Compositional BD Security

Thomas Bauereiss Andrei Popescu

March 19, 2025

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*: $\bigwedge x\ xs\ ys.\ P\ xs\ ys \Longrightarrow P\ (x\ \#\ xs)\ ys$
and *NC*: $\bigwedge xs\ y\ ys.\ P\ xs\ ys \Longrightarrow P\ xs\ (y\ \#\ ys)$
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*: $\bigwedge x\ xs.\ P\ xs\ [] \Longrightarrow P\ (x\ \#\ xs)\ []$
and *NC*: $\bigwedge y\ ys.\ P\ []\ ys \Longrightarrow P\ []\ (y\ \#\ ys)$
and *CC*: $\bigwedge x\ xs\ y\ ys.$
  $P\ xs\ ys \Longrightarrow$
  $(\bigwedge\ ys'.\ length\ ys' \leq Suc\ (length\ ys) \Longrightarrow P\ xs\ ys') \Longrightarrow$
  $(\bigwedge\ xs'.\ length\ xs' \leq Suc\ (length\ xs) \Longrightarrow P\ xs'\ ys) \Longrightarrow$
  $P\ (x\ \#\ xs)\ (y\ \#\ ys)$
shows $P\ xs\ ys$
proof (*induction rule*: *measure-induct2*[*of* $\lambda xs\ ys.\ length\ xs + length\ ys$, *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:
⋀ *tr*. *validSystemTraces tr* ⟹
(∃ *tr1 tr2*. *validSystemTraces1 tr1* ∧ *validSystemTraces2 tr2* ∧ *comp tr1 tr2 tr*)
and
V-comp:
⋀ *tr1 tr2 tr*.

$validSystemTraces1\ tr1 \Longrightarrow validSystemTraces2\ tr2 \Longrightarrow comp\ tr1\ tr2\ tr$
$\Longrightarrow compV\ (V1\ tr1)\ (V2\ tr2)\ (V\ tr)$
and
O-comp:
$\bigwedge\ tr1\ tr2\ tr.$
$validSystemTraces1\ tr1 \Longrightarrow validSystemTraces2\ tr2 \Longrightarrow comp\ tr1\ tr2\ tr$
$\Longrightarrow compO\ (O1\ tr1)\ (O2\ tr2)\ (O\ tr)$
and
TT-comp:
$\bigwedge\ tr1\ tr2\ tr.$
$validSystemTraces1\ tr1 \Longrightarrow validSystemTraces2\ tr2 \Longrightarrow comp\ tr1\ tr2\ tr$
$\Longrightarrow TT\ tr \Longrightarrow TT1\ tr1 \wedge TT2\ tr2$
and
B-comp:
$\bigwedge\ vl1\ vl2\ vl\ vl'.$
$compV\ vl1\ vl2\ vl \Longrightarrow B\ vl\ vl'$
$\Longrightarrow \exists\ vl1'\ vl2'.\ compV\ vl1'\ vl2'\ vl' \wedge B1\ vl1\ vl1' \wedge B2\ vl2\ vl2'$
and
O-V-comp:
$\bigwedge\ tr1\ tr2\ vl\ ol.$
$validSystemTraces1\ tr1 \Longrightarrow validSystemTraces2\ tr2 \Longrightarrow$
$compV\ (V1\ tr1)\ (V2\ tr2)\ vl \Longrightarrow compO\ (O1\ tr1)\ (O2\ tr2)\ ol$
$\Longrightarrow \exists\ tr.\ validSystemTraces\ tr \wedge O\ tr = ol \wedge V\ tr = vl$
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 $tr1'$ **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 $tr2'$ **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'*. *validSystemTraces tr'* $\wedge$ *O tr'* = *O tr* $\wedge$ *V tr'* = *vl'* **by** *auto*
qed

end

type-synonym (*'state1*,*'state2*) *cstate* = *'state1* × *'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* φ*1 f1* γ*1 g1 T1 B1* +
Two: *BD-Security-TS istate2 validTrans2 srcOf2 tgtOf2* φ*2 f2* γ*2 g2 T2 B2*
for
istate1 :: *'state1* **and** *validTrans1* :: *'trans1* ⇒ *bool*
and
srcOf1 :: *'trans1* ⇒ *'state1* **and** *tgtOf1* :: *'trans1* ⇒ *'state1*
and
φ*1* :: *'trans1* ⇒ *bool* **and** *f1* :: *'trans1* ⇒ *'value1*
and
γ*1* :: *'trans1* ⇒ *bool* **and** *g1* :: *'trans1* ⇒ *'obs1*
and
T1 :: *'trans1* ⇒ *bool* **and** *B1* :: *'value1 list* ⇒ *'value1 list* ⇒ *bool*
and

istate2 :: *'state2* **and** *validTrans2* :: *'trans2* ⇒ *bool*
and
srcOf2 :: *'trans2* ⇒ *'state2* **and** *tgtOf2* :: *'trans2* ⇒ *'state2*
and
φ*2* :: *'trans2* ⇒ *bool* **and** *f2* :: *'trans2* ⇒ *'value2*
and
γ*2* :: *'trans2* ⇒ *bool* **and** *g2* :: *'trans2* ⇒ *'obs2*
and
T2 :: *'trans2* ⇒ *bool* **and** *B2* :: *'value2 list* ⇒ *'value2 list* ⇒ *bool*
+
fixes
isCom1 :: *'trans1* ⇒ *bool* **and** *isCom2* :: *'trans2* ⇒ *bool*
and
sync :: *'trans1* ⇒ *'trans2* ⇒ *bool*
and
isComV1 :: *'value1* ⇒ *bool* **and** *isComV2* :: *'value2* ⇒ *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
$isCom1\text{-}isComV1$: $\bigwedge trn1.\ validTrans1\ trn1 \Longrightarrow One.reach\ (srcOf1\ trn1) \Longrightarrow \varphi 1\ trn1 \Longrightarrow isCom1\ trn1 \longleftrightarrow isComV1\ (f1\ trn1)$
and
$isCom1\text{-}isComO1$: $\bigwedge trn1.\ validTrans1\ trn1 \Longrightarrow One.reach\ (srcOf1\ trn1) \Longrightarrow \gamma 1\ trn1 \Longrightarrow isCom1\ trn1 \longleftrightarrow isComO1\ (g1\ trn1)$
and
$isCom2\text{-}isComV2$: $\bigwedge trn2.\ validTrans2\ trn2 \Longrightarrow Two.reach\ (srcOf2\ trn2) \Longrightarrow \varphi 2\ trn2 \Longrightarrow isCom2\ trn2 \longleftrightarrow isComV2\ (f2\ trn2)$
and
$isCom2\text{-}isComO2$: $\bigwedge trn2.\ validTrans2\ trn2 \Longrightarrow Two.reach\ (srcOf2\ trn2) \Longrightarrow \gamma 2\ trn2 \Longrightarrow isCom2\ trn2 \longleftrightarrow isComO2\ (g2\ trn2)$
and
$sync\text{-}syncV$:
$\bigwedge trn1\ trn2.$
$validTrans1\ trn1 \Longrightarrow One.reach\ (srcOf1\ trn1) \Longrightarrow$
$validTrans2\ trn2 \Longrightarrow Two.reach\ (srcOf2\ trn2) \Longrightarrow$
$isCom1\ trn1 \Longrightarrow isCom2\ trn2 \Longrightarrow \varphi 1\ trn1 \Longrightarrow \varphi 2\ trn2 \Longrightarrow$
$sync\ trn1\ trn2 \Longrightarrow syncV\ (f1\ trn1)\ (f2\ trn2)$
and
$sync\text{-}syncO$:
$\bigwedge trn1\ trn2.$
$validTrans1\ trn1 \Longrightarrow One.reach\ (srcOf1\ trn1) \Longrightarrow$
$validTrans2\ trn2 \Longrightarrow Two.reach\ (srcOf2\ trn2) \Longrightarrow$
$isCom1\ trn1 \Longrightarrow isCom2\ trn2 \Longrightarrow \gamma 1\ trn1 \Longrightarrow \gamma 2\ trn2 \Longrightarrow$
$sync\ trn1\ trn2 \Longrightarrow syncO\ (g1\ trn1)\ (g2\ trn2)$
and
$sync\text{-}\varphi 1\text{-}\varphi 2$:
$\bigwedge trn1\ trn2.$
$validTrans1\ trn1 \Longrightarrow One.reach\ (srcOf1\ trn1) \Longrightarrow$
$validTrans2\ trn2 \Longrightarrow Two.reach\ (srcOf2\ trn2) \Longrightarrow$
$isCom1\ trn1 \Longrightarrow isCom2\ trn2 \Longrightarrow$
$sync\ trn1\ trn2 \Longrightarrow \varphi 1\ trn1 \longleftrightarrow \varphi 2\ trn2$
and
$sync\text{-}\varphi\text{-}\gamma$:
$\bigwedge trn1\ trn2.$
$validTrans1\ trn1 \Longrightarrow One.reach\ (srcOf1\ trn1) \Longrightarrow$
$validTrans2\ trn2 \Longrightarrow Two.reach\ (srcOf2\ trn2) \Longrightarrow$
$isCom1\ trn1 \Longrightarrow isCom2\ trn2 \Longrightarrow$
$\gamma 1\ trn1 \Longrightarrow \gamma 2\ trn2 \Longrightarrow$
$syncO\ (g1\ trn1)\ (g2\ trn2) \Longrightarrow$
$(\varphi 1\ trn1 \Longrightarrow \varphi 2\ trn2 \Longrightarrow syncV\ (f1\ trn1)\ (f2\ trn2))$
$\Longrightarrow$

sync trn1 trn2
and
isCom1-γ1: $\bigwedge$ *trn1*. *validTrans1 trn1* $\Longrightarrow$ *One.reach* (*srcOf1 trn1*) $\Longrightarrow$ *isCom1 trn1* $\Longrightarrow$ *γ1 trn1*
and
isCom2-γ2: $\bigwedge$ *trn2*. *validTrans2 trn2* $\Longrightarrow$ *Two.reach* (*srcOf2 trn2*) $\Longrightarrow$ *isCom2 trn2* $\Longrightarrow$ *γ2 trn2*
and
isCom2-V2: $\bigwedge$ *trn2*. *validTrans2 trn2* $\Longrightarrow$ *Two.reach* (*srcOf2 trn2*) $\Longrightarrow$ *φ2 trn2* $\Longrightarrow$ *isCom2 trn2*
and
Dummy: *istate1* = *istate1* ∧ *srcOf1* = *srcOf1* ∧ *tgtOf1* = *tgtOf1* ∧ *T1* = *T1* ∧ *B1* = *B1* ∧
istate2 = *istate2* ∧ *srcOf2* = *srcOf2* ∧ *tgtOf2* = *tgtOf2* ∧ *T2* = *T2* ∧ *B2* = *B2*
begin

lemma *sync-γ1-γ2*:
$\bigwedge$ *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 $\Longrightarrow$ *γ1 trn1* $\longleftrightarrow$ *γ2 trn2*
using *isCom1-γ1 isCom2-γ2*
by *auto*

