

# Compositional BD Security

Thomas Bauereiss      Andrei Popescu

February 6, 2026

## Abstract

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

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Binary compositionality theorem</b>	<b>3</b>
<b>3</b>	<b>Trivial security properties</b>	<b>12</b>
<b>4</b>	<b>Transporting BD Security</b>	<b>14</b>
<b>5</b>	<b>N-ary compositionality theorem</b>	<b>18</b>
<b>6</b>	<b>Combining independent secret sources</b>	<b>42</b>

## 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 “ $\sigma$ ”) to range over states.

## 2 Binary compositionality theorem

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

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

```
lemma list2-induct[case-names NilNil Cons1 Cons2]:
assumes NN:  $P \ [] \ []$ 
and CN:  $\bigwedge x \ xs \ ys. P \ xs \ ys \implies P \ (x \ \# \ xs) \ ys$ 
and NC:  $\bigwedge xs \ y \ ys. P \ xs \ ys \implies P \ xs \ (y \ \# \ ys)$ 
shows  $P \ xs \ ys$ 
 $\langle proof \rangle$ 
```

```
lemma list2-induct[case-names NilNil ConsNil NilCons ConsCons]:
assumes NN:  $P \ [] \ []$ 
and CN:  $\bigwedge x \ xs. P \ xs \ [] \implies P \ (x \ \# \ xs) \ []$ 
and NC:  $\bigwedge y \ ys. P \ [] \ ys \implies P \ [] \ (y \ \# \ ys)$ 
and CC:  $\bigwedge x \ xs \ y \ ys.
  P \ xs \ ys \implies
  (\bigwedge ys'. \text{length } ys' \leq \text{Suc } (\text{length } ys) \implies P \ xs \ ys') \implies
  (\bigwedge xs'. \text{length } xs' \leq \text{Suc } (\text{length } xs) \implies P \ xs' \ ys) \implies
  P \ (x \ \# \ xs) \ (y \ \# \ ys)$ 
shows  $P \ xs \ ys$ 
 $\langle proof \rangle$ 
```

```
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  $\Rightarrow$  bool*  
**and**  
*V1 :: 'traces1  $\Rightarrow$  'values1 and O1 :: 'traces1  $\Rightarrow$  'observations1*  
**and**  
*TT1 :: 'traces1  $\Rightarrow$  bool*  
**and**  
*B1 :: 'values1  $\Rightarrow$  'values1  $\Rightarrow$  bool*  
**and**

*validSystemTraces2 :: 'traces2  $\Rightarrow$  bool*  
**and**  
*V2 :: 'traces2  $\Rightarrow$  'values2 and O2 :: 'traces2  $\Rightarrow$  'observations2*  
**and**  
*TT2 :: 'traces2  $\Rightarrow$  bool*  
**and**  
*B2 :: 'values2  $\Rightarrow$  'values2  $\Rightarrow$  bool*  
**and**

*validSystemTraces :: 'traces  $\Rightarrow$  bool*  
**and**  
*V :: 'traces  $\Rightarrow$  'values and O :: 'traces  $\Rightarrow$  'observations*  
**and**  
*TT :: 'traces  $\Rightarrow$  bool*  
**and**  
*B :: 'values  $\Rightarrow$  'values  $\Rightarrow$  bool*

**+**  
**fixes**  
*comp :: 'traces1  $\Rightarrow$  'traces2  $\Rightarrow$  'traces  $\Rightarrow$  bool*  
**and**  
*compO :: 'observations1  $\Rightarrow$  'observations2  $\Rightarrow$  'observations  $\Rightarrow$  bool*  
**and**  
*compV :: 'values1  $\Rightarrow$  'values2  $\Rightarrow$  'values  $\Rightarrow$  bool*

**assumes**  
*validSystemTraces:*  
 $\bigwedge tr. \text{validSystemTraces } tr \implies$   
 $(\exists tr1 tr2. \text{validSystemTraces1 } tr1 \wedge \text{validSystemTraces2 } tr2 \wedge \text{comp } tr1 tr2 tr)$   
**and**  
*V-comp:*  
 $\bigwedge tr1 tr2 tr.$   
 $\text{validSystemTraces1 } tr1 \implies \text{validSystemTraces2 } tr2 \implies \text{comp } tr1 tr2 tr$   
 $\implies \text{compV } (V1 tr1) (V2 tr2) (V tr)$   
**and**  
*O-comp:*  
 $\bigwedge tr1 tr2 tr.$   
 $\text{validSystemTraces1 } tr1 \implies \text{validSystemTraces2 } tr2 \implies \text{comp } tr1 tr2 tr$

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

**abbreviation** *secure* **where** *secure*  $\equiv \text{Comp.secure}$   
**abbreviation** *secure1* **where** *secure1*  $\equiv \text{One.secure}$   
**abbreviation** *secure2* **where** *secure2*  $\equiv \text{Two.secure}$

**theorem** *secure1-secure2-secure:*  
**assumes** *s1: secure1* **and** *s2: secure2*  
**shows** *secure*  
 $\langle \text{proof} \rangle$

**end**

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

