source-secure: lsecure\ source$

and $B-source-in-B-sinks: \bigwedge nid\ tr\ vl\ vl1.$

$\llbracket B\ source\ vl\ vl1; vl = lV\ source\ tr; lValidFrom\ source\ (istate\ source)\ tr; never\ (T\ source)\ tr;$

$nid \in nodes; nid \neq source \rrbracket$

$\implies B\ nid\ (projectSrcV\ nid\ vl)\ (projectSrcV\ nid\ vl1)$

begin

sublocale $BD-Security-TS-Network-Preserve-Source-Security$

using $source-in-nodes\ source-secure\ B-source-in-B-sinks\ srcSyncV-projectSrcV$

by $(unfold-locales)\ auto$

end

An alternative composition setup that derives parameters $comOfV$, $syncV$, etc. from $comOf$, $sync$, etc.

locale $BD-Security-TS-Network' = TS-Network\ istate\ validTrans\ srcOf\ tgtOf\ nodes$
 $comOf\ tgtNodeOf\ sync$

for

$istate :: ('nodeid, 'state) \ nstate$ **and** $validTrans :: 'nodeid \Rightarrow 'trans \Rightarrow bool$

and

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

and

```

    nodes :: 'nodeid set
and
    comOf :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  com
and
    tgtNodeOf :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'nodeid
and
    sync :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  bool
+
fixes
     $\varphi$  :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  bool and f :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'val
and
     $\gamma$  :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  bool and g :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'obs
and
    T :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  bool and B :: 'nodeid  $\Rightarrow$  'val list  $\Rightarrow$  'val list  $\Rightarrow$  bool
and
    source :: 'nodeid
assumes
    g-comOf:  $\bigwedge$  nid trn1 trn2.
      [[validTrans nid trn1; lreach nid (srcOf nid trn1);  $\gamma$  nid trn1;
        validTrans nid trn2; lreach nid (srcOf nid trn2);  $\gamma$  nid trn2;
        g nid trn2 = g nid trn1]]  $\implies$  comOf nid trn2 = comOf nid trn1
and
    f-comOf:  $\bigwedge$  nid trn1 trn2.
      [[validTrans nid trn1; lreach nid (srcOf nid trn1);  $\varphi$  nid trn1;
        validTrans nid trn2; lreach nid (srcOf nid trn2);  $\varphi$  nid trn2;
        f nid trn2 = f nid trn1]]  $\implies$  comOf nid trn2 = comOf nid trn1
and
    g-tgtNodeOf:  $\bigwedge$  nid trn1 trn2.
      [[validTrans nid trn1; lreach nid (srcOf nid trn1);  $\gamma$  nid trn1;
        validTrans nid trn2; lreach nid (srcOf nid trn2);  $\gamma$  nid trn2;
        g nid trn2 = g nid trn1]]  $\implies$  tgtNodeOf nid trn2 = tgtNodeOf nid trn1
and
    f-tgtNodeOf:  $\bigwedge$  nid trn1 trn2.
      [[validTrans nid trn1; lreach nid (srcOf nid trn1);  $\varphi$  nid trn1;
        validTrans nid trn2; lreach nid (srcOf nid trn2);  $\varphi$  nid trn2;
        f nid trn2 = f nid trn1]]  $\implies$  tgtNodeOf nid trn2 = tgtNodeOf nid trn1
and
    sync- $\varphi$ 1- $\varphi$ 2:
       $\bigwedge$  nid1 trn1 nid2 trn2.
        validTrans nid1 trn1  $\implies$  lreach nid1 (srcOf nid1 trn1)  $\implies$ 
        validTrans nid2 trn2  $\implies$  lreach nid2 (srcOf nid2 trn2)  $\implies$ 
        comOf nid1 trn1 = Send  $\implies$  tgtNodeOf nid1 trn1 = nid2  $\implies$ 
        comOf nid2 trn2 = Recv  $\implies$  tgtNodeOf nid2 trn2 = nid1  $\implies$ 
        sync nid1 trn1 nid2 trn2  $\implies$   $\varphi$  nid1 trn1  $\longleftrightarrow$   $\varphi$  nid2 trn2
and
    sync- $\varphi$ - $\gamma$ :
       $\bigwedge$  nid1 trn1 nid2 trn2.
        validTrans nid1 trn1  $\implies$  lreach nid1 (srcOf nid1 trn1)  $\implies$ 
        validTrans nid2 trn2  $\implies$  lreach nid2 (srcOf nid2 trn2)  $\implies$ 

```

$$\begin{aligned}
& comOf\ nid1\ trn1 = Send \implies tgtNodeOf\ nid1\ trn1 = nid2 \implies \\
& comOf\ nid2\ trn2 = Recv \implies tgtNodeOf\ nid2\ trn2 = nid1 \implies \\
& (\gamma\ nid1\ trn1 \implies \gamma\ nid2\ trn2 \implies \\
& \quad \exists\ trn1'\ trn2'. \\
& \quad validTrans\ nid1\ trn1' \wedge lreach\ nid1\ (srcOf\ nid1\ trn1') \wedge \gamma\ nid1\ trn1' \wedge g \\
& \quad nid1\ trn1' = g\ nid1\ trn1 \wedge \\
& \quad validTrans\ nid2\ trn2' \wedge lreach\ nid2\ (srcOf\ nid2\ trn2') \wedge \gamma\ nid2\ trn2' \wedge g \\
& \quad nid2\ trn2' = g\ nid2\ trn2 \wedge \\
& \quad sync\ nid1\ trn1'\ nid2\ trn2') \implies \\
& (\varphi\ nid1\ trn1 \implies \varphi\ nid2\ trn2 \implies \\
& \quad \exists\ trn1'\ trn2'. \\
& \quad validTrans\ nid1\ trn1' \wedge lreach\ nid1\ (srcOf\ nid1\ trn1') \wedge \varphi\ nid1\ trn1' \wedge f \\
& \quad nid1\ trn1' = f\ nid1\ trn1 \wedge \\
& \quad validTrans\ nid2\ trn2' \wedge lreach\ nid2\ (srcOf\ nid2\ trn2') \wedge \varphi\ nid2\ trn2' \wedge f \\
& \quad nid2\ trn2' = f\ nid2\ trn2 \wedge \\
& \quad sync\ nid1\ trn1'\ nid2\ trn2') \\
& \implies \\
& sync\ nid1\ trn1\ nid2\ trn2
\end{aligned}$$