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

fun *validTrans* :: (*'state1*, *'trans1*, *'state2*, *'trans2*) *ctrans* ⇒ *bool* **where**
validTrans(*Trans1 s2 trn1*) = (*validTrans1 trn1* ∧ ¬ *isCom1 trn1*)
|*validTrans* (*Trans2 s1 trn2*) = (*validTrans2 trn2* ∧ ¬ *isCom2 trn2*)
|*validTrans* (*CTrans trn1 trn2*) =
(*validTrans1 trn1* ∧ *validTrans2 trn2* ∧ *isCom1 trn1* ∧ *isCom2 trn2* ∧ *sync trn1 trn2*)

fun *srcOf* :: (*'state1*, *'trans1*, *'state2*, *'trans2*) *ctrans* ⇒ *'state1* × *'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* ⇒ *'state1* × *'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 *φ* :: (*'state1*, *'trans1*, *'state2*, *'trans2*) *ctrans* ⇒ *bool* **where**
φ (*Trans1 s2 trn1*) = *φ1 trn1*
|*φ* (*Trans2 s1 trn2*) = *φ2 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 $compV\text{-}cases\text{-}V[consumes\ 3,\ case\text{-}names\ Nil\ Step1\ Com]$:
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 $Step1$:
$\bigwedge vll1\ vll2\ vll\ v1.$
$vl1 = v1\ \#\ vll1 \Longrightarrow$
$Two.V\ tr2 = vll2 \Longrightarrow$
$vl = Value1\ v1\ \#\ vll \Longrightarrow$

compV vll1 vll2 vll $\Longrightarrow \neg$ *isComV1 v1* $\Longrightarrow$ *P*
and *Com*:
$\bigwedge$*vll1 vll2 vll v1 v2*.
vl1 = *v1* # *vll1* $\Longrightarrow$
Two.V tr2 = *v2* # *vll2* $\Longrightarrow$
vl = *CValue v1 v2* # *vll* $\Longrightarrow$
compV vll1 vll2 vll $\Longrightarrow$
isComV1 v1 $\Longrightarrow$ *isComV2 v2* $\Longrightarrow$ *syncV v1 v2* $\Longrightarrow$ *P*
shows *P*
using *c* **proof** *cases*
case (*Step2 vll2 vll1 v2*)
obtain *tr2a trn2 tr2b* **where** *tr2*: *tr2* = *tr2a* @ *trn2* # *tr2b* **and**
φ*2*: φ*2 trn2* **and** *f2*: *f2 trn2* = *v2*
using ‹*Two.V tr2* = *v2* # *vll2*› **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'* φ*2*] ‹$\neg$ *isComV2 v2*›
using φ*2 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 $\Longrightarrow \neg$ *isComO1 o1*
$\Longrightarrow$ *compO* (*o1* # *ol1*) *ol2* (*Obs1 o1* # *ol*)
|*Step2*[*intro*]:
compO ol1 ol2 ol $\Longrightarrow \neg$ *isComO2 o2*
$\Longrightarrow$ *compO ol1* (*o2* # *ol2*) (*Obs2 o2* # *ol*)
|*Com*[*intro*]:
compO ol1 ol2 ol $\Longrightarrow$ *isComO1 o1* $\Longrightarrow$ *isComO2 o2* $\Longrightarrow$ *syncO o1 o2*
$\Longrightarrow$ *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* $\Longrightarrow \neg$ *isCom1 trn1* $\Longrightarrow$
 ccomp (*srcOf1 trn1*) *s2* (*trn1* # *tr1*) *tr2* (*Trans1 s2 trn1* # *tr*)
|
Step2[*intro*]:
ccomp s1 (*tgtOf2 trn2*) *tr1 tr2 tr* $\Longrightarrow \neg$ *isCom2 trn2* $\Longrightarrow$
 ccomp s1 (*srcOf2 trn2*) *tr1* (*trn2* # *tr2*) (*Trans2 s1 trn2* # *tr*)
|
Com[*intro*]:
ccomp (*tgtOf1 trn1*) (*tgtOf2 trn2*) *tr1 tr2 tr* $\Longrightarrow$
 isCom1 trn1 $\Longrightarrow$ *isCom2 trn2* $\Longrightarrow$ *sync trn1 trn2* $\Longrightarrow$
 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* φ *f* γ *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.prems 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.prems 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.prems 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 *∃ tr1 tr2. One.validFrom istate1 tr1 ∧ Two.validFrom istate2 tr2 ∧ comp tr1 tr2 tr*
using *assms valid* **unfolding** *comp-def icstate-def validFrom-def* **by**(*cases tr*) *fastforce+*

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** *icstate-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 φ1 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 φ2 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 φ1 trn1; cases φ2 trn2; simp add: isCom1-isComV1 isCom2-isComV2)
      (use sync-φ1-φ2 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* γ*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* γ*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* γ*1 trn1*; *cases* γ*2 trn2*; *simp add*: *isCom1-isComO1 isCom2-isComO2*)
(*use sync-*γ*1-*γ*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.prems* **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 *γ1*: *γ1 trn1* **using** *trn1 isCom1-γ1 s1* **by** *auto*
hence *isComO1* (*g1 trn1*) **using** *γ1 com1 s1 isCom1-isComO1 trn1* **by** *blast*
hence *False* **using** ‹*compO* (*One.O* (*trn1* # *tr1*)) (*Two.O* []) *obl*›
using *γ1* **by** (*auto elim*: *compO.cases*)
thus *?thesis* **by** *simp*
next
case *False* **note** *com1* = *False*
show *?thesis* **proof**(*cases φ1 trn1*)
case *True* **note** *φ1* = *True*
hence *comv1*: ¬ *isComV1* (*f1 trn1*) **using** *φ1 com1 isCom1-isComV1 trn1 s1* **by** *blast*
with ‹*compV* (*One.V* (*trn1* # *tr1*)) (*Two.V* []) *vl*› *φ1*
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 γ1 trn1*)
case *True* **note** *γ1* = *True*
hence ¬ *isComO1* (*g1 trn1*) **using** *γ1 com1 isCom1-isComO1 trn1 s1* **by** *blast*
with ‹*compO* (*One.O* (*trn1* # *tr1*)) (*Two.O* []) *obl*› *γ1*
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* ∧ *V trr* = *vll* **by** *auto*
thus *?thesis*
unfolding *obl vl* **using** *trn1 com1 s1 φ1 γ1*
by (*intro exI*[*of* - *Trans1 s2 trn1* # *trr*]) *auto*
next
case *False* **note** *γ1* = *False*
note *obl* = ‹*compO* (*One.O* (*trn1* # *tr1*)) (*Two.O* []) *obl*›
from *ConsNil.IH*[*OF tr1* - *rs1' rs2 vll*] *obl γ1* **obtain** *trr* **where**
validFrom (*?s1*, *s2*) *trr* **and** *O trr* = *obl* ∧ *V trr* = *vll* **by** *auto*
thus *?thesis*
unfolding *obl vl* **using** *trn1 com1 s1 φ1 γ1*
by (*intro exI*[*of* - *Trans1 s2 trn1* # *trr*]) *auto*
qed
next
case *False* **note** *φ1* = *False*
note *vl* = ‹*compV* (*One.V* (*trn1* # *tr1*)) (*Two.V* []) *vl*›
show *?thesis* **proof**(*cases γ1 trn1*)