**locale** *BD-Security-TS-Comp* =  
*One: BD-Security-TS istate1 validTrans1 srcOf1 tgtOf1  $\varphi_1$  f1  $\gamma_1$  g1 T1 B1 +*  
*Two: BD-Security-TS istate2 validTrans2 srcOf2 tgtOf2  $\varphi_2$  f2  $\gamma_2$  g2 T2 B2*  
**for**  
*istate1 :: 'state1* **and** *validTrans1 :: 'trans1*  $\Rightarrow$  *bool*  
**and**  
*srcOf1 :: 'trans1*  $\Rightarrow$  *'state1* **and** *tgtOf1 :: 'trans1*  $\Rightarrow$  *'state1*  
**and**  
 *$\varphi_1$  :: 'trans1*  $\Rightarrow$  *bool* **and** *f1 :: 'trans1*  $\Rightarrow$  *'value1*  
**and**

$\gamma_1 :: 'trans1 \Rightarrow bool$  **and**  $g_1 :: 'trans1 \Rightarrow 'obs1$   
**and**  
 $T_1 :: 'trans1 \Rightarrow bool$  **and**  $B_1 :: 'value1\ list \Rightarrow 'value1\ list \Rightarrow bool$   
**and**  
 $istate2 :: 'state2$  **and**  $validTrans2 :: 'trans2 \Rightarrow bool$   
**and**  
 $srcOf2 :: 'trans2 \Rightarrow 'state2$  **and**  $tgtOf2 :: 'trans2 \Rightarrow 'state2$   
**and**  
 $\varphi_2 :: 'trans2 \Rightarrow bool$  **and**  $f_2 :: 'trans2 \Rightarrow 'value2$   
**and**  
 $\gamma_2 :: 'trans2 \Rightarrow bool$  **and**  $g_2 :: 'trans2 \Rightarrow 'obs2$   
**and**  
 $T_2 :: 'trans2 \Rightarrow bool$  **and**  $B_2 :: 'value2\ list \Rightarrow 'value2\ list \Rightarrow bool$   
+  
**fixes**  
 $isCom1 :: 'trans1 \Rightarrow bool$  **and**  $isCom2 :: 'trans2 \Rightarrow bool$   
**and**  
 $sync :: 'trans1 \Rightarrow 'trans2 \Rightarrow bool$   
**and**  
 $isComV1 :: 'value1 \Rightarrow bool$  **and**  $isComV2 :: 'value2 \Rightarrow bool$   
**and**  
 $syncV :: 'value1 \Rightarrow 'value2 \Rightarrow bool$   
**and**  
 $isComO1 :: 'obs1 \Rightarrow bool$  **and**  $isComO2 :: 'obs2 \Rightarrow bool$   
**and**  
 $syncO :: 'obs1 \Rightarrow 'obs2 \Rightarrow bool$   
  
**assumes**  
 $isCom1\text{-}isComV1: \bigwedge trn1. validTrans1\ trn1 \Longrightarrow One.reach\ (srcOf1\ trn1) \Longrightarrow$   
 $\varphi_1\ trn1 \Longrightarrow isCom1\ trn1 \longleftrightarrow isComV1\ (f1\ trn1)$   
**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**  
 $sync\text{-}syncO:$

$\wedge$   $trn1\ trn2.$   
 $validTrans1\ trn1 \implies One.reach\ (srcOf1\ trn1) \implies$   
 $validTrans2\ trn2 \implies Two.reach\ (srcOf2\ trn2) \implies$   
 $isCom1\ trn1 \implies isCom2\ trn2 \implies \gamma1\ trn1 \implies \gamma2\ trn2 \implies$   
 $sync\ trn1\ trn2 \implies syncO\ (g1\ trn1)\ (g2\ trn2)$

**and**

*sync- $\varphi1$ - $\varphi2$ :*  
 $\wedge$   $trn1\ trn2.$   
 $validTrans1\ trn1 \implies One.reach\ (srcOf1\ trn1) \implies$   
 $validTrans2\ trn2 \implies Two.reach\ (srcOf2\ trn2) \implies$   
 $isCom1\ trn1 \implies isCom2\ trn2 \implies$   
 $sync\ trn1\ trn2 \implies \varphi1\ trn1 \longleftrightarrow \varphi2\ trn2$

**and**

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

**and**

*isCom1- $\gamma1$ :*  $\wedge$   $trn1.$   $validTrans1\ trn1 \implies One.reach\ (srcOf1\ trn1) \implies isCom1$   
 $trn1 \implies \gamma1\ trn1$

**and**

*isCom2- $\gamma2$ :*  $\wedge$   $trn2.$   $validTrans2\ trn2 \implies Two.reach\ (srcOf2\ trn2) \implies isCom2$   
 $trn2 \implies \gamma2\ trn2$

**and**

*isCom2-V2:*  $\wedge$   $trn2.$   $validTrans2\ trn2 \implies Two.reach\ (srcOf2\ trn2) \implies \varphi2\ trn2$   
 $\implies isCom2\ trn2$

**and**

*Dummy:*  $istate1 = istate1 \wedge srcOf1 = srcOf1 \wedge tgtOf1 = tgtOf1 \wedge T1 = T1 \wedge$   
 $B1 = B1 \wedge$   
 $istate2 = istate2 \wedge srcOf2 = srcOf2 \wedge tgtOf2 = tgtOf2 \wedge T2 = T2 \wedge B2$   
 $= B2$

**begin**

**lemma** *sync- $\gamma1$ - $\gamma2$ :*  
 $\wedge$   $trn1\ trn2.$   
 $validTrans1\ trn1 \implies One.reach\ (srcOf1\ trn1) \implies$   
 $validTrans2\ trn2 \implies Two.reach\ (srcOf2\ trn2) \implies$   
 $isCom1\ trn1 \implies isCom2\ trn2 \implies$   
 $sync\ trn1\ trn2 \implies \gamma1\ trn1 \longleftrightarrow \gamma2\ trn2$   
 {proof}

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