and

$$isCom\text{-}\gamma: \bigwedge\ nid\ trn. validTrans\ nid\ trn \implies lreach\ nid\ (srcOf\ nid\ trn) \implies comOf\ nid\ trn = Send \vee comOf\ nid\ trn = Recv \implies \gamma\ nid\ trn$$

and

$$\begin{aligned}
& \varphi\text{-source}: \bigwedge\ nid\ trn. \llbracket validTrans\ nid\ trn; lreach\ nid\ (srcOf\ nid\ trn); \varphi\ nid\ trn; nid \\
& \neq\ source; nid \in\ nodes \rrbracket \\
& \implies isCom\ nid\ trn \wedge tgtNodeOf\ nid\ trn = source \wedge source \in
\end{aligned}$$

nodes

begin

definition $reachableO\ nid\ obs = (\exists\ trn. validTrans\ nid\ trn \wedge lreach\ nid\ (srcOf\ nid\ trn) \wedge \gamma\ nid\ trn \wedge g\ nid\ trn = obs)$

definition $reachableV\ nid\ sec = (\exists\ trn. validTrans\ nid\ trn \wedge lreach\ nid\ (srcOf\ nid\ trn) \wedge \varphi\ nid\ trn \wedge f\ nid\ trn = sec)$

definition $invO\ nid\ obs = inv\text{-}into\ \{trn. validTrans\ nid\ trn \wedge lreach\ nid\ (srcOf\ nid\ trn) \wedge \gamma\ nid\ trn\}\ (g\ nid)\ obs$

definition $invV\ nid\ sec = inv\text{-}into\ \{trn. validTrans\ nid\ trn \wedge lreach\ nid\ (srcOf\ nid\ trn) \wedge \varphi\ nid\ trn\}\ (f\ nid)\ sec$

definition $comOfO\ nid\ obs = (if\ reachableO\ nid\ obs\ then\ comOf\ nid\ (invO\ nid\ obs)\ else\ Internal)$

definition $tgtNodeOfO\ nid\ obs = tgtNodeOf\ nid\ (invO\ nid\ obs)$

definition $comOfV\ nid\ sec = (if\ reachableV\ nid\ sec\ then\ comOf\ nid\ (invV\ nid\ sec)\ else\ Internal)$

definition $tgtNodeOfV\ nid\ sec = tgtNodeOf\ nid\ (invV\ nid\ sec)$

definition $syncO\ nid1\ obs1\ nid2\ obs2 =$

$$\begin{aligned}
& (\exists\ trn1\ trn2. validTrans\ nid1\ trn1 \wedge lreach\ nid1\ (srcOf\ nid1\ trn1) \wedge \gamma\ nid1\ trn1 \\
& \wedge g\ nid1\ trn1 = obs1 \wedge \\
& \quad validTrans\ nid2\ trn2 \wedge lreach\ nid2\ (srcOf\ nid2\ trn2) \wedge \gamma\ nid2\ trn2 \wedge \\
& \quad g\ nid2\ trn2 = obs2 \wedge
\end{aligned}$$

sync nid1 trn1 nid2 trn2)

definition *syncV nid1 sec1 nid2 sec2 =*
 $(\exists trn1 trn2. \text{validTrans } nid1 trn1 \wedge \text{lreach } nid1 (\text{srcOf } nid1 trn1) \wedge \varphi \text{ } nid1 trn1$
 $\wedge f \text{ } nid1 trn1 = sec1 \wedge$
 $\text{validTrans } nid2 trn2 \wedge \text{lreach } nid2 (\text{srcOf } nid2 trn2) \wedge \varphi \text{ } nid2 trn2 \wedge$
 $f \text{ } nid2 trn2 = sec2 \wedge$
 $\text{sync } nid1 trn1 nid2 trn2)$

lemmas *comp-O-V-defs = comOfO-def tgtNodeOfO-def comOfV-def tgtNodeOfV-def*
syncO-def syncV-def
reachableO-def reachableV-def

lemma *reachableV-D:*
assumes *reachableV nid sec*
shows *validTrans nid (invV nid sec) and lreach nid (srcOf nid (invV nid sec))*
and $\varphi \text{ } nid (\text{invV } nid \text{ } sec)$ and $f \text{ } nid (\text{invV } nid \text{ } sec) = sec$
using *assms unfolding reachableV-def invV-def inv-into-def by (auto intro: someI2-ex)*

lemma *reachableO-D:*
assumes *reachableO nid obs*
shows *validTrans nid (invO nid obs) and lreach nid (srcOf nid (invO nid obs))*
and $\gamma \text{ } nid (\text{invO } nid \text{ } obs)$ and $g \text{ } nid (\text{invO } nid \text{ } obs) = obs$
using *assms unfolding reachableO-def invO-def inv-into-def by (auto intro: someI2-ex)*