case *True* **note** *γ1 = True*
hence ¬ *isComO1* (*g1 trn1*) **using** *γ1 com1 isCom1-isComO1 trn1 s1* **by** *blast*
with ‹*compO* (*One.O* (*trn1* # *tr1*)) (*Two.O* []) *obl*› *γ1*
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 - obll*] *vl φ1* **obtain** *trr* **where**
validFrom (*?s1*, *s2*) *trr* **and** *O trr = obll* ∧ *V trr = vl* **by** *auto*
thus *?thesis*
unfolding *obl vl* **using** *trn1 com1 s1 φ1 γ1*
by (*intro exI*[*of - Trans1 s2 trn1* # *trr*]) *auto*
next
case *False* **note** *γ1 = False*
note *obl* = ‹*compO* (*One.O* (*trn1* # *tr1*)) (*Two.O* []) *obl*›
from *ConsNil.IH*[*OF tr1 - rs1′ rs2 -*] *vl φ1 obl γ1* **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 φ1 γ1*
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** *NilCons.prems* **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 *γ2*: *γ2 trn2* **using** *trn2 isCom2-γ2 s2* **by** *auto*
hence *isComO2* (*g2 trn2*) **using** *γ2 com2 isCom2-isComO2 trn2 s2* **by** *blast*
hence *False* **using** ‹*compO* (*One.O* []) (*Two.O* (*trn2* # *tr2*)) *obl*›
using *γ2* **by** (*auto elim*: *compO.cases*)
thus *?thesis* **by** *simp*
next
case *False* **note** *com2 = False*
show *?thesis* **proof**(*cases φ2 trn2*)
case *True* **note** *φ2 = True*
hence *comv1*: ¬ *isComV2* (*f2 trn2*) **using** *φ2 com2 isCom2-isComV2 trn2 s2* **by** *blast*
with ‹*compV* (*One.V* []) (*Two.V* (*trn2* # *tr2*)) *vl*› *φ2*
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 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*› $\gamma 2$
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* $\varphi 2$ $\gamma 2$
by (*intro exI*[*of - Trans2 s1 trn2* # *trr*]) *auto*
next
case *False* **note** $\gamma 2$ = *False*
note *obl* = ‹*compO* (*One.O* []) (*Two.O* (*trn2* # *tr2*)) *obl*›
from *NilCons.IH*[*OF - tr2 rs1 rs2′ vll*] *obl* $\gamma 2$ **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* $\varphi 2$ $\gamma 2$
by (*intro exI*[*of - Trans2 s1 trn2* # *trr*]) *auto*
qed
next
case *False* **note** $\varphi 2$ = *False*
note *vl* = ‹*compV* (*One.V* []) (*Two.V* (*trn2* # *tr2*)) *vl*›
show *?thesis* **proof**(*cases* $\gamma 2$ *trn2*)
case *True* **note** $\gamma 2$ = *True*
hence ¬ *isComO2* (*g2 trn2*) **using** $\gamma 2$ *com2 isCom2-isComO2 trn2 s2* **by**
blast
with ‹*compO* (*One.O* []) (*Two.O* (*trn2* # *tr2*)) *obl*› $\gamma 2$
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* $\varphi 2$ **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* $\varphi 2$ $\gamma 2$
by (*intro exI*[*of - Trans2 s1 trn2* # *trr*]) *auto*
next
case *False* **note** $\gamma 2$ = *False*
note *obl* = ‹*compO* (*One.O* []) (*Two.O* (*trn2* # *tr2*)) *obl*›
from *NilCons.IH*[*OF - tr2 rs1 rs2′ -*] *vl* $\varphi 2$ *obl* $\gamma 2$ **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* $\varphi 2$ $\gamma 2$
by (*intro exI*[*of - Trans2 s1 trn2* # *trr*]) *auto*
qed
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.prems 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 *φ12* **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* ∧ *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*
qed
next
case (*Step1 obll1 obll ob1*) **note** *Step1O* = *Step1*
show *?thesis* **proof**(*cases γ1 trn1*)
case *True* **note** *γ1* = *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*: ¬ *isCom1 trn1* **using** *Step1O γ1 g1 isCom1-isComO1 trn1*
s1 **by** *blast*
show *?thesis* **using** *trr2 vl s2*(*2*) **proof**(*cases rule*: *compV-cases-V*)
case *Nil*
hence *φ12*: ¬ *φ1 trn1* ∧ ¬ *φ2 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* ∧ *V trr* =
vl
by *auto*
thus *?thesis* **using** *trn1 Step1O g1 φ12 γ1 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 φ1 trn1*)
case *False* **note** *φ1* = *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* ∧ *V trr*
= *vl*
by *auto*
thus *?thesis* **using** *trn1 Step1O g1 φ1 γ1 trn1 s1 isCom1-isComO1*
by (*intro exI*[*of - Trans1 s2 trn1 # trr*]) *auto*
next
case *True* **note** *φ1* = *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* ∧ *V trr*
= *vll*

```
            by auto
              thus ?thesis using trn1 Step1O Step1V f1 φ1 g1 φ1 γ1 trn1 s1
isCom1-isComO1
            by (intro exI[of - Trans1 s2 trn1 # trr]) auto
           qed
          next
           case (Com vll1 vll2 vll v1 v2)
            hence φ1: ¬ φ1 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 obll]
           obtain trr where validFrom (?s1, s2) trr and O trr = obll ∧ V trr =
vl
           by auto
           thus ?thesis using trn1 Step1O g1 φ1 γ1 trn1 s1 isCom1-isComO1
           by (intro exI[of - Trans1 s2 trn1 # trr]) auto
          qed
         next
          case False note γ1 = False
          hence obl: compO (One.O tr1) (Two.O ?tr2) obl using obl by simp
          hence φ1: φ1 trn1 and com1: ¬ isCom1 trn1 using φγ1 γ1 isCom1-γ1
trn1 s1 by auto
          show ?thesis using trr2 vl s2(2) proof(cases rule: compV-cases-V)
           case Nil hence False using φ1 by auto
           thus ?thesis by simp
          next
           case Com hence False using φ1 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 φ1 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 ∧ V trr =
vll
           by auto
             thus ?thesis using trn1 Step1O Step1V f1 φ1 φ1 γ1 trn1 s1 is-
Com1-isComV1
           by(intro exI[of - Trans1 s2 trn1 # trr]) auto
          qed
         qed
        next
         case (Step2 obl2 obll ob2) note Step2O = Step2
        hence com2: ¬ isCom2 trn2 using isCom2-γ2[OF trn2] isCom2-isComO2[OF
trn2] s2 by auto
         hence φ2: ¬ φ2 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*: *g2 trn2* = *ob2* **and** *obl2* = *Two.O tr2* **using** *Step2* **by** *auto*
hence *obll*: *compO* (*One.O ?tr1*) (*Two.O tr2*) *obll* **using** *Step2* **by** *auto*
from *ConsCons.IH*(*3*)[*OF - trr1 tr2 s1*(*2*) *rs2' vl obll*]
obtain *trr* **where** *validFrom* (*s1*, *?s2*) *trr* **and** *O trr* = *obll* ∧ *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 obll 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 *obll*: *compO* (*One.O tr1*) (*Two.O tr2*) *obll* **using** *obl ComO* $\gamma 1$ $\gamma 2$
by *auto*
have *g1*: *g1 trn1* = *ob1* **and** *obll1* = *One.O tr1* **and**
g2: *g2 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*-φ-γ[*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*) ∧ *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* ∧ $\varphi 2$ *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 *φ12* **by** *auto*
hence *vll*: *compV* (*One.V tr1*) (*Two.V tr2*) *vll* **using** *ComV com1 com2 φ12* **by** *auto*
from *ConsCons.IH*(*1*)[*OF tr1 tr2 rs1′ rs2′ vll obll*]
obtain *trr* **where** *validFrom* (*?s1*, *?s2*) *trr* **and** *O trr* = *obll* ∧ *V trr* = *vll*
by *auto*
thus *?thesis* **using** *trn1 trn2 ComO ComV f1 f2 φ12 g1 g2 γ1 γ2 com1 com2 sync s1 s2*
by (*intro exI*[*of - CTrans trn1 trn2 # trr*]) *auto*
qed
next
case *False*
hence *φ12*: ¬ *φ1 trn1* ∧ ¬ *φ2 trn2* **using** *sync-φ1-φ2*[*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 obll*]
obtain *trr* **where** *validFrom* (*?s1*, *?s2*) *trr* **and** *O trr* = *obll* ∧ *V trr* = *vl* **by** *auto*
thus *?thesis* **using** *trn1 trn2 ComO φ12 g1 g2 γ1 γ2 com1 com2 sync s1 s2*
by (*intro exI*[*of - CTrans trn1 trn2 # trr*]) *auto*
qed
next
case *False*
hence *φ2*: *φ2 trn2* **and** *com2*: ¬ *isCom2 trn2* **using** *φγ2 isCom2-γ2 trn2 s2* **by** *auto*
have *False* **using** *trr2 vl s2*(*2*) *φ2 com2 isCom2-V2*[*OF trn2*] *s2* **by** (*cases rule*: *compV-cases-V*) *auto*
thus *?thesis* **by** *simp*
qed
next
case *False* **note** *γ1* = *False*
hence *obl*: *compO* (*One.O tr1*) (*Two.O ?tr2*) *obl* **using** *obl* **by** *simp*
have *φ1*: *φ1 trn1* **and** *com1*: ¬ *isCom1 trn1* **using** *γ1 φγ1 isCom1-γ1 trn1 s1* **by** *auto*
show *?thesis* **using** *trr2 vl s2*(*2*) **proof**(*cases rule*: *compV-cases-V*)
case *Nil* **hence** *False* **using** *φ1* **by** *auto*
thus *?thesis* **by** *simp*
next
case *Com* **hence** *False* **using** *com1 φ1 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** *φ1* **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* ∧ *V trr* = *vll*
by *auto*
thus *?thesis* **using** *trn1 Step1V f1 φ1 γ1 trn1 s1 isCom1-isComO1 com1*
by (*intro exI*[*of - Trans1 s2 trn1 # trr*]) *auto*
qed
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 ∃ *tr*. *validFrom icstate tr* ∧ *O tr* = *ol* ∧ *V tr* = *vl*
using *assms pullback-O-V-aux One.reach.Istate Two.reach.Istate* **unfolding** *icstate-def* **by** *auto*

end

sublocale *BD-Security-TS-Comp* ⊆ *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* $\Longrightarrow$ *secure2* $\Longrightarrow$ *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* $\Longrightarrow$ *validSystemTrace tr* $\Longrightarrow$ *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* $\Longrightarrow$ *O tr* = *ol*
and $\bigwedge$*tr vl vl1*. *B* (*V tr*) *vl1* $\Longrightarrow$ *validSystemTrace tr* $\Longrightarrow$ ($\exists$ *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 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 ≡ (∀ *obs*. *OV-compatible obs vl* ⟶ *OV-compatible obs vl1*)

definition *validObs* :: *'observations* ⇒ *bool* **where**
validObs obs ≡ (∃ *tr*. *validSystemTrace tr* ∧ *O tr* = *obs*)

definition *validVal* :: *'values* ⇒ *bool* **where**
validVal vl ≡ (∃ *tr*. *validSystemTrace tr* ∧ *V tr* = *vl*)

lemma *OV-total-secure*:
assumes *OV*: ⋀*obs vl*. *validObs obs* ⟹ *validVal vl* ⟹ *OV-compatible obs vl*
⟹ (∃ *tr*. *validSystemTrace tr* ∧ *O tr* = *obs* ∧ *V tr* = *vl*)
and *BV*: ⋀*vl vl1*. *B vl vl1* ⟹ *validVal vl* ⟹ *V-compatible vl vl1* ∧ *validVal vl1*
shows *secure*
unfolding *secure-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-compatible* (*O tr*) (*V tr*)
unfolding *validObs-def validVal-def OV-compatible-def* **by** *blast+*
moreover then have *validVal vl1* **and** *OV-compatible* (*O tr*) *vl1*
using *B BV* **unfolding** *V-compatible-def vl* **by** *blast+*
ultimately show ∃ *tr1*. *validSystemTrace tr1* ∧ *O tr1* = *O tr* ∧ *V tr1* = *vl1*
using *OV* **by** *blast*
qed

lemma *unconstrained-secure*:
assumes ⋀*tr*. *validSystemTrace tr*
and *BV*: ⋀*vl vl1*. *B vl vl1* ⟹ *validVal vl* ⟹ *V-compatible vl vl1* ∧ *validVal vl1*
shows *secure*
using *assms* **by** (*intro OV-total-secure*) (*auto simp*: *OV-compatible-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-Security*
imports *Bounded-Deducibility-Security.BD-Security-TS*
begin

locale *Abstract-BD-Security-Trans* =
Orig: *Abstract-BD-Security validSystemTrace V O B TT*
+ *Prime*: *Abstract-BD-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
$vST\text{-}vST'$: $validSystemTrace\ tr \Longrightarrow validSystemTrace'\ (translateTrace\ tr)$
and
$vST'\text{-}vST$: $validSystemTrace'\ tr' \Longrightarrow (\exists\, tr.\ validSystemTrace\ tr \wedge translateTrace\ tr = tr')$
and
$V'\text{-}V$: $validSystemTrace\ tr \Longrightarrow V'\ (translateTrace\ tr) = translateVal\ (V\ tr)$
and
$O'\text{-}O$: $validSystemTrace\ tr \Longrightarrow O'\ (translateTrace\ tr) = translateObs\ (O\ tr)$
and
$B'\text{-}B$: $B'\ vl'\ vl1' \Longrightarrow validSystemTrace\ tr \Longrightarrow TT\ tr \Longrightarrow translateVal\ (V\ tr) = vl'$
$\Longrightarrow (\exists\, vl1.\ translateVal\ vl1 = vl1' \wedge B\ (V\ tr)\ vl1)$
and
$TT'\text{-}TT$: $TT'\ (translateTrace\ tr) \Longrightarrow validSystemTrace\ tr \Longrightarrow TT\ tr$
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'*. *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* φ *f* γ *g T B*
+ *Prime?*: *BD-Security-TS istate' validTrans' srcOf' tgtOf'* φ' *f'* γ' *g' T' B'*
for *istate* :: *'state* **and** *validTrans* :: *'trans* $\Rightarrow$ *bool*
and *srcOf* :: *'trans* $\Rightarrow$ *'state* **and** *tgtOf* :: *'trans* $\Rightarrow$ *'state*
and φ :: *'trans* $\Rightarrow$ *bool* **and** *f* :: *'trans* $\Rightarrow$ *'val*
and γ :: *'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 φ' :: *'trans'* $\Rightarrow$ *bool* **and** *f'* :: *'trans'* $\Rightarrow$ *'val'*
and γ' :: *'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
  γ′-γ: $validTrans\ trn \Longrightarrow Orig.reach\ (srcOf\ trn) \Longrightarrow \gamma'\ (translateTrans\ trn) \Longrightarrow \gamma\ trn \wedge translateObs\ (g\ trn) = Some\ (g'\ (translateTrans\ trn))$
and
  γ-γ′: $validTrans\ trn \Longrightarrow Orig.reach\ (srcOf\ trn) \Longrightarrow \gamma\ trn \Longrightarrow \gamma'\ (translateTrans\ trn) \vee translateObs\ (g\ trn) = None$
and
  φ′-φ: $validTrans\ trn \Longrightarrow Orig.reach\ (srcOf\ trn) \Longrightarrow \varphi'\ (translateTrans\ trn) \Longrightarrow \varphi\ trn \wedge translateVal\ (f\ trn) = Some\ (f'\ (translateTrans\ trn))$
and
  φ-φ′: $validTrans\ trn \Longrightarrow Orig.reach\ (srcOf\ trn) \Longrightarrow \varphi\ trn \Longrightarrow \varphi'\ (translateTrans\ trn) \vee translateVal\ (f\ trn) = None$
and
  T-T′: $T\ trn \Longrightarrow validTrans\ trn \Longrightarrow Orig.reach\ (srcOf\ trn) \Longrightarrow T'\ (translateTrans\ trn)$
and
  B′-B: $B'\ vl'\ vl1' \Longrightarrow Orig.validFrom\ istate\ tr \Longrightarrow never\ T\ tr \Longrightarrow these\ (map\ translateVal\ (Orig.V\ tr)) = vl'$
    $\Longrightarrow (\exists\, vl1.\ these\ (map\ translateVal\ vl1) = vl1' \wedge B\ (Orig.V\ tr)\ vl1)$
begin

definition *translateTrace* :: *′trans list ⇒ ′trans′ list*
where *translateTrace = map translateTrans*

definition *translateO* :: *′obs list ⇒ ′obs′ list*
where *translateO ol = these (map translateObs ol)*

definition *translateV* :: *′val list ⇒ ′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.prems* **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.prems* **unfolding** *Prime.validFrom-Cons* **by** *auto*
    show *thesis* **proof** (*rule Cons.IH*)
    show *Prime.validFrom* (*tgtOf′ trn′*) *tr′* **using** *Cons.prems* **unfolding** *Prime.validFrom-Cons* **by** *auto*
      show *tgtOf′ trn′ = translateState* (*tgtOf trn*) **using** *trn Cons.prems*(*4*) *tgtOf′-tgtOf* **by** *auto*
      show *Orig.reach* (*tgtOf trn*) **using** *trn Cons.prems*(*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.prems*
        by (*intro Cons.prems*(*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** *φ′-φ*[*of trn*] *φ-φ′*[*of trn*] *Cons*(*3*) *Cons.IH*
      by (*cases φ trn*; *cases φ′* (*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** γ'-γ[*of trn*] γ-γ'[*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*. *Orig.validFrom istate tr* $\wedge$ *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. translateV vl1 = vl1'* $\wedge$ *B (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* $\Longrightarrow$ *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 γ' :: *'trans* $\Rightarrow$ *bool*
where γ' *trn* $\equiv$ γ *trn* $\wedge$ *translateObs (g trn)* $\neq$ *None*

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

sublocale *Prime?*: *BD-Security-TS istate validTrans srcOf tgtOf* φ *f* γ' *g' T B* **.**

sublocale *BD-Security-TS-Trans istate validTrans srcOf tgtOf* φ *f* γ *g T B*
  istate validTrans srcOf tgtOf φ *f* γ' *g' T B*
  id id translateObs Some
by *(unfold-locales) (auto simp:* γ'*-def g'-def)*

theorem *Orig.secure* $\Longrightarrow$ *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* $= 'nodeid \Rightarrow 'state$
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
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 :: $'nodeid \Rightarrow 'trans \Rightarrow 'nodeid \Rightarrow 'trans \Rightarrow bool$
assumes
finite-nodes: *finite nodes*
and
isCom-tgtNodeOf: $\bigwedge nid\ trn.$
$[\![validTrans\ nid\ trn;\ comOf\ nid\ trn = Send \lor comOf\ nid\ trn = Recv;$
Transition-System.reach (*istate nid*) (*validTrans nid*) (*srcOf nid*) (*tgtOf nid*) (*srcOf nid trn*)$]\!]$
$\Longrightarrow tgtNodeOf\ nid\ trn \neq nid$
begin

abbreviation *isCom* :: $'nodeid \Rightarrow 'trans \Rightarrow bool$
where *isCom nid trn* $\equiv$ $(comOf\ nid\ trn = Send \lor comOf\ nid\ trn = Recv) \land tgtNodeOf\ nid\ trn \in nodes$

abbreviation *lreach* :: $'nodeid \Rightarrow 'state \Rightarrow bool$
where *lreach nid s* $\equiv$ *Transition-System.reach* (*istate nid*) (*validTrans nid*) (*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* ⇒ *bool* **where**
Local: *nValidTrans* (*LTrans s nid trn*) =
(*validTrans nid trn* ∧ *srcOf nid trn* = *s nid* ∧ *nid* ∈ *nodes* ∧ ¬*isCom nid trn*)
| *Comm*: *nValidTrans* (*CTrans s nid1 trn1 nid2 trn2*) =
(*validTrans nid1 trn1* ∧ *srcOf nid1 trn1* = *s nid1* ∧ *comOf nid1 trn1* = *Send*
∧ *tgtNodeOf nid1 trn1* = *nid2* ∧
validTrans nid2 trn2 ∧ *srcOf nid2 trn2* = *s nid2* ∧ *comOf nid2 trn2* = *Recv*
∧ *tgtNodeOf nid2 trn2* = *nid1* ∧
nid1 ∈ *nodes* ∧ *nid2* ∈ *nodes* ∧ *nid1* ≠ *nid2* ∧
sync nid1 trn1 nid2 trn2)

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

fun *nTgtOf* :: ($'nodeid$, $'state$, $'trans$) *ntrans* ⇒ ($'nodeid$, $'state$) *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 ≠ [] ⟹ *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 ≠ [] ⟹ *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.prems*, *cases trn*) (*auto*)
qed
qed
qed

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

inductive *comp* :: ($'nodeid$, $'state$) *nstate* ⇒ ($'nodeid$, $'state$, $'trans$) *ntrans list* ⇒ *bool*
where
Nil: *comp s* []
| *Local*: ⋀*s trn s' tr nid*.
⟦*comp s tr*; *tgtOf nid trn* = *s nid*; *s'* = *s*(*nid* := *srcOf nid trn*); *nid* ∈ *nodes*; ¬*isCom nid trn*⟧
⟹ *comp s'* (*LTrans s' nid trn* # *tr*)
| *Comm*: ⋀*s trn1 trn2 s' tr nid1 nid2*.
⟦*comp s tr*; *tgtOf nid1 trn1* = *s nid1*; *tgtOf nid2 trn2* = *s nid2*;
s' = *s*(*nid1* := *srcOf nid1 trn1*, *nid2* := *srcOf nid2 trn2*);
nid1 ∈ *nodes*; *nid2* ∈ *nodes*; *nid1* ≠ *nid2*;
comOf nid1 trn1 = *Send*; *tgtNodeOf nid1 trn1* = *nid2*;
comOf nid2 trn2 = *Recv*; *tgtNodeOf nid2 trn2* = *nid1*;
sync nid1 trn1 nid2 trn2⟧
⟹ *comp s'* (*CTrans s' nid1 trn1 nid2 trn2* # *tr*)

abbreviation *lValidFrom* :: $'nodeid$ ⇒ $'state$ ⇒ $'trans$ *list* ⇒ *bool* **where**
lValidFrom nid ≡ *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* $\Longrightarrow$ *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 ⟷ *comp s tr* ∧ (∀ *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* ⟹ *P* **by** (*cases trn*) *auto*
lemma *validE*: *valid tr* ⟹ *P* **by** (*induction rule*: *valid.induct*) (*auto elim*: *nValidTransE*)
lemma *validFrom-iff-Nil*: *validFrom s tr* ⟷ *tr* = [] **unfolding** *validFrom-def* **by** (*auto elim*: *validE*)
lemma *reach-istate*: *reach s* ⟹ *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* ⇒ ′*trans* ⇒ *bool*
and
srcOf :: ′*nodeid* ⇒ ′*trans* ⇒ ′*state* **and** *tgtOf* :: ′*nodeid* ⇒ ′*trans* ⇒ ′*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
φ :: $'nodeid \Rightarrow 'trans \Rightarrow bool$ **and** f :: $'nodeid \Rightarrow 'trans \Rightarrow 'val$
and
γ :: $'nodeid \Rightarrow 'trans \Rightarrow bool$ **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
$comOfV$ :: $'nodeid \Rightarrow 'val \Rightarrow com$
and
$tgtNodeOfV$ :: $'nodeid \Rightarrow 'val \Rightarrow 'nodeid$
and
$syncV$ :: $'nodeid \Rightarrow 'val \Rightarrow 'nodeid \Rightarrow 'val \Rightarrow bool$
and
$comOfO$ :: $'nodeid \Rightarrow 'obs \Rightarrow com$
and
$tgtNodeOfO$ :: $'nodeid \Rightarrow 'obs \Rightarrow 'nodeid$
and
$syncO$ :: $'nodeid \Rightarrow 'obs \Rightarrow 'nodeid \Rightarrow 'obs \Rightarrow bool$

and
$source$ :: $'nodeid$

assumes
comOfV-comOf[*simp*]:
$\bigwedge nid\ trn.\ [\![validTrans\ nid\ trn;\ lreach\ nid\ (srcOf\ nid\ trn);\ \varphi\ nid\ trn]\!] \Longrightarrow comOfV\ nid\ (f\ nid\ trn) = comOf\ nid\ trn$
and
tgtNodeOfV-tgtNodeOf[*simp*]:
$\bigwedge nid\ trn.\ [\![validTrans\ nid\ trn;\ lreach\ nid\ (srcOf\ nid\ trn);\ \varphi\ nid\ trn;\ comOf\ nid\ trn = Send \vee comOf\ nid\ trn = Recv]\!]$
$\Longrightarrow tgtNodeOfV\ nid\ (f\ nid\ trn) = tgtNodeOf\ nid\ trn$
and
comOfO-comOf[*simp*]:
$\bigwedge nid\ trn.\ [\![validTrans\ nid\ trn;\ lreach\ nid\ (srcOf\ nid\ trn);\ \gamma\ nid\ trn]\!] \Longrightarrow comOfO\ nid\ (g\ nid\ trn) = comOf\ nid\ trn$
and
tgtNodeOfO-tgtNodeOf[*simp*]:
$\bigwedge nid\ trn.\ [\![validTrans\ nid\ trn;\ lreach\ nid\ (srcOf\ nid\ trn);\ \gamma\ nid\ trn;\ comOf\ nid\ trn = Send \vee comOf\ nid\ trn = Recv]\!]$