```

fun validTrans :: ('state1, 'trans1, 'state2, 'trans2) ctrans => bool where
  validTrans (Trans1 s2 trn1) = (validTrans1 trn1 & \neg isCom1 trn1)
| validTrans (Trans2 s1 trn2) = (validTrans2 trn2 & \neg 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
| φ (CTrans trn1 trn2) = (φ1 trn1 ∨ φ2 trn2)

```

```

fun f :: ('state1, 'trans1, 'state2, 'trans2) ctrans => ('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 γ :: ('state1, 'trans1, 'state2, 'trans2) ctrans => bool where
  γ (Trans1 s2 trn1) = γ1 trn1
| γ (Trans2 s1 trn2) = γ2 trn2
| γ (CTrans trn1 trn2) = (γ1 trn1 ∨ γ2 trn2)

```

```

fun g :: ('state1, 'trans1, 'state2, 'trans2) ctrans => ('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 => bool
where
  T (Trans1 s2 trn1) = T1 trn1
|
  T (Trans2 s1 trn2) = T2 trn2
|
  T (CTrans trn1 trn2) = (T1 trn1 ∨ T2 trn2)

```

```

inductive compV :: 'value1 list => 'value2 list => ('value1, 'value2) cvalue list =>
bool
where

```

$Nil[intro!,simp]: compV [] [] []$   
 $|Step1[intro]:$   
 $compV vl1 vl2 vl \implies \neg isComV1 v1$   
 $\implies compV (v1 \# vl1) vl2 (Value1 v1 \# vl)$   
 $|Step2[intro]:$   
 $compV vl1 vl2 vl \implies \neg isComV2 v2$   
 $\implies compV vl1 (v2 \# vl2) (Value2 v2 \# vl)$   
 $|Com[intro]:$   
 $compV vl1 vl2 vl \implies isComV1 v1 \implies isComV2 v2 \implies syncV v1 v2$   
 $\implies compV (v1 \# vl1) (v2 \# vl2) (CValue v1 v2 \# vl)$

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

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

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

**and**  $rs2: Two.reach\ s2$

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

**and**  $Step1:$

$\bigwedge vll1\ vll2\ vll\ v1.$

$vl1 = v1 \# vll1 \implies$

$Two.V\ tr2 = vll2 \implies$

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

$compV\ vll1\ vll2\ vll \implies \neg isComV1\ v1 \implies P$

**and**  $Com:$

$\bigwedge vll1\ vll2\ vll\ v1\ v2.$

$vl1 = v1 \# vll1 \implies$

$Two.V\ tr2 = v2 \# vll2 \implies$

$vl = CValue\ v1\ v2 \# vll \implies$

$compV\ vll1\ vll2\ vll \implies$

$isComV1\ v1 \implies isComV2\ v2 \implies syncV\ v1\ v2 \implies P$

**shows**  $P$

$\langle proof \rangle$

**inductive**  $compO :: 'obs1\ list \Rightarrow 'obs2\ list \Rightarrow ('obs1, 'obs2)\ cobs\ list \Rightarrow bool$

**where**

$Nil[intro!,simp]: compO [] [] []$

$|Step1[intro]:$

$compO\ ol1\ ol2\ ol \implies \neg isComO1\ ol1$

$\implies compO\ (ol1 \# ol1)\ ol2\ (Obs1\ ol1 \# ol)$

$|Step2[intro]:$

$compO\ ol1\ ol2\ ol \implies \neg isComO2\ ol2$

$\implies compO\ ol1\ (ol2 \# ol2)\ (Obs2\ ol2 \# ol)$

$|Com[intro]:$

$compO\ ol1\ ol2\ ol \implies isComO1\ ol1 \implies isComO2\ ol2 \implies syncO\ ol1\ ol2$

$\implies compO\ (ol1 \# ol1)\ (ol2 \# ol2)\ (CObs\ ol1\ ol2 \# ol)$

**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**  $comp \equiv ccomp\ ystate1\ ystate2$

**end**

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

**context** *BD-Security-TS-Comp*

**begin**

**lemma** *valid*:

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

**shows**

$\exists tr1\ tr2.$

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

$\langle proof \rangle$

**lemma** *validFrom*:

**assumes** *validFrom* *icstate* *tr*

**shows**  $\exists tr1\ tr2. One.validFrom\ ystate1\ tr1 \wedge Two.validFrom\ ystate2\ tr2 \wedge comp$   
 $tr1\ tr2\ tr$

$\langle proof \rangle$

**lemma** *reach-reach12*:

**assumes** *reach* *s*

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

*<proof>*

**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)*

*<proof>*

**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)*

*<proof>*

**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)*

*<proof>*

**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)*

*<proof>*

**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*

*<proof>*

**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*

*<proof>*