sublocale *BD-Security-TS-Network*
where *comOfV = comOfV and tgtNodeOfV = tgtNodeOfV and syncV = syncV*
and comOfO = comOfO and tgtNodeOfO = tgtNodeOfO and syncO = syncO
proof (*unfold-locales, goal-cases*)
case (*1 nid trn*) **then show** *?case by (auto intro!: f-comOf reachableV-D simp:*
comp-O-V-defs) next
case (*2 nid trn*) **then show** *?case by (auto intro!: f-tgtNodeOf reachableV-D*
simp: comp-O-V-defs) next
case (*3 nid trn*) **then show** *?case by (auto intro!: g-comOf reachableO-D simp:*
comp-O-V-defs)
case (*4 nid trn*) **then show** *?case by (auto intro!: g-tgtNodeOf reachableO-D*
simp: comp-O-V-defs) next
case (*5 nid1 trn1 nid2 trn2*) **then show** *?case unfolding comp-O-V-defs by*
auto next
case (*6 nid1 trn1 nid2 trn2*) **then show** *?case unfolding comp-O-V-defs by*
auto next
case (*7 nid1 trn1 nid2 trn2*) **then show** *?case using sync- φ 1- φ 2 by blast next*
case (*8 nid1 trn1 nid2 trn2*) **then show** *?case unfolding comp-O-V-defs by*
(intro sync- φ - γ) next
case (*9 nid trn*) **then show** *?case by (intro isCom- γ) next*
case (*10 nid trn*) **then show** *?case by (intro φ -source)*
qed

lemma *comOf-invV:*
assumes *validTrans nid trn and lreach nid (srcOf nid trn) and $\varphi \text{ } nid trn$*

shows $comOf\ nid\ (invV\ nid\ (f\ nid\ trn)) = comOf\ nid\ trn$
using *assms* **by** (*auto intro!*: *f-comOf reachableV-D simp: reachableV-def*)

lemma *comOfV-SendE*:

assumes $comOfV\ nid\ v = Send$

obtains trn **where** $validTrans\ nid\ trn$ **and** $breach\ nid\ (srcOf\ nid\ trn)$ **and** $\varphi\ nid\ trn$ **and** $f\ nid\ trn = v$

and $comOf\ nid\ trn = Send$

using *assms* **unfolding** *comOfV-def* **by** (*auto simp: reachableV-def comOf-invV split: if-splits*)

lemma *comOfV-RecvE*:

assumes $comOfV\ nid\ v = Recv$

obtains trn **where** $validTrans\ nid\ trn$ **and** $breach\ nid\ (srcOf\ nid\ trn)$ **and** $\varphi\ nid\ trn$ **and** $f\ nid\ trn = v$

and $comOf\ nid\ trn = Recv$

using *assms* **unfolding** *comOfV-def* **by** (*auto simp: reachableV-def comOf-invV split: if-splits*)

fun *secComp* :: ('nodeid, 'val) nvalue list \Rightarrow bool **where**

$secComp\ [] = True$

| $secComp\ (LVal\ nid\ s\ \#\ sl) =$

$(secComp\ sl \wedge nid \in nodes \wedge$

$\neg(\exists trn. validTrans\ nid\ trn \wedge breach\ nid\ (srcOf\ nid\ trn) \wedge \varphi\ nid\ trn \wedge f\ nid\ trn = s \wedge$

$(comOf\ nid\ trn = Send \vee comOf\ nid\ trn = Recv) \wedge tgtNodeOf\ nid\ trn$

$\in nodes))$

| $secComp\ (CVal\ nid1\ s1\ nid2\ s2\ \#\ sl) =$

$(secComp\ sl \wedge nid1 \in nodes \wedge nid2 \in nodes \wedge nid1 \neq nid2 \wedge$

$(\exists trn1\ trn2. validTrans\ nid1\ trn1 \wedge breach\ nid1\ (srcOf\ nid1\ trn1) \wedge \varphi\ nid1\ trn1 \wedge f\ nid1\ trn1 = s1 \wedge$

$validTrans\ nid2\ trn2 \wedge breach\ nid2\ (srcOf\ nid2\ trn2) \wedge \varphi\ nid2\ trn2$

$\wedge f\ nid2\ trn2 = s2 \wedge$

$comOf\ nid1\ trn1 = Send \wedge tgtNodeOf\ nid1\ trn1 = nid2 \wedge$

$comOf\ nid2\ trn2 = Recv \wedge tgtNodeOf\ nid2\ trn2 = nid1 \wedge$

$sync\ nid1\ trn1\ nid2\ trn2))$

lemma *syncedSecs-iff-nValidV*: $secComp\ sl \longleftrightarrow list-all\ nValidV\ sl$

proof (*induction sl rule: secComp.induct*)

case 2 then show ?case **by** (*auto elim!: comOfV-SendE comOfV-RecvE*) **next**

case (3 $nid1\ v1\ nid2\ v2\ sl$)

have $nValidV\ (CVal\ nid1\ v1\ nid2\ v2) =$

$(\exists trn1\ trn2. nid1 \in nodes \wedge nid2 \in nodes \wedge nid1 \neq nid2 \wedge$

$validTrans\ nid1\ trn1 \wedge breach\ nid1\ (srcOf\ nid1\ trn1) \wedge \varphi\ nid1\ trn1 \wedge f\ nid1\ trn1 = v1 \wedge$

$validTrans\ nid2\ trn2 \wedge breach\ nid2\ (srcOf\ nid2\ trn2) \wedge \varphi\ nid2\ trn2 \wedge$

$f\ nid2\ trn2 = v2 \wedge$

$comOf\ nid1\ trn1 = Send \wedge tgtNodeOf\ nid1\ trn1 = nid2 \wedge$

$comOf\ nid2\ trn2 = Recv \wedge tgtNodeOf\ nid2\ trn2 = nid1 \wedge$

```

      sync nid1 trn1 nid2 trn2)
    by (auto simp: syncV-def) blast
  with 3 show ?case by auto
qed auto