$\Longrightarrow tgtNodeOfO\ nid\ (g\ nid\ trn) = tgtNodeOf\ nid\ trn$
and
sync-syncV:
$\bigwedge nid1\ trn1\ nid2\ trn2.$

$validTrans\ nid1\ trn1 \Longrightarrow lreach\ nid1\ (srcOf\ nid1\ trn1) \Longrightarrow$
$validTrans\ nid2\ trn2 \Longrightarrow lreach\ nid2\ (srcOf\ nid2\ trn2) \Longrightarrow$
$comOf\ nid1\ trn1 = Send \Longrightarrow tgtNodeOf\ nid1\ trn1 = nid2 \Longrightarrow$
$comOf\ nid2\ trn2 = Recv \Longrightarrow tgtNodeOf\ nid2\ trn2 = nid1 \Longrightarrow$
$\varphi\ nid1\ trn1 \Longrightarrow \varphi\ nid2\ trn2 \Longrightarrow$
$sync\ nid1\ trn1\ nid2\ trn2 \Longrightarrow syncV\ nid1\ (f\ nid1\ trn1)\ nid2\ (f\ nid2\ trn2)$
and
sync-syncO:
$\bigwedge nid1\ trn1\ nid2\ trn2.$
$validTrans\ nid1\ trn1 \Longrightarrow lreach\ nid1\ (srcOf\ nid1\ trn1) \Longrightarrow$
$validTrans\ nid2\ trn2 \Longrightarrow lreach\ nid2\ (srcOf\ nid2\ trn2) \Longrightarrow$
$comOf\ nid1\ trn1 = Send \Longrightarrow tgtNodeOf\ nid1\ trn1 = nid2 \Longrightarrow$
$comOf\ nid2\ trn2 = Recv \Longrightarrow tgtNodeOf\ nid2\ trn2 = nid1 \Longrightarrow$
$\gamma\ nid1\ trn1 \Longrightarrow \gamma\ nid2\ trn2 \Longrightarrow$
$sync\ nid1\ trn1\ nid2\ trn2 \Longrightarrow syncO\ nid1\ (g\ nid1\ trn1)\ nid2\ (g\ nid2\ trn2)$
and
sync-φ1-φ2:
$\bigwedge nid1\ trn1\ nid2\ trn2.$
$validTrans\ nid1\ trn1 \Longrightarrow lreach\ nid1\ (srcOf\ nid1\ trn1) \Longrightarrow$
$validTrans\ nid2\ trn2 \Longrightarrow lreach\ nid2\ (srcOf\ nid2\ trn2) \Longrightarrow$
$comOf\ nid1\ trn1 = Send \Longrightarrow tgtNodeOf\ nid1\ trn1 = nid2 \Longrightarrow$
$comOf\ nid2\ trn2 = Recv \Longrightarrow tgtNodeOf\ nid2\ trn2 = nid1 \Longrightarrow$
$sync\ nid1\ trn1\ nid2\ trn2 \Longrightarrow \varphi\ nid1\ trn1 \longleftrightarrow \varphi\ nid2\ trn2$
and
sync-φ-γ:
$\bigwedge nid1\ trn1\ nid2\ trn2.$
$validTrans\ nid1\ trn1 \Longrightarrow lreach\ nid1\ (srcOf\ nid1\ trn1) \Longrightarrow$
$validTrans\ nid2\ trn2 \Longrightarrow lreach\ nid2\ (srcOf\ nid2\ trn2) \Longrightarrow$
$comOf\ nid1\ trn1 = Send \Longrightarrow tgtNodeOf\ nid1\ trn1 = nid2 \Longrightarrow$
$comOf\ nid2\ trn2 = Recv \Longrightarrow tgtNodeOf\ nid2\ trn2 = nid1 \Longrightarrow$
$\gamma\ nid1\ trn1 \Longrightarrow \gamma\ nid2\ trn2 \Longrightarrow$
$syncO\ nid1\ (g\ nid1\ trn1)\ nid2\ (g\ nid2\ trn2) \Longrightarrow$
$(\varphi\ nid1\ trn1 \Longrightarrow \varphi\ nid2\ trn2 \Longrightarrow syncV\ nid1\ (f\ nid1\ trn1)\ nid2\ (f\ nid2\ trn2))$
$\Longrightarrow$
$sync\ nid1\ trn1\ nid2\ trn2$
and
isCom-γ: $\bigwedge nid\ trn.\ validTrans\ nid\ trn \Longrightarrow lreach\ nid\ (srcOf\ nid\ trn) \Longrightarrow comOf\ nid\ trn = Send \vee comOf\ nid\ trn = Recv \Longrightarrow \gamma\ nid\ trn$
and
φ-source: $\bigwedge nid\ trn.\ [\![validTrans\ nid\ trn;\ lreach\ nid\ (srcOf\ nid\ trn);\ \varphi\ nid\ trn;\ nid \neq source;\ nid \in nodes]\!]$
$\Longrightarrow isCom\ nid\ trn \wedge tgtNodeOf\ nid\ trn = source \wedge source \in nodes$
begin

abbreviation $isComO\ nid\ obs \equiv (comOfO\ nid\ obs = Send \vee comOfO\ nid\ obs = Recv) \wedge tgtNodeOfO\ nid\ obs \in nodes$
abbreviation $isComV\ nid\ val \equiv (comOfV\ nid\ val = Send \vee comOfV\ nid\ val = Recv) \wedge tgtNodeOfV\ nid\ val \in nodes$

fun $n\varphi$ :: $('nodeid, 'state, 'trans)$ *ntrans* $\Rightarrow$ *bool* **where**
$n\varphi$ (*LTrans s nid trn*) = φ *nid trn*
| $n\varphi$ (*CTrans s nid1 trn1 nid2 trn2*) = (φ *nid1 trn1* $\vee$ φ *nid2 trn2*)

fun *nf* :: $('nodeid, 'state, 'trans)$ *ntrans* $\Rightarrow$ $('nodeid, 'val)$ *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 $n\gamma$ :: $('nodeid, 'state, 'trans)$ *ntrans* $\Rightarrow$ *bool* **where**
$n\gamma$ (*LTrans s nid trn*) = γ *nid trn*
| $n\gamma$ (*CTrans s nid1 trn1 nid2 trn2*) = (γ *nid1 trn1* $\vee$ γ *nid2 trn2*)

fun *ng* :: $('nodeid, 'state, 'trans)$ *ntrans* $\Rightarrow$ $('nodeid, 'obs)$ *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* $\Rightarrow$ *bool* **where**
nT (*LTrans s nid trn*) = *T nid trn*
| *nT* (*CTrans s nid1 trn1 nid2 trn2*) = (*T nid1 trn1* $\vee$ *T nid2 trn2*)

fun *decompV* :: $('nodeid, 'val)$ *nvalue list* $\Rightarrow$ *'nodeid* $\Rightarrow$ *'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* $\Rightarrow$ *bool* **where**
nValidV (*LVal nid v*) = (*nid* $\in$ *nodes* $\wedge$ $\neg$*isComV nid v*)
| *nValidV* (*CVal nid1 v1 nid2 v2*) =
(*nid1* $\in$ *nodes* $\wedge$ *nid2* $\in$ *nodes* $\wedge$ *nid1* $\neq$ *nid2* $\wedge$ *syncV nid1 v1 nid2 v2* $\wedge$
comOfV nid1 v1 = *Send* $\wedge$ *tgtNodeOfV nid1 v1* = *nid2* $\wedge$ *comOfV nid2 v2* = *Recv* $\wedge$ *tgtNodeOfV nid2 v2* = *nid1*)

fun *decompO* :: $('nodeid, 'obs)$ *nobs list* $\Rightarrow$ *'nodeid* $\Rightarrow$ *'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* ⇒ (*'nodeid*, *'val*) *nvalue list* ⇒ *bool*
where
nB vl vl' ≡ (∀ *nid* ∈ *nodes*. *B nid* (*decompV vl nid*) (*decompV vl' nid*)) ∧
(*list-all nValidV vl* ⟶ *list-all nValidV vl'*)

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

lemma *decompV-subDecompV*[*simp*]: *nid* ∈ *nds* ⟹ *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 nφ nf nγ ng nT nB*
.

abbreviation *lV* :: *'nodeid* ⇒ *'trans list* ⇒ *'val list* **where**
lV nid ≡ *BD-Security-TS.V* (*φ nid*) (*f nid*)

abbreviation *lO* :: *'nodeid* ⇒ *'trans list* ⇒ *'obs list* **where**
lO nid ≡ *BD-Security-TS.O* (*γ nid*) (*g nid*)

abbreviation *lTT* :: *'nodeid* ⇒ *'trans list* ⇒ *bool* **where**
lTT nid ≡ *never* (*T nid*)

abbreviation *lsecure* :: *'nodeid* ⇒ *bool* **where**
lsecure nid ≡ *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*
φ *nid f nid* γ *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*φ *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 φ *nid1 trn1* $\longleftrightarrow$ φ *nid2 trn2*
using *Cons.prems CTrans* **by** (*intro sync-*φ*1-*φ*2*) *auto*
then show *?thesis* **using** *Cons.prems CTrans tr Node.V-Cons-unfold* **by** (*cases n*φ *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*
φ *nid f nid* γ *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*γ *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 γ *nid1 trn1* **and** γ *nid2 trn2*
using *Cons.prems CTrans* **by** (*auto intro*: *isCom-*γ)
then show *?thesis* **using** *Cons.prems CTrans tr Node.O-Cons-unfold* **by** (*cases n*γ *trn*) *auto*
qed
qed *auto*
qed

lemma *nTT-TT*: *never nT tr* $\Longrightarrow$ *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.prems*
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.prems* **by** (*auto elim*: *reach-lreach*)
ultimately show *?thesis* **using** *Cons.prems 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.prems* **by** (*auto elim*: *reach-lreach*)
moreover then have φ *nid1 trn1* $\longleftrightarrow$ φ *nid2 trn2*
using *Cons.prems CTrans* **by** (*intro sync-φ1-φ2*) *auto*
ultimately show *?thesis* **using** *Cons.prems tr* **by** (*cases nφ trn*) (*auto intro*: *sync-syncV*)
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* $\Longrightarrow$ *P* **by** (*cases v*) *auto*
lemma *list-all-nValidV-Nil*: *list-all nValidV vl* $\Longrightarrow$ *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*
φ *source f source* γ *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'* $\Longrightarrow$ *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 $\Longrightarrow$ *Node.reach s* $\Longrightarrow$ *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.prems Cons.IH*[*of tgtOf source trn*]
      using *isCom-tgtNodeOf* **by** (*cases* φ *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 φ = φ *source* **and** *f* = *f source* **and** γ = γ *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 φ' = $n\varphi$ **and** *f'* = *nf* **and** γ' = $n\gamma$ **and** *g'* = *ng* **and** *T'* = *nT* **and** *B'* = *nB*
  and *translateState* = λ*s. istate*(*source* := *s*)
  and *translateTrans* = λ*trn. LTrans* (*istate*(*source* := *srcOf source trn*)) *source trn*
  and *translateObs* = λ*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 φ*-source'*:
$\bigwedge$*nid trn*. $[\![$*validTrans nid trn*; *Orig.lreach nid* (*srcOf nid trn*); φ *nid trn*; *nid* $\neq$ *source*; *nid* $\in$ *nodes'*$]\!]$
$\Longrightarrow$ *Orig.isCom nid trn* $\wedge$ *tgtNodeOf nid trn* = *source* $\wedge$ *source* $\in$ *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-*φ*1-*φ*2* **by** *auto* **next**
case *10* **then show** *?case* **using** *Orig.sync-*φ*-*γ **by** *auto* **next**
case *11* **then show** *?case* **using** *Orig.isCom-*γ **by** *auto* **next**
case *12* **then show** *?case* **using** φ*-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 φ*-NID-source*:
$\bigwedge$*trn*. $[\![$*validTrans NID trn*; *Sub.lreach NID* (*srcOf NID trn*); φ *NID trn*$]\!]$
$\Longrightarrow$ *Sub.isCom NID trn* $\wedge$ *tgtNodeOf NID trn* = *source* $\wedge$ *source* $\in$ *nodes*
begin

sublocale *Node*: *BD-Security-TS istate NID validTrans NID srcOf NID tgtOf NID*
φ *NID f NID* γ *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* ⇒ *bool* **where**
isCom1 (*LTrans s nid trn*) = (*nid* ∈ *nodes* ∧ *isCom nid trn* ∧ *tgtNodeOf nid trn* = *NID*)
| *isCom1* - = *False*

definition *isCom2 trn* = (∃ *nid*. *nid* ∈ *nodes* ∧ *isCom NID trn* ∧ *tgtNodeOf NID trn* = *nid*)

fun *Sync* :: (*′nodeid,′state,′trans*) *ntrans* ⇒ *′trans* ⇒ *bool* **where**
Sync (*LTrans s nid trn*) *trn′* = (*tgtNodeOf nid trn* = *NID* ∧ *tgtNodeOf NID trn′* = *nid* ∧
((*sync nid trn NID trn′* ∧ *comOf nid trn* = *Send* ∧ *comOf NID trn′* = *Recv*)
∨ (*sync NID trn′ nid trn* ∧ *comOf NID trn′* = *Send* ∧ *comOf nid trn* = *Recv*)))
| *Sync* - - = *False*

fun *isComV1* :: (*′nodeid,′val*) *nvalue* ⇒ *bool* **where**
isComV1 (*LVal nid v*) = (*nid* ∈ *nodes* ∧ *isComV nid v* ∧ *tgtNodeOfV nid v* = *NID*)
| *isComV1* - = *False*

definition *isComV2 v* = (∃ *nid*. *nid* ∈ *nodes* ∧ *isComV NID v* ∧ *tgtNodeOfV NID v* = *nid*)

fun *SyncV* :: (*′nodeid,′val*) *nvalue* ⇒ *′val* ⇒ *bool* **where**
SyncV (*LVal nid v1*) *v2* = (*tgtNodeOfV nid v1* = *NID* ∧ *tgtNodeOfV NID v2* = *nid* ∧
((*syncV nid v1 NID v2* ∧ *comOfV nid v1* = *Send* ∧ *comOfV NID v2* = *Recv*)
∨ (*syncV NID v2 nid v1* ∧ *comOfV NID v2* = *Send* ∧ *comOfV nid v1* = *Recv*)))