**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'*

*<proof>*

**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**  $\text{compO} \ (One.O \ tr1) \ (Two.O \ tr2) \ obl$   
**shows**  $\exists tr. \text{validFrom} \ (s1,s2) \ tr \wedge O \ tr = obl \wedge V \ tr = vl$   
 $\langle proof \rangle$

**lemma** *pullback-O-V*:  
**assumes**  $One.\text{validFrom} \ ystate1 \ tr1$  **and**  $Two.\text{validFrom} \ ystate2 \ tr2$   
**and**  $\text{compV} \ (One.V \ tr1) \ (Two.V \ tr2) \ vl$   
**and**  $\text{compO} \ (One.O \ tr1) \ (Two.O \ tr2) \ ol$   
**shows**  $\exists tr. \text{validFrom} \ icstate \ tr \wedge O \ tr = ol \wedge V \ tr = vl$   
 $\langle proof \rangle$

**end**

**sublocale** *BD-Security-TS-Comp*  $\subseteq K? : \text{Abstract-BD-Security-Comp}$  **where**  
 $\text{validSystemTraces1} = One.\text{validFrom} \ ystate1$  **and**  $V1 = One.V$  **and**  $O1 = One.O$   
**and**  $TT1 = \text{never} \ T1$  **and**  $B1 = B1$  **and**  
 $\text{validSystemTraces2} = Two.\text{validFrom} \ ystate2$  **and**  $V2 = Two.V$  **and**  $O2 =$   
 $Two.O$   
**and**  $TT2 = \text{never} \ T2$  **and**  $B2 = B2$  **and**  
 $\text{validSystemTraces} = \text{validFrom} \ icstate$  **and**  $V = V$  **and**  $O = O$   
**and**  $TT = \text{never} \ T$  **and**  $B = B$  **and**  
 $\text{comp} = \text{comp}$  **and**  $\text{compO} = \text{compO}$  **and**  $\text{compV} = \text{compV}$   
 $\langle proof \rangle$

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

**theorem**  $\text{secure1} \implies \text{secure2} \implies \text{secure}$   
 $\langle proof \rangle$

**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**  $B\text{-id} :: 'value \Rightarrow 'value \Rightarrow \text{bool}$

**where**  $B\text{-id } vl \ vl1 \equiv (vl1 = vl)$

**context** *Abstract-BD-Security*  
**begin**

**lemma** *B-id-secure*:

**assumes**  $\bigwedge tr \ vl \ vl1. B (V \ tr) \ vl1 \implies \text{validSystemTrace } tr \implies B\text{-id } (V \ tr) \ vl1$   
**shows** *secure*  
*<proof>*

**lemma** *O-const-secure*:

**assumes**  $\bigwedge tr. \text{validSystemTrace } tr \implies O \ tr = ol$   
**and**  $\bigwedge tr \ vl \ vl1. B (V \ tr) \ vl1 \implies \text{validSystemTrace } tr \implies (\exists tr1. \text{validSystemTrace } tr1 \wedge V \ tr1 = vl1)$   
**shows** *secure*  
*<proof>*

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

**definition** *V-compatible* :: *'values*  $\Rightarrow$  *'values*  $\Rightarrow$  *bool* **where**  
 $V\text{-compatible } vl \ vl1 \equiv (\forall obs. OV\text{-compatible } obs \ vl \longrightarrow OV\text{-compatible } obs \ vl1)$

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

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

**lemma** *OV-total-secure*:

**assumes**  $OV: \bigwedge obs \ vl. \text{validObs } obs \implies \text{validVal } vl \implies OV\text{-compatible } obs \ vl$   
 $\implies (\exists tr. \text{validSystemTrace } tr \wedge O \ tr = obs \wedge V \ tr = vl)$   
**and**  $BV: \bigwedge vl \ vl1. B \ vl \ vl1 \implies \text{validVal } vl \implies V\text{-compatible } vl \ vl1 \wedge \text{validVal } vl1$   
**shows** *secure*  
*<proof>*

**lemma** *unconstrained-secure*:

**assumes**  $\bigwedge tr. \text{validSystemTrace } tr$   
**and**  $BV: \bigwedge vl \ vl1. B \ vl \ vl1 \implies \text{validVal } vl \implies V\text{-compatible } vl \ vl1 \wedge \text{validVal } vl1$   
**shows** *secure*  
*<proof>*

**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-vST': validSystemTrace tr  $\Longrightarrow$  validSystemTrace' (translateTrace tr)
and
  vST'-vST: validSystemTrace' tr'  $\Longrightarrow$  ( $\exists$  tr. validSystemTrace tr  $\wedge$  translateTrace tr = tr')
and
  V'-V: validSystemTrace tr  $\Longrightarrow$  V' (translateTrace tr) = translateVal (V tr)
and
  O'-O: validSystemTrace tr  $\Longrightarrow$  O' (translateTrace tr) = translateObs (O tr)