```

lemma *nB-secComp*:

$$nB\ sl\ sl1 \longleftrightarrow (\forall\ nid \in\ nodes.\ B\ nid\ (decompV\ sl\ nid)\ (decompV\ sl1\ nid)) \wedge (secComp\ sl \longrightarrow secComp\ sl1)$$

unfolding *nB-def syncedSecs-iff-nValidV ..*

end

end

6 Combining independent secret sources

This theory formalizes the discussion about considering combined sources of secrets from [2, Appendix E].

theory *Independent-Secrets*

imports *Bounded-Deducibility-Security.BD-Security-TS*

begin

locale *Abstract-BD-Security-Two-Secrets =*

One: Abstract-BD-Security validSystemTrace V1 O1 B1 TT1

+ Two: Abstract-BD-Security validSystemTrace V2 O2 B2 TT2

for

validSystemTrace :: 'traces \Rightarrow bool

and

V1 :: 'traces \Rightarrow 'values1

and

O1 :: 'traces \Rightarrow 'observations1

and

B1 :: 'values1 \Rightarrow 'values1 \Rightarrow bool

and

TT1 :: 'traces \Rightarrow bool

and

V2 :: 'traces \Rightarrow 'values2

and

O2 :: 'traces \Rightarrow 'observations2

and

B2 :: 'values2 \Rightarrow 'values2 \Rightarrow bool

and

TT2 :: 'traces \Rightarrow bool

+

fixes

O :: 'traces \Rightarrow 'observations

assumes

O1-O: $O1\ tr = O1\ tr' \implies \text{validSystemTrace}\ tr \implies \text{validSystemTrace}\ tr' \implies O\ tr = O\ tr'$

and

O2-O: $O2\ tr = O2\ tr' \implies \text{validSystemTrace}\ tr \implies \text{validSystemTrace}\ tr' \implies O\ tr = O\ tr'$

and

O1-V2: $O1\ tr = O1\ tr' \implies \text{validSystemTrace}\ tr \implies \text{validSystemTrace}\ tr' \implies B1\ (V1\ tr)\ (V1\ tr') \implies V2\ tr = V2\ tr'$

and

O2-V1: $O2\ tr = O2\ tr' \implies \text{validSystemTrace}\ tr \implies \text{validSystemTrace}\ tr' \implies B2\ (V2\ tr)\ (V2\ tr') \implies V1\ tr = V1\ tr'$

and

O1-TT2: $O1\ tr = O1\ tr' \implies \text{validSystemTrace}\ tr \implies \text{validSystemTrace}\ tr' \implies B1\ (V1\ tr)\ (V1\ tr') \implies TT2\ tr = TT2\ tr'$

begin

definition $V\ tr = (V1\ tr, V2\ tr)$

definition $B\ vl\ vl' = (B1\ (\text{fst}\ vl)\ (\text{fst}\ vl') \wedge B2\ (\text{snd}\ vl)\ (\text{snd}\ vl'))$

definition $TT\ tr = (TT1\ tr \wedge TT2\ tr)$

sublocale *Abstract-BD-Security* $\text{validSystemTrace}\ V\ O\ B\ TT$.

theorem *two-secure*:

assumes *One.secure* **and** *Two.secure*

shows *secure*

unfolding *secure-def*

proof (*intro allI impI, elim conjE*)

fix $tr\ vl\ vl'$

assume $tr: \text{validSystemTrace}\ tr$ **and** $TT: TT\ tr$ **and** $B: B\ vl\ vl'$ **and** $V\text{-tr}: V\ tr = vl$

then obtain $vl1'\ vl2'$ **where** $vl: vl = (V1\ tr, V2\ tr)$ **and** $vl': vl' = (vl1', vl2')$

by (*cases vl, cases vl'*) (*auto simp: V-def*)

obtain tr' **where** $tr': \text{validSystemTrace}\ tr'$ **and** $O1: O1\ tr' = O1\ tr$ **and** $V1: V1\ tr' = vl1'$

using *assms(1)* $tr\ TT\ B$ **by** (*auto elim: One.secureE simp: TT-def B-def V-def vl vl'*)

then have $O': O\ tr' = O\ tr$ **and** $V2': V2\ tr = V2\ tr'$ **and** $TT2': TT2\ tr = TT2\ tr'$

using $B\ tr\ V1$ **by** (*auto intro: O1-O O1-V2 simp: O1-TT2 B-def vl vl'*)

obtain tr'' **where** $tr'': \text{validSystemTrace}\ tr''$ **and** $O2: O2\ tr'' = O2\ tr'$ **and** $V2: V2\ tr'' = vl2'$

using *assms(2)* $tr'\ TT2'\ TT\ B\ V2'$

by (*elim Two.secureE[of tr' vl2']*) (*auto simp: TT-def B-def vl vl'*)

moreover then have $O\ tr'' = O\ tr'$ **and** $V1\ tr' = V1\ tr''$

using $B\ tr'\ V2$ **by** (*auto intro: O2-O O2-V1 simp: B-def V2'\ vl vl'*)