| *SyncV* - - = *False*

fun *CmpV* :: (*′nodeid,′val*) *nvalue* ⇒ *′val* ⇒ (*′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* ⇒ *bool* **where**
isComO1 (*LObs nid obs*) = (*nid* ∈ *nodes* ∧ *isComO nid obs* ∧ *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-Security-TS-Comp istate Sub.nValidTrans Sub.nSrcOf Sub.nTgtOf Sub.nφ Sub.nf Sub.nγ Sub.ng Sub.nT Sub.nB*
istate NID validTrans NID srcOf NID tgtOf NID φ NID f NID γ 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.nφ 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.nγ 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*) φ *NID trn2*
then show *isCom2 trn2* = *isComV2* (*f NID trn2*)
unfolding *isCom2-def isComV2-def* **by** *auto*
next
fix *trn2*
assume *trn2*: *validTrans NID trn2 Node.reach* (*srcOf NID trn2*) γ *NID trn2*
then show *isCom2 trn2* = *isComO2* (*g NID trn2*)
unfolding *isCom2-def isComO2-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.n*φ *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.n*γ *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.n*φ *trn1* = φ *NID trn2* **proof** (*cases trn1*)
case (*LTrans s nid trn*)
with *trn12* **show** *?thesis* **using** *Sub.sync-*φ*1-*φ*2*[*of nid trn NID trn2*] *Sub.sync-*φ*1-*φ*2*[*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* $\Longrightarrow$ φ *NID trn2* $\Longrightarrow$ *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.n*γ *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 γ *NID trn2* **unfolding** *isCom2-def* **by** (*auto intro*: *Sub.isCom-*γ)
next
  fix *trn2*
  assume *validTrans NID trn2 Node.reach* (*srcOf NID trn2*) φ *NID trn2*
  then show *isCom2 trn2* **using** φ*-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* ⇒ '*state*) × '*state*) ⇒ ('*nodeid* ⇒ '*state*) **where**
  translateState (*sSub*, *sNode*) = (*sSub*(*NID* := *sNode*))

fun *translateTrans* :: ('*nodeid* ⇒ '*state*, ('*nodeid*, '*state*, '*trans*) *ntrans*, '*state*, '*trans*) *ctrans* ⇒ ('*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*) *nobs*, '*obs*) *cobs* ⇒ ('*nodeid*, '*obs*) *nobs* **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* ⇒ ('*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.*φ

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

lemma *validFrom-compValidV*: *Comp.reach s* ⟹ *Comp.validFrom s tr* ⟹ *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.prems 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* ⟹ *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*
⟷ *vl1* = *subDecompV* (*map translateVal vl*) *nodes* ∧ *vl2* = *decompV* (*map translateVal vl*) *NID*
proof
assume *Comp.compV vl1 vl2 vl*
then show *vl1* = *subDecompV* (*map translateVal vl*) *nodes* ∧ *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* ∧ *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 ¬ *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.prems* **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 ¬ (*nid1* = *NID* ∨ *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 ∃ *trn*. *Comp.validTrans trn* ∧ *Comp.srcOf trn* = *s* ∧ *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*) *nγ* (*translateTrans trn*)
then show *Comp.γ trn* ∧ (*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-γ*[*of nid trnSub*] *isCom-γ*[*of NID trnNode*] **by** *auto*
qed *auto*
next
fix *trn*
assume *trn*: *Comp.validTrans trn Comp.reach* (*Comp.srcOf trn*) *Comp.γ trn*
then show *nγ* (*translateTrans trn*) ∨ (*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*: *compV-decompV intro*: *nValidV-subDecompV list-all-compValidV-nValidV*)
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* φ*-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* φ*-source* **unfolding** *nodes-nodesLeftRight*

proof (*induction rule*: *finite-induct*)
case (*insert nid nodesMerged*)
interpret *Left'*: *BD-Security-TS-Network-Change-Nodes* **where** *nodes'* = *nodesLeft* ∪ *nodesMerged*
using *finite-nodes nodes-nodesLeftRight insert* **by** (*unfold-locales*) *auto*
interpret *Node*: *BD-Security-TS istate nid validTrans nid srcOf nid tgtOf nid* φ *nid f nid* γ *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* ∪ *nodesMerged* **and** *NID* = *nid*
using *insert.prems 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 ∀ *nid* ∈ *nodes*. *lsecure nid*
shows *secure*
proof (*cases source* ∈ *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 φ *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 φ :: *'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* (λ-. []) []
| *Local*: $[\![$*compSec sls sl*; *isComV nid s* $\longrightarrow$ *tgtNodeOfV nid s* $\notin$ *nodes*; *nid* $\in$ *nodes*$]\!]$
$\Longrightarrow$ *compSec* (*sls*(*nid* := *s* # *sls nid*)) (*LMSec nid s* # *sl*)
| *Comm*: $[\![$*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$]\!]$
$\Longrightarrow$ *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*φ *nf n*γ *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.prems*[*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.prems[of CVal nid1 s1 nid2 s2 # sl']*) *auto*
qed
qed

interpretation *Trans*: *BD-Security-TS-Trans istate nValidTrans nSrcOf nTgtOf* $n\varphi$ *Net.nf* $n\gamma$ *ng nT Net.nB*
istate nValidTrans nSrcOf nTgtOf $n\varphi$ *nf* $n\gamma$ *ng nT nB*
id id Some Some o translateSec
proof (*unfold-locales, 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** $\forall$ *nid* $\in$ *nodes. B nid* (*decompV* (*Net.V tr*) *nid*) (*sls1 nid*)
using ‹*nB sl' sl1'*› ‹*these* (*map* (*Some* $\circ$ *translateSec*) (*Net.V tr*)) = *sl'*› **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 $\forall$ *nid* $\in$ *nodes. lsecure 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*: $[\![$*srcSyncV nid vlSrc vlNode*; $\neg$*isComV source v* $\vee$ *tgtNodeOfV source v* $\neq$ *nid*$]\!]$
$\Longrightarrow$ *srcSyncV nid* (*v* # *vlSrc*) *vlNode*
| *Send*: $[\![$*srcSyncV nid vlSrc vlNode*; *comOfV source vSrc* = *Send*; *comOfV nid vNode* = *Recv*;
tgtNodeOfV source vSrc = *nid*; *tgtNodeOfV nid vNode* = *source*;
syncV source vSrc nid vNode$]\!]$ $\Longrightarrow$ *srcSyncV nid* (*vSrc* # *vlSrc*) (*vNode* # *vlNode*)
| *Recv*: $[\![$*srcSyncV nid vlSrc vlNode*; *comOfV source vSrc* = *Recv*; *comOfV nid vNode* = *Send*;
tgtNodeOfV source vSrc = *nid*; *tgtNodeOfV nid vNode* = *source*;
syncV nid vNode source vSrc$]\!]$ $\Longrightarrow$ *srcSyncV nid* (*vSrc* # *vlSrc*) (*vNode* # *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 *binSyncV* :: *'nodeid* $\Rightarrow$ *'nodeid* $\Rightarrow$ *'val list* $\Rightarrow$ *'val list* $\Rightarrow$ *bool*
for *nid1 nid2* :: *'nodeid*
where
Nil: *binSyncV nid1 nid2* [] []
| *Int1*: $[\![$*binSyncV nid1 nid2 vl1 vl2*; $\neg$*isComV nid1 v* $\vee$ *tgtNodeOfV nid1 v* $\neq$ *nid2*$]\!]$
$\Longrightarrow$ *binSyncV nid1 nid2* (*v* # *vl1*) *vl2*
| *Int2*: $[\![$*binSyncV nid1 nid2 vl1 vl2*; $\neg$*isComV nid2 v* $\vee$ *tgtNodeOfV nid2 v* $\neq$ *nid1*$]\!]$
$\Longrightarrow$ *binSyncV nid1 nid2 vl1* (*v* # *vl2*)
| *Send*: $[\![$*binSyncV nid1 nid2 vl1 vl2*; *comOfV nid1 v1* = *Send*; *comOfV nid2 v2* = *Recv*;
tgtNodeOfV nid1 v1 = *nid2*; *tgtNodeOfV nid2 v2* = *nid1*;
syncV nid1 v1 nid2 v2$]\!]$ $\Longrightarrow$ *binSyncV nid1 nid2* (*v1* # *vl1*) (*v2* # *vl2*)
| *Recv*: $[\![$*binSyncV nid1 nid2 vl1 vl2*; *comOfV nid1 v1* = *Recv*; *comOfV nid2 v2* = *Send*;
tgtNodeOfV nid1 v1 = *nid2*; *tgtNodeOfV nid2 v2* = *nid1*;
syncV nid2 v2 nid1 v1$]\!]$ $\Longrightarrow$ *binSyncV nid1 nid2* (*v1* # *vl1*) (*v2* # *vl2*)

lemma *srcSyncV-binSyncV*:
assumes *source* $\in$ *nodes* **and** *nid2* $\in$ *nodes*
shows *srcSyncV nid2 vl1 vl2* $\longleftrightarrow$ (*binSyncV source nid2 vl1 vl2* $\wedge$
list-all (λ*v*. *isComV nid2 v* $\wedge$ *tgtNodeOfV nid2 v* = *source*) *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 *binSyncV source nid2 vl1 vl2*
and *list-all* (λ*v*. *isComV nid2 v* $\wedge$ *tgtNodeOfV nid2 v* = *source*) *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*: $\bigwedge nid\ tr\ vl'\ vl1.$
$[\![B\ source\ (lV\ source\ tr)\ vl1;\ srcSyncV\ nid\ (lV\ source\ tr)\ vl';$
$lValidFrom\ source\ (istate\ source)\ tr;\ never\ (T\ source)\ tr;$
$nid \in nodes;\ nid \neq source]\!]$
$\Longrightarrow (\exists vl1'.\ B\ nid\ vl'\ vl1' \wedge srcSyncV\ nid\ vl1\ vl1')$
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 $list\text{-}all\ (\lambda v.\ isComV\ source\ v \longrightarrow tgtNodeOfV\ source\ v \neq source)\ (decompV\ vl'\ source)$
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 $list\text{-}all\ (\lambda v.\ isComV\ source\ v \longrightarrow tgtNodeOfV\ source\ v \neq source)\ (lV\ source\ tr)$
(**is** *?goal tr*)

proof −
interpret *Node*: *BD-Security-TS istate source validTrans source srcOf source tgtOf source*
φ *source f source* γ *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* φ *source trn*) *auto*
qed *auto*
qed

lemma *merge-projection*:
assumes *isProjectionOf p vl*
and *list-all* (λv. *isComV source v* $\longrightarrow$ *tgtNodeOfV source v* $\neq$ *source*) *vl*
obtains vl' **where** $\forall$ *nid* $\in$ $nodes'$. *decompV* vl' *nid* = *p nid*
and *decompV* vl' *source* = *vl*
and *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* (λv. *isComV source v* $\longrightarrow$ *tgtNodeOfV source v* $\neq$ *source*) *vl*
using *Cons*(*4*) **by** *auto*
next
fix vl'
assume $\forall$ *nid*$\in$$nodes'$. *decompV* vl' *nid* = *p nid* **and** *decompV* vl' *source* = *vl*
and *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* $\longrightarrow$ *tgtNodeOfV source v* $\neq$ *source*) *vl*
using *Cons*(*4*) **by** *auto*
next
fix *vl'*
assume *vl'*: $\forall$ *nid*$\in$*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* $\neq$ *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'* $\longrightarrow$ *nid* = *source*
using *Cons.prems Net.*φ*-source*[*of nid trn'*] *reach-lreach* **by** *auto*
ultimately show *?thesis* **using** *Cons.IH*[*OF tr r*] **by** (*cases n*φ *trn*) *auto*
next
case (*CTrans s' nid1 trn1 nid2 trn2*)
moreover then have *n*φ *trn* $\longrightarrow$ (*nid1* = *source* $\vee$ *nid2* = *source*)