```

**and**  
 $B'-B: B' \text{ vl}' \text{ vl1}' \implies \text{validSystemTrace } tr \implies TT \text{ tr} \implies \text{translateVal } (V \text{ tr}) = \text{vl}'$   
 $\implies (\exists \text{vl1}. \text{translateVal } \text{vl1} = \text{vl1}' \wedge B (V \text{ tr}) \text{ vl1})$

**and**  
 $TT'-TT: TT' (\text{translateTrace } tr) \implies \text{validSystemTrace } tr \implies TT \text{ tr}$   
**begin**

**lemma** *translate-secure*:

**assumes** *Orig.secure*

**shows** *Prime.secure*

*<proof>*

**end**

**locale** *BD-Security-TS-Trans* =

*Orig: BD-Security-TS istate validTrans srcOf tgtOf  $\varphi$  f  $\gamma$  g T B*  
 $+ \text{Prime?}: \text{BD-Security-TS istate}' \text{ validTrans}' \text{ srcOf}' \text{ tgtOf}' \varphi' f' \gamma' g' T' B'$

**for** *istate* :: 'state **and** *validTrans* :: 'trans  $\implies$  bool  
**and** *srcOf* :: 'trans  $\implies$  'state **and** *tgtOf* :: 'trans  $\implies$  'state  
**and**  $\varphi$  :: 'trans  $\implies$  bool **and**  $f$  :: 'trans  $\implies$  'val  
**and**  $\gamma$  :: 'trans  $\implies$  bool **and**  $g$  :: 'trans  $\implies$  'obs  
**and**  $T$  :: 'trans  $\implies$  bool **and**  $B$  :: 'val list  $\implies$  'val list  $\implies$  bool  
**and** *istate'* :: 'state' **and** *validTrans'* :: 'trans'  $\implies$  bool  
**and** *srcOf'* :: 'trans'  $\implies$  'state' **and** *tgtOf'* :: 'trans'  $\implies$  'state'  
**and**  $\varphi'$  :: 'trans'  $\implies$  bool **and**  $f'$  :: 'trans'  $\implies$  'val'  
**and**  $\gamma'$  :: 'trans'  $\implies$  bool **and**  $g'$  :: 'trans'  $\implies$  'obs'  
**and**  $T'$  :: 'trans'  $\implies$  bool **and**  $B'$  :: 'val' list  $\implies$  'val' list  $\implies$  bool

**+**

**fixes**

*translateState* :: 'state  $\implies$  'state'

**and**

*translateTrans* :: 'trans  $\implies$  'trans'