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

unfolding $V\text{-def}\ V1\ vl'\ O'$ **by** *auto*

qed

end

locale *BD-Security-TS-Two-Secrets* =

One: *BD-Security-TS* *istate* *validTrans* *srcOf* *tgtOf* φ_1 *f1* γ_1 *g1* *T1* *B1*
+ *Two*: *BD-Security-TS* *istate* *validTrans* *srcOf* *tgtOf* φ_2 *f2* γ_2 *g2* *T2* *B2*
for *istate* :: 'state **and** *validTrans* :: 'trans \Rightarrow bool
and *srcOf* :: 'trans \Rightarrow 'state **and** *tgtOf* :: 'trans \Rightarrow 'state
and φ_1 :: 'trans \Rightarrow bool **and** *f1* :: 'trans \Rightarrow 'val1
and γ_1 :: 'trans \Rightarrow bool **and** *g1* :: 'trans \Rightarrow 'obs1
and *T1* :: 'trans \Rightarrow bool **and** *B1* :: 'val1 list \Rightarrow 'val1 list \Rightarrow bool
and φ_2 :: 'trans \Rightarrow bool **and** *f2* :: 'trans \Rightarrow 'val2
and γ_2 :: 'trans \Rightarrow bool **and** *g2* :: 'trans \Rightarrow 'obs2
and *T2* :: 'trans \Rightarrow bool **and** *B2* :: 'val2 list \Rightarrow 'val2 list \Rightarrow bool
+
fixes γ :: 'trans \Rightarrow bool **and** *g* :: 'trans \Rightarrow 'obs
assumes γ - γ_{12} : $\bigwedge tr\ trn. One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow \gamma\ trn \Longrightarrow \gamma_1\ trn$
 $\wedge\ \gamma_2\ trn$
and *O1*- γ : $\bigwedge tr\ tr'\ trn\ trn'. One.O\ tr = One.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma_1\ trn \Longrightarrow \gamma_1\ trn' \Longrightarrow g_1\ trn = g_1\ trn' \Longrightarrow \gamma\ trn = \gamma\ trn'$
and *O1*-*g*: $\bigwedge tr\ tr'\ trn\ trn'. One.O\ tr = One.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma_1\ trn \Longrightarrow \gamma_1\ trn' \Longrightarrow g_1\ trn = g_1\ trn' \Longrightarrow \gamma\ trn \Longrightarrow \gamma\ trn' \Longrightarrow g\ trn = g\ trn'$
and *O2*- γ : $\bigwedge tr\ tr'\ trn\ trn'. Two.O\ tr = Two.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma_2\ trn \Longrightarrow \gamma_2\ trn' \Longrightarrow g_2\ trn = g_2\ trn' \Longrightarrow \gamma\ trn = \gamma\ trn'$
and *O2*-*g*: $\bigwedge tr\ tr'\ trn\ trn'. Two.O\ tr = Two.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma_2\ trn \Longrightarrow \gamma_2\ trn' \Longrightarrow g_2\ trn = g_2\ trn' \Longrightarrow \gamma\ trn \Longrightarrow \gamma\ trn' \Longrightarrow g\ trn = g\ trn'$
and φ_2 - γ_1 : $\bigwedge tr\ trn. One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow \varphi_2\ trn \Longrightarrow \gamma_1\ trn$
and γ_1 - φ_2 : $\bigwedge tr\ tr'\ trn\ trn'. One.O\ tr = One.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma_1\ trn \Longrightarrow \gamma_1\ trn' \Longrightarrow g_1\ trn = g_1\ trn' \Longrightarrow \varphi_2\ trn = \varphi_2\ trn'$
and *g1*-*f2*: $\bigwedge tr\ tr'\ trn\ trn'. One.O\ tr = One.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma_1\ trn \Longrightarrow \gamma_1\ trn' \Longrightarrow g_1\ trn = g_1\ trn' \Longrightarrow \varphi_2\ trn \Longrightarrow \varphi_2\ trn' \Longrightarrow f_2\ trn = f_2\ trn'$
and φ_1 - γ_2 : $\bigwedge tr\ trn. One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow \varphi_1\ trn \Longrightarrow \gamma_2\ trn$
and γ_2 - φ_1 : $\bigwedge tr\ tr'\ trn\ trn'. Two.O\ tr = Two.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma_2\ trn \Longrightarrow \gamma_2\ trn' \Longrightarrow g_2\ trn = g_2\ trn' \Longrightarrow \varphi_1\ trn = \varphi_1\ trn'$
and *g2*-*f1*: $\bigwedge tr\ tr'\ trn\ trn'. Two.O\ tr = Two.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma_2\ trn \Longrightarrow \gamma_2\ trn' \Longrightarrow g_2\ trn = g_2\ trn' \Longrightarrow \varphi_1\ trn \Longrightarrow \varphi_1\ trn' \Longrightarrow f_1\ trn = f_1\ trn'$
and *T2*- γ_1 : $\bigwedge tr\ trn. One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow never\ T2\ tr \Longrightarrow T2\ trn \Longrightarrow \gamma_1\ trn$
and γ_1 -*T2*: $\bigwedge tr\ tr'\ trn\ trn'. One.O\ tr = One.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma_1\ trn \Longrightarrow \gamma_1\ trn' \Longrightarrow g_1\ trn = g_1\ trn' \Longrightarrow T2\ trn = T2\ trn'$
begin

definition $O\ tr = \text{map } g\ (\text{filter } \gamma\ tr)$

lemma $O\text{-Nil-never}$: $O\ tr = [] \iff \text{never } \gamma\ tr$ **unfolding** $O\text{-def}$ **by** (*induction tr*) *auto*

lemma $\text{Nil-}O\text{-never}$: $[] = O\ tr \iff \text{never } \gamma\ tr$ **unfolding** $O\text{-def}$ **by** (*induction tr*) *auto*

lemma $O\text{-append}$: $O\ (tr\ @\ tr') = O\ tr\ @\ O\ tr'$ **unfolding** $O\text{-def}$ **by** *auto*

lemma $\text{never-}\gamma 12\text{-never-}\gamma$: $\text{One.validFrom\ istrate}\ (tr\ @\ tr') \implies \text{never } \gamma 1\ tr' \vee \text{never } \gamma 2\ tr' \implies \text{never } \gamma\ tr'$

proof (*induction tr'*) *rule: rev-induct*

case (*snoc trn tr'*)