using *Cons.prems Net.*φ*-source*[*of nid1 trn1*] *Net.*φ*-source*[*of nid2 trn2*] *reach-lreach*
by (*cases nid1* $\neq$ *source*) *auto*
ultimately show *?thesis* **using** *Cons.IH*[*OF tr r*] **by** (*cases n*φ *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.prems Net.*φ*-source*[*of nid′ trn′*] *reach-lreach LTrans* **by** *auto*
then show *?thesis* **using** *Cons.IH*[*OF tr r*] *Cons.prems 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.prems* **by** *auto*
show *?thesis* **proof** (*cases*)
assume φ: *n*φ *trn*
then have *nid1* = *source* $\vee$ *nid2* = *source*
using *CTrans Cons.prems 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 ¬*nφ trn*
with *Cons.IH*[*OF tr r*] show *?thesis* by *auto*
qed
qed
qed (*auto intro*: *srcSyncV.Nil*)

sublocale *BD-Security-TS-Trans istate nValidTrans nSrcOf nTgtOf nφ nf nγ ng nT nB*
istate nValidTrans nSrcOf nTgtOf nφ nf′ nγ 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 *φ*: *nφ* (*id trn*)
show *nφ trn* ∧ (*Some* ∘ *translateVal*) (*nf trn*) = *Some* (*nf′* (*id trn*))
proof (*cases trn*)
case (*LTrans s nid trn′*)
moreover then have *nid* = *source* using *trn r φ Net.φ-source*[*of nid trn′*] *reach-lreach* by *auto*
ultimately show *?thesis* using *φ* by *auto*
next
case (*CTrans s nid1 trn1 nid2 trn2*)
moreover then have *nid1* = *source* ∨ *nid2* = *source*
using *trn r φ Net.φ-source*[*of nid1 trn1*] *Net.φ-source*[*of nid2 trn2*] *reach-lreach*[*OF r*]
by (*cases nid1* ≠ *source*) *auto*
ultimately show *?thesis* using *φ* 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* ∘ *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 ∀ *nid* ∈ *nodes′*. *srcSyncV nid* (*decompV* (*V tr*) *source*) (*decompV* (*V*

tr) *nid*)
and *tgt-vl1′*: *list-all* (λv. *isComV source v* $\longrightarrow$ *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*. $\forall$ *nid* $\in$ *nodes′*. *B nid* (*decompV* (*V tr*) *nid*) (*p nid*) $\wedge$ *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′*: $\forall$ *nid* $\in$ *nodes′*. *B nid* (*decompV* (*V tr*) *nid*) (*p nid*)
unfolding *isProjectionOf-def* **by** *auto*
then obtain *vl1* **where** *p*: $\forall$ *nid* $\in$ *nodes′*. *decompV vl1 nid* = *p 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 $\forall$ *nid* $\in$ *nodes*. *B nid* (*decompV* (*V tr*) *nid*) (*decompV vl1 nid*)
proof
fix *nid*
assume *nid* $\in$ *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*. *these* (*map* (*Some* $\circ$ *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 $\forall$ *nid* $\in$ *nodes′*. *lsecure 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*: *comOfV source vSrc* = *Send* $\Longrightarrow$ *tgtNodeOfV source vSrc* = *nid* $\Longrightarrow$ *nid* $\neq$ *source* $\Longrightarrow$ (*syncV source vSrc nid vn* $\longleftrightarrow$ *vn* = *getTgtV vSrc*) $\wedge$ *tgtNodeOfV nid* (*getTgtV vSrc*) = *source* $\wedge$ *comOfV nid* (*getTgtV vSrc*) = *Recv*
and *getTgtV-Recv*: *comOfV source vSrc* = *Recv* $\Longrightarrow$ *tgtNodeOfV source vSrc* = *nid* $\Longrightarrow$ *nid* $\neq$ *source* $\Longrightarrow$ (*syncV nid vn source vSrc* $\longleftrightarrow$ *vn* = *getTgtV vSrc*) $\wedge$

tgtNodeOfV nid (*getTgtV vSrc*) = *source* ∧ *comOfV nid* (*getTgtV vSrc*) = *Send*
begin

abbreviation *projectSrcV nid vlSrc*
≡ *map getTgtV* (*filter* (λ*v*. *isComV source v* ∧ *tgtNodeOfV source v* = *nid*) *vlSrc*)

lemma *srcSyncV-projectSrcV*:
assumes *nid* ∈ *nodes* − {*source*}
shows *srcSyncV nid vlSrc vln* ⟷ *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*: *src-SyncV.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* ∈ *nodes*
and *source-secure*: *lsecure source*
and *B-source-in-B-sinks*: ⋀*nid tr vl vl1*.
⟦*B source vl vl1*; *vl* = *lV source tr*; *lValidFrom source* (*istate source*) *tr*; *never* (*T source*) *tr*;
 nid ∈ *nodes*; *nid* ≠ *source*⟧
⟹ *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* ⇒ ′*trans* ⇒ *bool*
 and
 srcOf :: ′*nodeid* ⇒ ′*trans* ⇒ ′*state* **and** *tgtOf* :: ′*nodeid* ⇒ ′*trans* ⇒ ′*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
$g\text{-}comOf$: $\bigwedge nid\ trn1\ trn2.$
$[\![validTrans\ nid\ trn1;\ lreach\ nid\ (srcOf\ nid\ trn1);\ \gamma\ nid\ trn1;$
$validTrans\ nid\ trn2;\ lreach\ nid\ (srcOf\ nid\ trn2);\ \gamma\ nid\ trn2;$
$g\ nid\ trn2 = g\ nid\ trn1]\!] \Longrightarrow comOf\ nid\ trn2 = comOf\ nid\ trn1$
and
$f\text{-}comOf$: $\bigwedge nid\ trn1\ trn2.$
$[\![validTrans\ nid\ trn1;\ lreach\ nid\ (srcOf\ nid\ trn1);\ \varphi\ nid\ trn1;$
$validTrans\ nid\ trn2;\ lreach\ nid\ (srcOf\ nid\ trn2);\ \varphi\ nid\ trn2;$
$f\ nid\ trn2 = f\ nid\ trn1]\!] \Longrightarrow comOf\ nid\ trn2 = comOf\ nid\ trn1$
and
$g\text{-}tgtNodeOf$: $\bigwedge nid\ trn1\ trn2.$
$[\![validTrans\ nid\ trn1;\ lreach\ nid\ (srcOf\ nid\ trn1);\ \gamma\ nid\ trn1;$
$validTrans\ nid\ trn2;\ lreach\ nid\ (srcOf\ nid\ trn2);\ \gamma\ nid\ trn2;$
$g\ nid\ trn2 = g\ nid\ trn1]\!] \Longrightarrow tgtNodeOf\ nid\ trn2 = tgtNodeOf\ nid\ trn1$
and
$f\text{-}tgtNodeOf$: $\bigwedge nid\ trn1\ trn2.$
$[\![validTrans\ nid\ trn1;\ lreach\ nid\ (srcOf\ nid\ trn1);\ \varphi\ nid\ trn1;$
$validTrans\ nid\ trn2;\ lreach\ nid\ (srcOf\ nid\ trn2);\ \varphi\ nid\ trn2;$
$f\ nid\ trn2 = f\ nid\ trn1]\!] \Longrightarrow tgtNodeOf\ nid\ trn2 = tgtNodeOf\ nid\ trn1$
and
$sync\text{-}\varphi 1\text{-}\varphi 2$:
$\bigwedge nid1\ trn1\ nid2\ trn2.$
$validTrans\ nid1\ trn1 \Longrightarrow lreach\ nid1\ (srcOf\ nid1\ trn1) \Longrightarrow$
$validTrans\ nid2\ trn2 \Longrightarrow lreach\ nid2\ (srcOf\ nid2\ trn2) \Longrightarrow$
$comOf\ nid1\ trn1 = Send \Longrightarrow tgtNodeOf\ nid1\ trn1 = nid2 \Longrightarrow$
$comOf\ nid2\ trn2 = Recv \Longrightarrow tgtNodeOf\ nid2\ trn2 = nid1 \Longrightarrow$
$sync\ nid1\ trn1\ nid2\ trn2 \Longrightarrow \varphi\ nid1\ trn1 \longleftrightarrow \varphi\ nid2\ trn2$
and
$sync\text{-}\varphi\text{-}\gamma$:
$\bigwedge nid1\ trn1\ nid2\ trn2.$
$validTrans\ nid1\ trn1 \Longrightarrow lreach\ nid1\ (srcOf\ nid1\ trn1) \Longrightarrow$
$validTrans\ nid2\ trn2 \Longrightarrow lreach\ nid2\ (srcOf\ nid2\ trn2) \Longrightarrow$

$comOf\ nid1\ trn1 = Send \Longrightarrow tgtNodeOf\ nid1\ trn1 = nid2 \Longrightarrow$
$comOf\ nid2\ trn2 = Recv \Longrightarrow tgtNodeOf\ nid2\ trn2 = nid1 \Longrightarrow$
$(\gamma\ nid1\ trn1 \Longrightarrow \gamma\ nid2\ trn2 \Longrightarrow$
$\exists\, trn1'\ trn2'.$
$validTrans\ nid1\ trn1' \wedge lreach\ nid1\ (srcOf\ nid1\ trn1') \wedge \gamma\ nid1\ trn1' \wedge g\ nid1\ trn1' = g\ nid1\ trn1\ \wedge$
$validTrans\ nid2\ trn2' \wedge lreach\ nid2\ (srcOf\ nid2\ trn2') \wedge \gamma\ nid2\ trn2' \wedge g\ nid2\ trn2' = g\ nid2\ trn2\ \wedge$
$sync\ nid1\ trn1'\ nid2\ trn2') \Longrightarrow$
$(\varphi\ nid1\ trn1 \Longrightarrow \varphi\ nid2\ trn2 \Longrightarrow$
$\exists\, trn1'\ trn2'.$
$validTrans\ nid1\ trn1' \wedge lreach\ nid1\ (srcOf\ nid1\ trn1') \wedge \varphi\ nid1\ trn1' \wedge f\ nid1\ trn1' = f\ nid1\ trn1\ \wedge$
$validTrans\ nid2\ trn2' \wedge lreach\ nid2\ (srcOf\ nid2\ trn2') \wedge \varphi\ nid2\ trn2' \wedge f\ nid2\ trn2' = f\ nid2\ trn2\ \wedge$
$sync\ nid1\ trn1'\ nid2\ trn2')$
$\Longrightarrow$
$sync\ nid1\ trn1\ nid2\ trn2$
and
isCom-γ: $\bigwedge nid\ trn.\ validTrans\ nid\ trn \Longrightarrow lreach\ nid\ (srcOf\ nid\ trn) \Longrightarrow comOf\ nid\ trn = Send \vee comOf\ nid\ trn = Recv \Longrightarrow \gamma\ nid\ trn$
and
φ*-source*: $\bigwedge nid\ trn.\ [\![validTrans\ nid\ trn;\ lreach\ nid\ (srcOf\ nid\ trn);\ \varphi\ nid\ trn;\ nid \neq source;\ nid \in nodes]\!]$
$\Longrightarrow isCom\ nid\ trn \wedge tgtNodeOf\ nid\ trn = source \wedge source \in nodes$
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 $syncO\ nid1\ obs1\ nid2\ obs2 =$
$(\exists\, trn1\ trn2.\ validTrans\ nid1\ trn1 \wedge lreach\ nid1\ (srcOf\ nid1\ trn1) \wedge \gamma\ nid1\ trn1 \wedge g\ nid1\ trn1 = obs1\ \wedge$
$validTrans\ nid2\ trn2 \wedge lreach\ nid2\ (srcOf\ nid2\ trn2) \wedge \gamma\ nid2\ trn2 \wedge g\ nid2\ trn2 = obs2\ \wedge$

sync nid1 trn1 nid2 trn2)
definition *syncV nid1 sec1 nid2 sec2* =
($\exists$ *trn1 trn2*. *validTrans nid1 trn1* $\wedge$ *lreach nid1* (*srcOf nid1 trn1*) $\wedge$ φ *nid1 trn1* $\wedge$ *f nid1 trn1* = *sec1* $\wedge$
validTrans nid2 trn2 $\wedge$ *lreach nid2* (*srcOf nid2 trn2*) $\wedge$ φ *nid2 trn2* $\wedge$ *f nid2 trn2* = *sec2* $\wedge$
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 φ *nid* (*invV nid sec*) **and** *f nid* (*invV nid 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 γ *nid* (*invO nid obs*) **and** *g nid* (*invO nid 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-φ1-φ2* **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** φ *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** *lreach nid* (*srcOf nid trn*) **and** φ *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** *lreach nid* (*srcOf nid trn*) **and** φ *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* ⇒ *bool* **where**
secComp [] = *True*
| *secComp* (*LVal nid s* # *sl*) =
(*secComp sl* ∧ *nid* ∈ *nodes* ∧
¬(∃ *trn*. *validTrans nid trn* ∧ *lreach nid* (*srcOf nid trn*) ∧ φ *nid trn* ∧ *f nid trn* = *s* ∧
(*comOf nid trn* = *Send* ∨ *comOf nid trn* = *Recv*) ∧ *tgtNodeOf nid trn* ∈ *nodes*))
| *secComp* (*CVal nid1 s1 nid2 s2* # *sl*) =
(*secComp sl* ∧ *nid1* ∈ *nodes* ∧ *nid2* ∈ *nodes* ∧ *nid1* ≠ *nid2* ∧
(∃ *trn1 trn2*. *validTrans nid1 trn1* ∧ *lreach nid1* (*srcOf nid1 trn1*) ∧ φ *nid1 trn1* ∧ *f nid1 trn1* = *s1* ∧
validTrans nid2 trn2 ∧ *lreach nid2* (*srcOf nid2 trn2*) ∧ φ *nid2 trn2* ∧ *f nid2 trn2* = *s2* ∧
comOf nid1 trn1 = *Send* ∧ *tgtNodeOf nid1 trn1* = *nid2* ∧
comOf nid2 trn2 = *Recv* ∧ *tgtNodeOf nid2 trn2* = *nid1* ∧
sync nid1 trn1 nid2 trn2))

lemma *syncedSecs-iff-nValidV*: *secComp sl* ⟷ *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*) =
(∃ *trn1 trn2*. *nid1* ∈ *nodes* ∧ *nid2* ∈ *nodes* ∧ *nid1* ≠ *nid2* ∧
validTrans nid1 trn1 ∧ *lreach nid1* (*srcOf nid1 trn1*) ∧ φ *nid1 trn1* ∧ *f nid1 trn1* = *v1* ∧
validTrans nid2 trn2 ∧ *lreach nid2* (*srcOf nid2 trn2*) ∧ φ *nid2 trn2* ∧ *f nid2 trn2* = *v2* ∧
comOf nid1 trn1 = *Send* ∧ *tgtNodeOf nid1 trn1* = *nid2* ∧
comOf nid2 trn2 = *Recv* ∧ *tgtNodeOf nid2 trn2* = *nid1* ∧