**and**

*translateObs* :: 'obs  $\implies$  'obs' option

**and**

*translateVal* :: 'val  $\implies$  'val' option

**assumes**

*vT-vT'*: *validTrans* *trn*  $\implies$  *Orig.reach* (*srcOf* *trn*)  $\implies$  *validTrans'* (*translateTrans* *trn*)

**and**

*vT'-vT*: *validTrans'* *trn'*  $\implies$  *srcOf'* *trn'* = *translateState* *s*  $\implies$  *Orig.reach* *s*  $\implies$   
 $(\exists \text{trn}. \text{validTrans } \text{trn} \wedge \text{srcOf } \text{trn} = s \wedge \text{translateTrans } \text{trn} = \text{trn}')$

**and**

*srcOf'-srcOf*: *validTrans* *trn*  $\implies$  *Orig.reach* (*srcOf* *trn*)  $\implies$  *srcOf'* (*translateTrans* *trn*) = *translateState* (*srcOf* *trn*)

**and**

*tgtOf'-tgtOf*: *validTrans* *trn*  $\implies$  *Orig.reach* (*srcOf* *trn*)  $\implies$  *tgtOf'* (*translateTrans* *trn*) = *translateState* (*tgtOf* *trn*)

**and**

*istate'-istate: istate' = translateState istate*

**and**

$\gamma'-\gamma: \text{validTrans } trn \implies \text{Orig.reach } (\text{srcOf } trn) \implies \gamma' (\text{translateTrans } trn) \implies \gamma trn \wedge \text{translateObs } (g trn) = \text{Some } (g' (\text{translateTrans } trn))$

**and**

$\gamma'-\gamma': \text{validTrans } trn \implies \text{Orig.reach } (\text{srcOf } trn) \implies \gamma trn \implies \gamma' (\text{translateTrans } trn) \vee \text{translateObs } (g trn) = \text{None}$

**and**

$\varphi'-\varphi: \text{validTrans } trn \implies \text{Orig.reach } (\text{srcOf } trn) \implies \varphi' (\text{translateTrans } trn) \implies \varphi trn \wedge \text{translateVal } (f trn) = \text{Some } (f' (\text{translateTrans } trn))$

**and**

$\varphi'-\varphi': \text{validTrans } trn \implies \text{Orig.reach } (\text{srcOf } trn) \implies \varphi trn \implies \varphi' (\text{translateTrans } trn) \vee \text{translateVal } (f trn) = \text{None}$

**and**

$T-T': T trn \implies \text{validTrans } trn \implies \text{Orig.reach } (\text{srcOf } trn) \implies T' (\text{translateTrans } trn)$

**and**

$B'-B: B' vl' vl1' \implies \text{Orig.validFrom } istate tr \implies \text{never } T tr \implies \text{these } (\text{map } \text{translateVal } (\text{Orig.V } tr)) = vl'$

$\implies (\exists vl1. \text{these } (\text{map } \text{translateVal } vl1) = vl1' \wedge B (\text{Orig.V } tr) vl1)$