then show *?case using* $\gamma\text{-}\gamma 12$ [*of tr @ tr' trn*] **by** (*auto simp: One.validFrom-append*)

qed *auto*

lemma $\text{never-}\gamma 1\text{-never-}\varphi 2$: $\text{One.validFrom\ istrate}\ (tr\ @\ tr') \implies \text{never } \gamma 1\ tr' \implies \text{never } \varphi 2\ tr'$

proof (*induction tr'*) *rule: rev-induct*

case (*snoc trn tr'*)

then show *?case using* $\varphi 2\text{-}\gamma 1$ [*of tr @ tr' trn*] **by** (*auto simp: One.validFrom-append*)

qed *auto*

lemma $\text{never-}\gamma 2\text{-never-}\varphi 1$: $\text{One.validFrom\ istrate}\ (tr\ @\ tr') \implies \text{never } \gamma 2\ tr' \implies \text{never } \varphi 1\ tr'$

proof (*induction tr'*) *rule: rev-induct*

case (*snoc trn tr'*)

then show *?case using* $\varphi 1\text{-}\gamma 2$ [*of tr @ tr' trn*] **by** (*auto simp: One.validFrom-append*)

qed *auto*

lemma $\text{never-}\gamma 1\text{-never-}T2$: $\text{One.validFrom\ istrate}\ (tr\ @\ tr') \implies \text{never } T2\ tr \implies \text{never } \gamma 1\ tr' \implies \text{never } T2\ tr'$

proof (*induction tr'*) *rule: rev-induct*

case (*snoc trn tr'*)

then show *?case using* $T2\text{-}\gamma 1$ [*of tr @ tr' trn*] **by** (*auto simp: One.validFrom-append*)

qed *auto*

sublocale $\text{Abstract-BD-Security-Two-Secrets}$ $\text{One.validFrom\ istrate}\ \text{One.V}\ \text{One.O}\ B1\ \text{never } T1\ \text{Two.V}\ \text{Two.O}\ B2\ \text{never } T2\ O$

proof

fix *tr tr'*

assume $\text{One.O}\ tr = \text{One.O}\ tr'$ $\text{One.validFrom\ istrate}\ tr\ \text{One.validFrom\ istrate}\ tr'$

then show $O\ tr = O\ tr'$

proof (*induction One.O tr arbitrary: tr tr' rule: rev-induct*)

case (*Nil tr tr'*)

then have $tr = []$ **using** $\text{never-}\gamma 12\text{-never-}\gamma$ [*of [] tr*] **by** (*auto simp: O-Nil-never One.O-Nil-never*)

show $O\ tr = O\ tr'$ **using** $\text{Nil never-}\gamma 12\text{-never-}\gamma$ [*of [] tr'*] **by** (*auto simp: tr Nil-O-never One.Nil-O-never*)

```

next
  case (snoc obs obsl tr tr')
  obtain tr1 trn tr2 where tr: tr = tr1 @ [trn] @ tr2 and trn:  $\gamma 1$  trn g1 trn =
  obs
    and tr1: One.O tr1 = obsl and tr2: never  $\gamma 1$  tr2
    using snoc(2) One.O-eq-RCons[of tr obsl obs] by auto
    obtain tr1' trn' tr2' where tr': tr' = tr1' @ [trn'] @ tr2' and trn':  $\gamma 1$  trn'
  g1 trn' = obs
    and tr1': One.O tr1' = obsl and tr2': never  $\gamma 1$  tr2'
    using snoc(2,3) One.O-eq-RCons[of tr' obsl obs] by auto
    have O tr1 = O tr1' using snoc(1)[of tr1 tr1'] tr1 tr1' snoc(4,5) unfolding
  tr tr'
    by (auto simp: One.validFrom-append)
    moreover have O [trn] = O [trn'] using O1- $\gamma$ [of tr1 tr1' trn trn'] O1-g[of tr1
  tr1' trn trn']
    using snoc(4,5) tr1 tr1' trn trn' by (auto simp: tr tr' O-def One.validFrom-append
  One.validFrom-Cons)
    moreover have O tr2 = [] and O tr2' = [] using tr2 tr2'
    using never- $\gamma 12$ -never- $\gamma$ [of tr1 ## trn tr2] never- $\gamma 12$ -never- $\gamma$ [of tr1' ##
  trn' tr2']
    using snoc(4,5) unfolding tr tr' by (auto simp: O-Nil-never)
    ultimately show O tr = O tr' unfolding tr tr' O-append by auto
  qed
next
  fix tr tr'
  assume Two.O tr = Two.O tr' One.validFrom istate tr One.validFrom istate tr'
  then show O tr = O tr'
  proof (induction Two.O tr arbitrary: tr tr' rule: rev-induct)
    case (Nil tr tr')
    then have tr: O tr = [] using never- $\gamma 12$ -never- $\gamma$ [of [] tr] by (auto simp:
  O-Nil-never Two.O-Nil-never)
    show O tr = O tr' using Nil never- $\gamma 12$ -never- $\gamma$ [of [] tr'] by (auto simp: tr
  Nil-O-never Two.Nil-O-never)
  next
    case (snoc obs obsl tr tr')
    obtain tr1 trn tr2 where tr: tr = tr1 @ [trn] @ tr2 and trn:  $\gamma 2$  trn g2 trn =
    obs
      and tr1: Two.O tr1 = obsl and tr2: never  $\gamma 2$  tr2
      using snoc(2) Two.O-eq-RCons[of tr obsl obs] by auto
      obtain tr1' trn' tr2' where tr': tr' = tr1' @ [trn'] @ tr2' and trn':  $\gamma 2$  trn'
    g2 trn' = obs
      and tr1': Two.O tr1' = obsl and tr2': never  $\gamma 2$  tr2'
      using snoc(2,3) Two.O-eq-RCons[of tr' obsl obs] by auto
      have O tr1 = O tr1' using snoc(1)[of tr1 tr1'] tr1 tr1' snoc(4,5) unfolding
    tr tr'
      by (auto simp: One.validFrom-append)
      moreover have O [trn] = O [trn'] using O2- $\gamma$ [of tr1 tr1' trn trn'] O2-g[of tr1
    tr1' trn trn']
      using snoc(4,5) tr1 tr1' trn trn' by (auto simp: tr tr' O-def One.validFrom-append