```
        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 ⟷ (∀ nid ∈ nodes. B nid (decompV sl nid) (decompV sl1 nid)) ∧
                         (secComp sl ⟶ 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 ⇒ bool
and
  V1 :: 'traces ⇒ 'values1
and
  O1 :: 'traces ⇒ 'observations1
and
  B1 :: 'values1 ⇒ 'values1 ⇒ bool
and
  TT1 :: 'traces ⇒ bool
and
  V2 :: 'traces ⇒ 'values2
and
  O2 :: 'traces ⇒ 'observations2
and
  B2 :: 'values2 ⇒ 'values2 ⇒ bool
and
  TT2 :: 'traces ⇒ bool
+
fixes
  O :: 'traces ⇒ 'observations
assumes
```

O1-O: *O1 tr* = *O1 tr′* $\Longrightarrow$ *validSystemTrace tr* $\Longrightarrow$ *validSystemTrace tr′* $\Longrightarrow$ *O tr* = *O tr′*
and
O2-O: *O2 tr* = *O2 tr′* $\Longrightarrow$ *validSystemTrace tr* $\Longrightarrow$ *validSystemTrace tr′* $\Longrightarrow$ *O tr* = *O tr′*
and
O1-V2: *O1 tr* = *O1 tr′* $\Longrightarrow$ *validSystemTrace tr* $\Longrightarrow$ *validSystemTrace tr′* $\Longrightarrow$ *B1* (*V1 tr*) (*V1 tr′*) $\Longrightarrow$ *V2 tr* = *V2 tr′*
and
O2-V1: *O2 tr* = *O2 tr′* $\Longrightarrow$ *validSystemTrace tr* $\Longrightarrow$ *validSystemTrace tr′* $\Longrightarrow$ *B2* (*V2 tr*) (*V2 tr′*) $\Longrightarrow$ *V1 tr* = *V1 tr′*
and
O1-TT2: *O1 tr* = *O1 tr′* $\Longrightarrow$ *validSystemTrace tr* $\Longrightarrow$ *validSystemTrace tr′* $\Longrightarrow$ *B1* (*V1 tr*) (*V1 tr′*) $\Longrightarrow$ *TT2 tr* = *TT2 tr′*
begin

definition *V tr* = (*V1 tr*, *V2 tr*)
definition *B vl vl′* = (*B1* (*fst vl*) (*fst vl′*) ∧ *B2* (*snd vl*) (*snd vl′*))
definition *TT tr* = (*TT1 tr* ∧ *TT2 tr*)

sublocale *Abstract-BD-Security 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*: *validSystemTrace tr* **and** *TT*: *TT tr* **and** *B*: *B vl vl′* **and** *V-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′*: *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′′*: *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 ∃ *tr1*. *validSystemTrace tr1* ∧ *O tr1* = *O tr* ∧ *V tr1* = *vl′*
    unfolding *V-def V1 vl′ O′* **by** *auto*
qed

end

locale *BD-Security-TS-Two-Secrets* =
One: *BD-Security-TS istate validTrans srcOf tgtOf* $\varphi 1$ *f1* $\gamma 1$ *g1 T1 B1*
+ *Two*: *BD-Security-TS istate validTrans srcOf tgtOf* $\varphi 2$ *f2* $\gamma 2$ *g2 T2 B2*
for *istate* :: *'state* **and** *validTrans* :: *'trans* ⇒ *bool*
and *srcOf* :: *'trans* ⇒ *'state* **and** *tgtOf* :: *'trans* ⇒ *'state*
and $\varphi 1$ :: *'trans* ⇒ *bool* **and** *f1* :: *'trans* ⇒ *'val1*
and $\gamma 1$ :: *'trans* ⇒ *bool* **and** *g1* :: *'trans* ⇒ *'obs1*
and *T1* :: *'trans* ⇒ *bool* **and** *B1* :: *'val1 list* ⇒ *'val1 list* ⇒ *bool*
and $\varphi 2$ :: *'trans* ⇒ *bool* **and** *f2* :: *'trans* ⇒ *'val2*
and $\gamma 2$ :: *'trans* ⇒ *bool* **and** *g2* :: *'trans* ⇒ *'obs2*
and *T2* :: *'trans* ⇒ *bool* **and** *B2* :: *'val2 list* ⇒ *'val2 list* ⇒ *bool*
+
fixes γ :: *'trans* ⇒ *bool* **and** *g* :: *'trans* ⇒ *'obs*
assumes γ-$\gamma 12$: $\bigwedge tr\ trn.\ One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow \gamma\ trn \Longrightarrow \gamma 1\ trn \wedge \gamma 2\ trn$
and *O1*-γ: $\bigwedge tr\ tr'\ trn\ trn'.\ One.O\ tr = One.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma 1\ trn \Longrightarrow \gamma 1\ trn' \Longrightarrow g1\ trn = g1\ trn' \Longrightarrow \gamma\ trn = \gamma\ trn'$
and *O1-g*: $\bigwedge tr\ tr'\ trn\ trn'.\ One.O\ tr = One.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma 1\ trn \Longrightarrow \gamma 1\ trn' \Longrightarrow g1\ trn = g1\ trn' \Longrightarrow \gamma\ trn \Longrightarrow \gamma\ trn' \Longrightarrow g\ trn = g\ trn'$
and *O2*-γ: $\bigwedge tr\ tr'\ trn\ trn'.\ Two.O\ tr = Two.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma 2\ trn \Longrightarrow \gamma 2\ trn' \Longrightarrow g2\ trn = g2\ trn' \Longrightarrow \gamma\ trn = \gamma\ trn'$
and *O2-g*: $\bigwedge tr\ tr'\ trn\ trn'.\ Two.O\ tr = Two.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma 2\ trn \Longrightarrow \gamma 2\ trn' \Longrightarrow g2\ trn = g2\ trn' \Longrightarrow \gamma\ trn \Longrightarrow \gamma\ trn' \Longrightarrow g\ trn = g\ trn'$
and $\varphi 2$-$\gamma 1$: $\bigwedge tr\ trn.\ One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow \varphi 2\ trn \Longrightarrow \gamma 1\ trn$
and $\gamma 1$-$\varphi 2$: $\bigwedge tr\ tr'\ trn\ trn'.\ One.O\ tr = One.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma 1\ trn \Longrightarrow \gamma 1\ trn' \Longrightarrow g1\ trn = g1\ trn' \Longrightarrow \varphi 2\ trn = \varphi 2\ trn'$
and *g1-f2*: $\bigwedge tr\ tr'\ trn\ trn'.\ One.O\ tr = One.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma 1\ trn \Longrightarrow \gamma 1\ trn' \Longrightarrow g1\ trn = g1\ trn' \Longrightarrow \varphi 2\ trn \Longrightarrow \varphi 2\ trn' \Longrightarrow f2\ trn = f2\ trn'$
and $\varphi 1$-$\gamma 2$: $\bigwedge tr\ trn.\ One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow \varphi 1\ trn \Longrightarrow \gamma 2\ trn$
and $\gamma 2$-$\varphi 1$: $\bigwedge tr\ tr'\ trn\ trn'.\ Two.O\ tr = Two.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma 2\ trn \Longrightarrow \gamma 2\ trn' \Longrightarrow g2\ trn = g2\ trn' \Longrightarrow \varphi 1\ trn = \varphi 1\ trn'$
and *g2-f1*: $\bigwedge tr\ tr'\ trn\ trn'.\ Two.O\ tr = Two.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma 2\ trn \Longrightarrow \gamma 2\ trn' \Longrightarrow g2\ trn = g2\ trn' \Longrightarrow \varphi 1\ trn \Longrightarrow \varphi 1\ trn' \Longrightarrow f1\ trn = f1\ trn'$
and *T2*-$\gamma 1$: $\bigwedge tr\ trn.\ One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow never\ T2\ tr \Longrightarrow T2\ trn \Longrightarrow \gamma 1\ trn$
and $\gamma 1$-*T2*: $\bigwedge tr\ tr'\ trn\ trn'.\ One.O\ tr = One.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma 1\ trn \Longrightarrow \gamma 1\ trn' \Longrightarrow g1\ trn = g1\ trn' \Longrightarrow T2\ trn = T2\ trn'$
begin

definition *O tr* = *map g* (*filter* γ *tr*)

lemma *O-Nil-never*: *O tr* = [] $\longleftrightarrow$ *never* γ *tr* **unfolding** *O-def* **by** (*induction tr*) *auto*
lemma *Nil-O-never*: [] = *O tr* $\longleftrightarrow$ *never* γ *tr* **unfolding** *O-def* **by** (*induction tr*) *auto*
lemma *O-append*: *O* (*tr* @ *tr'*) = *O tr* @ *O tr'* **unfolding** *O-def* **by** *auto*

lemma *never-*γ*12-never-*γ: *One.validFrom istate* (*tr* @ *tr'*) $\Longrightarrow$ *never* γ*1 tr'* $\vee$ *never* γ*2 tr'* $\Longrightarrow$ *never* γ *tr'*
proof (*induction tr' rule*: *rev-induct*)
  case (*snoc trn tr'*)
  then show *?case* **using** γ-γ*12*[*of tr* @ *tr' trn*] **by** (*auto simp*: *One.validFrom-append*)
qed *auto*

lemma *never-*γ*1-never-*φ*2*: *One.validFrom istate* (*tr* @ *tr'*) $\Longrightarrow$ *never* γ*1 tr'* $\Longrightarrow$ *never* φ*2 tr'*
proof (*induction tr' rule*: *rev-induct*)
  case (*snoc trn tr'*)
  then show *?case* **using** φ*2*-γ*1*[*of tr* @ *tr' trn*] **by** (*auto simp*: *One.validFrom-append*)
qed *auto*

lemma *never-*γ*2-never-*φ*1*: *One.validFrom istate* (*tr* @ *tr'*) $\Longrightarrow$ *never* γ*2 tr'* $\Longrightarrow$ *never* φ*1 tr'*
proof (*induction tr' rule*: *rev-induct*)
  case (*snoc trn tr'*)
  then show *?case* **using** φ*1*-γ*2*[*of tr* @ *tr' trn*] **by** (*auto simp*: *One.validFrom-append*)
qed *auto*

lemma *never-*γ*1-never-T2*: *One.validFrom istate* (*tr* @ *tr'*) $\Longrightarrow$ *never T2 tr* $\Longrightarrow$ *never* γ*1 tr'* $\Longrightarrow$ *never T2 tr'*
proof (*induction tr' rule*: *rev-induct*)
  case (*snoc trn tr'*)
  then show *?case* **using** *T2*-γ*1*[*of tr* @ *tr' trn*] **by** (*auto simp*: *One.validFrom-append*)
qed *auto*

sublocale *Abstract-BD-Security-Two-Secrets One.validFrom istate One.V One.O B1 never T1 Two.V Two.O B2 never T2 O*
proof
  fix *tr tr'*
  assume *One.O tr* = *One.O tr' One.validFrom istate tr One.validFrom istate 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*: *O tr* = [] **using** *never-*γ*12-never-*γ[*of* [] *tr*] **by** (*auto simp*: *O-Nil-never One.O-Nil-never*)
     show *O tr* = *O tr'* **using** *Nil never-*γ*12-never-*γ[*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*: γ*1 trn g1 trn* = *obs*
and *tr1*: *One.O tr1* = *obsl* **and** *tr2*: *never* γ*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'*: γ*1 trn' g1 trn'* = *obs*
and *tr1'*: *One.O tr1'* = *obsl* **and** *tr2'*: *never* γ*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-*γ[*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-*γ*12-never-*γ[*of tr1 ## trn tr2*] *never-*γ*12-never-*γ[*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-*γ*12-never-*γ[*of* [] *tr*] **by** (*auto simp*: *O-Nil-never Two.O-Nil-never*)
show *O tr* = *O tr'* **using** *Nil never-*γ*12-never-*γ[*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*: γ*2 trn g2 trn* = *obs*
and *tr1*: *Two.O tr1* = *obsl* **and** *tr2*: *never* γ*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'*: γ*2 trn' g2 trn'* = *obs*
and *tr1'*: *Two.O tr1'* = *obsl* **and** *tr2'*: *never* γ*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-*γ[*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-γ12-never-γ*[*of tr1 ## trn tr2*] *never-γ12-never-γ*[*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-γ1-never-φ2*[*of* [] *tr*]
   unfolding *Two.V-Nil-never One.Nil-O-never* **by** *auto*
  show *Two.V tr* = *Two.V tr′* **using** *never-γ1-never-φ2*[*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 *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** *γ1-φ2*[*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-γ1-never-φ2*[*of tr1 ## trn tr2*] *never-γ1-never-φ2*[*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-γ2-never-φ1*[*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** *γ1-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*) ⟶ *never T2 tr2*
and *never T2* (*tr1′ ## trn′*) ⟶ *never T2 tr2′*
using *never-γ1-never-T2*[*of tr1 ## trn tr2*] *never-γ1-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

References

[1] T. Bauereiß, 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. Bauereiß, 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. Bauereiß, 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.