**begin**

**definition** *translateTrace* :: 'trans list  $\Rightarrow$  'trans' list

**where** *translateTrace* = map *translateTrans*

**definition** *translateO* :: 'obs list  $\Rightarrow$  'obs' list

**where** *translateO* ol = these (map *translateObs* ol)

**definition** *translateV* :: 'val list  $\Rightarrow$  'val' list

**where** *translateV* vl = these (map *translateVal* vl)

**lemma** *validFrom-validFrom'*:

**assumes** *Orig.validFrom* s tr

**and** *Orig.reach* s

**shows** *Prime.validFrom* (translateState s) (translateTrace tr)

*<proof>*

**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

*<proof>*

**lemma** *V'-V*:

**assumes** *Orig.validFrom* s tr

**and** *Orig.reach* s

**shows** *Prime.V* (translateTrace tr) = translateV (*Orig.V* tr)

*<proof>*

**lemma** *O'-O*:

**assumes** *Orig.validFrom s tr*

**and** *Orig.reach s*

**shows** *Prime.O (translateTrace tr) = translateO (Orig.O tr)*

*<proof>*

**lemma** *TT'-TT*:

**assumes** *never T' (translateTrace tr)*

**and** *Orig.validFrom s tr*

**and** *Orig.reach s*

**shows** *never T tr*

*<proof>*

**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>*

**theorem** *Orig.secure  $\implies$  Prime.secure* *<proof>*

**end**

**locale** *BD-Security-TS-Weaken-Observations =*

*Orig: BD-Security-TS* **where** *g = g for g :: 'trans  $\Rightarrow$  'obs*

**+ fixes** *translateObs :: 'obs  $\Rightarrow$  'obs' option*

**begin**

**definition**  *$\gamma' :: 'trans \Rightarrow bool$*

**where**  *$\gamma' trn \equiv \gamma trn \wedge translateObs (g trn) \neq None$*

**definition**  *$g' :: 'trans \Rightarrow 'obs'$*

**where**  *$g' trn \equiv the (translateObs (g trn))$*

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

*<proof>*

**sublocale** *BD-Security-TS-Trans istate validTrans srcOf tgtOf  $\varphi f \gamma g T B$*

*istate validTrans srcOf tgtOf  $\varphi f \gamma' g' T B$*

*id id translateObs Some*

*<proof>*

**theorem** *Orig.secure  $\implies$  Prime.secure* *<proof>*

**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.
```

```
     $\llbracket$  validTrans nid trn; comOf nid trn = Send  $\vee$  comOf nid trn = Recv;
```

```
    Transition-System.reach (istate nid) (validTrans nid) (srcOf nid) (tgtOf nid)  
    (srcOf nid trn) $\rrbracket$ 
```