```

```

One.validFrom-Cons)
  moreover have  $O\ tr2 = []$  and  $O\ tr2' = []$  using  $tr2\ tr2'$ 
    using  $never\ \gamma12\ never\ \gamma[of\ tr1\ \#\#\ trn\ tr2]\ never\ \gamma12\ never\ \gamma[of\ tr1'\ \#\#\ trn'\ tr2']$ 
    using  $snoc(4,5)$  unfolding  $tr\ tr'$  by  $(auto\ simp:\ O\ Nil\ never)$ 
    ultimately show  $O\ tr = O\ tr'$  unfolding  $tr\ tr'\ O\ append$  by auto
  qed
next
fix  $tr\ tr'$ 
assume  $One.O\ tr = One.O\ tr'\ One.validFrom\ istate\ tr\ One.validFrom\ istate\ tr'$ 
then show  $Two.V\ tr = Two.V\ tr'$ 
proof (induction  $One.O\ tr$  arbitrary:  $tr\ tr'$  rule:  $rev\ induct$ )
  case  $(Nil\ tr\ tr')$ 
  then have  $tr:\ Two.V\ tr = []$  using  $never\ \gamma1\ never\ \varphi2[of\ []\ tr]$ 
    unfolding  $Two.V\ Nil\ never\ One.Nil\ O\ never$  by auto
  show  $Two.V\ tr = Two.V\ tr'$  using  $never\ \gamma1\ never\ \varphi2[of\ []\ tr']$  using  $Nil$ 
    unfolding  $tr\ Two.Nil\ V\ never\ One.O\ Nil\ never[symmetric]$  by auto
  next
  case  $(snoc\ obs\ obsl\ tr\ tr')$ 
  obtain  $tr1\ trn\ tr2$  where  $tr:\ tr = tr1\ @\ [trn]\ @\ tr2$  and  $trn:\ \gamma1\ trn\ g1\ trn = obs$ 
    and  $tr1:\ One.O\ tr1 = obsl$  and  $tr2:\ never\ \gamma1\ tr2$ 
    using  $snoc(2)\ One.O\ eq\ RCons[of\ tr\ obsl\ obs]$  by auto
  obtain  $tr1'\ trn'\ tr2'$  where  $tr':\ tr' = tr1'\ @\ [trn']\ @\ tr2'$  and  $trn':\ \gamma1\ trn' = g1\ trn' = obs$ 
    and  $tr1':\ One.O\ tr1' = obsl$  and  $tr2':\ never\ \gamma1\ tr2'$ 
    using  $snoc(2,3)\ One.O\ eq\ RCons[of\ tr'\ obsl\ obs]$  by auto
  have  $Two.V\ tr1 = Two.V\ tr1'$  using  $snoc(1)[of\ tr1\ tr1']\ tr1\ tr1'\ snoc(4,5)$ 
  unfolding  $tr\ tr'$ 
  by  $(auto\ simp:\ One.validFrom\ append)$ 
  moreover have  $Two.V\ [trn] = Two.V\ [trn']$  using  $\gamma1\ \varphi2[of\ tr1\ tr1'\ trn\ trn']\ g1\ f2[of\ tr1\ tr1'\ trn\ trn']$ 
    using  $snoc(4,5)\ tr1\ tr1'\ trn\ trn'$  unfolding  $tr\ tr'\ Two.V\ map\ filter$ 
    by  $(auto\ simp:\ One.validFrom\ append\ One.validFrom\ Cons)$ 
  moreover have  $Two.V\ tr2 = []$  and  $Two.V\ tr2' = []$  using  $tr2\ tr2'$ 
    using  $never\ \gamma1\ never\ \varphi2[of\ tr1\ \#\#\ trn\ tr2]\ never\ \gamma1\ never\ \varphi2[of\ tr1'\ \#\#\ trn'\ tr2']$ 
    using  $snoc(4,5)$  unfolding  $tr\ tr'$  by  $(auto\ simp:\ Two.V\ Nil\ never)$ 
    ultimately show  $Two.V\ tr = Two.V\ tr'$  unfolding  $tr\ tr'\ Two.V\ append$  by auto
  qed
next
fix  $tr\ tr'$ 
assume  $Two.O\ tr = Two.O\ tr'\ One.validFrom\ istate\ tr\ One.validFrom\ istate\ tr'$ 
then show  $One.V\ tr = One.V\ tr'$ 
proof (induction  $Two.O\ tr$  arbitrary:  $tr\ tr'$  rule:  $rev\ induct$ )
  case  $(Nil\ tr\ tr')$ 
  then have  $tr:\ One.V\ tr = []$  using  $never\ \gamma2\ never\ \varphi1[of\ []\ tr]$ 
    unfolding  $One.V\ Nil\ never\ Two.Nil\ O\ never$  by auto

```

```

show One.V tr = One.V tr' using never-γ2-never-φ1[of [] tr'] using Nil
  unfolding tr One.Nil-V-never Two.O-Nil-never[symmetric] by auto
next
  case (snoc obs obsl tr tr')
  obtain tr1 trn tr2 where tr: tr = tr1 @ [trn] @ tr2 and trn: γ2 trn g2 trn =
obs
    and tr1: Two.O tr1 = obsl and tr2: never γ2 tr2
    using snoc(2) Two.O-eq-RCons[of tr obsl obs] by auto
    obtain tr1' trn' tr2' where tr': tr' = tr1' @ [trn'] @ tr2' and trn': γ2 trn'
g2 trn' = obs
      and tr1': Two.O tr1' = obsl and tr2': never γ2 tr2'
      using snoc(2,3) Two.O-eq-RCons[of tr' obsl obs] by auto
      have One.V tr1 = One.V tr1' using snoc(1)[of tr1 tr1'] tr1 tr1' snoc(4,5)
unfolding tr tr'
  by (auto simp: One.validFrom-append)
  moreover have One.V [trn] = One.V [trn'] using γ2-φ1[of tr1 tr1' trn trn']
g2-f1[of tr1 tr1' trn trn']
    using snoc(4,5) tr1 tr1' trn trn' unfolding tr tr' Two.V-map-filter
    by (auto simp: One.validFrom-append One.validFrom-Cons)
    moreover have One.V tr2 = [] and One.V tr2' = [] using tr2 tr2'
    using never-γ2-never-φ1[of tr1 ## trn tr2] never-γ2-never-φ1[of tr1' ##
trn' tr2']
    using snoc(4,5) unfolding tr tr' by (auto simp: One.V-Nil-never)
    ultimately show One.V tr = One.V tr' unfolding tr tr' One.V-append by
auto
  qed
next
  fix tr tr'
  assume One.O tr = One.O tr' One.validFrom istate tr One.validFrom istate tr'
  then show never T2 tr = never T2 tr'
  proof (induction One.O tr arbitrary: tr tr' rule: rev-induct)
  case (Nil tr tr')
  then have tr: never T2 tr using never-γ1-never-T2[of [] tr]
    unfolding Two.V-Nil-never One.Nil-O-never by auto
    then show never T2 tr = never T2 tr' using never-γ1-never-T2[of [] tr']
using Nil
  unfolding tr Two.Nil-V-never One.O-Nil-never[symmetric] by auto
next
  case (snoc obs obsl tr tr')
  obtain tr1 trn tr2 where tr: tr = tr1 @ [trn] @ tr2 and trn: γ1 trn g1 trn =
obs
    and tr1: One.O tr1 = obsl and tr2: never γ1 tr2
    using snoc(2) One.O-eq-RCons[of tr obsl obs] by auto
    obtain tr1' trn' tr2' where tr': tr' = tr1' @ [trn'] @ tr2' and trn': γ1 trn'
g1 trn' = obs
      and tr1': One.O tr1' = obsl and tr2': never γ1 tr2'
      using snoc(2,3) One.O-eq-RCons[of tr' obsl obs] by auto
      have never T2 tr1 = never T2 tr1' using snoc(1)[of tr1 tr1'] tr1 tr1' snoc(4,5)
unfolding tr tr'

```

```

    by (auto simp: One.validFrom-append)
  moreover have  $T2\ trn = T2\ trn'$  using  $\gamma 1$ -T2[of tr1 tr1' trn trn']
    using snoc(4,5) tr1 tr1' trn trn' unfolding tr tr' Two.V-map-filter
    by (auto simp: One.validFrom-append One.validFrom-Cons)
  moreover have never T2 (tr1 ## trn)  $\longrightarrow$  never T2 tr2
    and never T2 (tr1' ## trn')  $\longrightarrow$  never T2 tr2'
    using never- $\gamma 1$ -never-T2[of tr1 ## trn tr2] never- $\gamma 1$ -never-T2[of tr1' ##
trn' tr2']
    using tr2 tr2' snoc(4,5) unfolding tr tr' by (auto simp: Two.V-Nil-never)
  ultimately show never T2 tr = never T2 tr' unfolding tr tr' by auto
qed
qed
end
end
end

```

References

- [1] T. Bauerei, A. Pesenti Gritti, A. Popescu, and F. Raimondi. Cosmed: A confidentiality-verified social media platform. In J. C. Blanchette and S. Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2016.
- [2] T. Bauerei, A. Pesenti Gritti, A. Popescu, and F. Raimondi. Cosmedis: A distributed social media platform with formally verified confidentiality guarantees. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 729–748. IEEE Computer Society, 2017.
- [3] T. Bauerei, A. Pesenti Gritti, A. Popescu, and F. Raimondi. Cosmed: A confidentiality-verified social media platform. *J. Autom. Reason.*, 61(1-4):113–139, 2018.
- [4] T. Bauereiss and A. Popescu. CoSMed: A confidentiality-verified social media platform. In M. Eberl, G. Klein, A. Lochbihler, T. Nipkow, L. Paulson, and R. Thiemann, editors, *Archive of Formal Proofs*, 2021.
- [5] T. Bauereiss and A. Popescu. CoSMedis: A confidentiality-verified distributed social media platform. In M. Eberl, G. Klein, A. Lochbihler, T. Nipkow, L. Paulson, and R. Thiemann, editors, *Archive of Formal Proofs*, 2021.
- [6] S. Kanav, P. Lammich, and A. Popescu. A conference management system with verified document confidentiality. In A. Biere and R. Bloem,

editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 2014.

- [7] 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.
- [8] A. Popescu, P. Lammich, and T. Bauereiss. Bounded-deducibility security. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*, 2014.
- [9] A. Popescu, P. Lammich, and T. Bauereiss. CoCon: A confidentiality-verified conference management system. In M. Eberl, G. Klein, A. Lochbihler, T. Nipkow, L. Paulson, and R. Thiemann, editors, *Archive of Formal Proofs*, 2021.
- [10] A. Popescu, P. Lammich, and P. Hou. Cocon: A conference management system with formally verified document confidentiality. *J. Autom. Reason.*, 65(2):321–356, 2021.