```
     $\implies$  tgtNodeOf nid trn  $\neq$  nid
```

```
begin
```

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

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

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

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

**sublocale**  $Transition\text{-}System\ istate\ nValidTrans\ nSrcOf\ nTgtOf\ \langle proof \rangle$

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

**lemma**  $nSrcOfTrFrom\text{-}nSrcOf\text{-}hd:$   
 $tr \neq [] \implies nSrcOfTrFrom\ s\ tr = nSrcOf\ (hd\ tr)$

*<proof>*

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

**lemma** *nTgtOfTrFrom-nTgtOf-last*:  
   $tr \neq [] \implies nTgtOfTrFrom s tr = nTgtOf (last tr)$   
  *<proof>*

**lemma** *reach-lreach*:  
**assumes** *reach s*  
**obtains** *lreach nid (s nid)*  
  *<proof>*

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

**inductive** *comp* :: ('nodeid, 'state) *nstate*  $\Rightarrow$  ('nodeid, 'state, 'trans) *ntrans list*  $\Rightarrow$  *bool*

**where**

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

**abbreviation** *lValidFrom* :: 'nodeid  $\Rightarrow$  'state  $\Rightarrow$  'trans list  $\Rightarrow$  *bool* **where**

*lValidFrom*  $nid \equiv TransitionSystem.validFrom (validTrans nid) (srcOf nid) (tgtOf nid)$

**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*⟩

**lemma** *validFrom-comp*: *validFrom s tr*  $\implies$  *comp s tr*  
⟨*proof*⟩

**lemma** *validFrom-lValidFrom*:  
**assumes** *validFrom s tr*  
**shows** *lValidFrom nid (s nid) (decomp tr nid)*  
⟨*proof*⟩

**lemma** *comp-validFrom*:  
**assumes** *comp s tr* **and**  $\bigwedge$ *nid. lValidFrom nid (s nid) (decomp tr nid)*  
**shows** *validFrom s tr*  
⟨*proof*⟩

**lemma** *validFrom-iff-comp*:  
*validFrom s tr*  $\longleftrightarrow$  *comp s tr*  $\wedge$  ( $\forall$  *nid. lValidFrom nid (s nid) (decomp tr nid))  
⟨*proof*⟩*

**end**

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

**lemma** *nValidTransE*: *nValidTrans trn*  $\implies$  *P* ⟨*proof*⟩  
**lemma** *validE*: *valid tr*  $\implies$  *P* ⟨*proof*⟩  
**lemma** *validFrom-iff-Nil*: *validFrom s tr*  $\longleftrightarrow$  *tr* = [] ⟨*proof*⟩  
**lemma** *reach-istate*: *reach s*  $\implies$  *s = istate* ⟨*proof*⟩

**end**

Definition of n-ary security property composition:

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

**for**  
  *istate* :: ('*nodeid*, '*state*) *nstate* **and** *validTrans* :: '*nodeid*  $\Rightarrow$  '*trans*  $\Rightarrow$  *bool*  
**and**  
  *srcOf* :: '*nodeid*  $\Rightarrow$  '*trans*  $\Rightarrow$  '*state* **and** *tgtOf* :: '*nodeid*  $\Rightarrow$  '*trans*  $\Rightarrow$  '*state*  
**and**  
  *nodes* :: '*nodeid* *set*  
**and**  
  *comOf* :: '*nodeid*  $\Rightarrow$  '*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**  
 $comOfV :: 'nodeid \Rightarrow 'val \Rightarrow com$   
**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**  
 $comOfV-comOf[simp]:$   
 $\bigwedge nid\ trn. \llbracket validTrans\ nid\ trn; lreach\ nid\ (srcOf\ nid\ trn); \varphi\ nid\ trn \rrbracket \Longrightarrow comOfV\ nid\ (f\ nid\ trn) = comOf\ nid\ trn$   
**and**  
 $tgtNodeOfV-tgtNodeOf[simp]:$   
 $\bigwedge nid\ trn. \llbracket validTrans\ nid\ trn; lreach\ nid\ (srcOf\ nid\ trn); \varphi\ nid\ trn; comOf\ nid\ trn = Send \vee comOf\ nid\ trn = Recv \rrbracket$   
 $\Longrightarrow tgtNodeOfV\ nid\ (f\ nid\ trn) = tgtNodeOf\ nid\ trn$   
**and**  
 $comOfO-comOf[simp]:$   
 $\bigwedge nid\ trn. \llbracket validTrans\ nid\ trn; lreach\ nid\ (srcOf\ nid\ trn); \gamma\ nid\ trn \rrbracket \Longrightarrow comOfO\ nid\ (g\ nid\ trn) = comOf\ nid\ trn$   
**and**  
 $tgtNodeOfO-tgtNodeOf[simp]:$   
 $\bigwedge nid\ trn. \llbracket validTrans\ nid\ trn; lreach\ nid\ (srcOf\ nid\ trn); \gamma\ nid\ trn; comOf\ nid\ trn = Send \vee comOf\ nid\ trn = Recv \rrbracket$   
 $\Longrightarrow tgtNodeOfO\ nid\ (g\ nid\ trn) = tgtNodeOf\ nid\ trn$   
**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*:

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

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

**and**

*sync-φ1-φ2*:

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

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

**and**

*sync-φ-γ*:

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

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

**and**

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

**and**

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

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

*nodes*

**begin**

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

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

**fun** *nφ* :: ('nodeid, 'state, 'trans) ntrans  $\Rightarrow$  bool **where**

$n\varphi \text{ (LTrans s nid trn)} = \varphi \text{ nid trn}$

$| n\varphi \text{ (CTrans s nid1 trn1 nid2 trn2)} = (\varphi \text{ nid1 trn1} \vee \varphi \text{ nid2 trn2})$

**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**  
 $n\gamma (LTrans\ s\ nid\ trn) = \gamma\ nid\ trn$   
 $| n\gamma (CTrans\ s\ nid1\ trn1\ nid2\ trn2) = (\gamma\ nid1\ trn1 \vee \gamma\ nid2\ trn2)$

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

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

**lemma** *decompV-subDecompV[simp]*: *nid*  $\in$  *nds*  $\implies$  *decompV* (*subDecompV* *vl* *nds*) *nid* = *decompV* *vl* *nid*  
 <proof>

**sublocale** *BD-Security-TS* *istate* *nValidTrans* *nSrcOf* *nTgtOf* *n $\varphi$*  *n $\psi$*  *n $\gamma$*  *n $T$*  *n $B$*   
 <proof>

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

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

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

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

**lemma** *decompV-decomp*:  
**assumes** *validFrom* *s* *tr*  
**and** *reach* *s*  
**shows** *decompV* (*V* *tr*) *nid* = *lV* *nid* (*decomp* *tr* *nid*)  
 <proof>

**lemma** *decompO-decomp*:  
**assumes** *validFrom* *s* *tr*  
**and** *reach* *s*  
**shows** *decompO* (*O* *tr*) *nid* = *lO* *nid* (*decomp* *tr* *nid*)  
 <proof>

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

*<proof>*

**lemma** *validFrom-nValidV*:  
**assumes** *validFrom s tr*  
**and** *reach s*  
**shows** *list-all nValidV (V tr)*  
*<proof>*

**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* *<proof>*

**lemma** *nValidVE*: *nValidV v  $\implies$  P* *<proof>*  
**lemma** *list-all-nValidV-Nil*: *list-all nValidV vl  $\implies$  vl = []* *<proof>*

**lemma** *trivially-secure*: *secure*  
*<proof>*

**end**

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

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

**sublocale** *Node*: *BD-Security-TS* *istate source validTrans source srcOf source tgtOf*  
*source*

$\varphi$  *source f source  $\gamma$  source g source T source B source*

*<proof>*

**lemma** [*simp*]: *decompV (map (LVal source) vl) source = vl*  
*<proof>*

**lemma** [*simp*]: *list-all nValidV vl'  $\implies$  map (LVal source) (decompV vl' source) =*  
*vl'*  
*<proof>*

**lemma** *Node-validFrom-nValidV*:

*Node.validFrom s tr  $\implies$  Node.reach s  $\implies$  list-all nValidV (map (LVal source)*  
*(Node.V tr))*

*<proof>*

**sublocale** *Trans?*: *BD-Security-TS-Trans*

**where**  $istate = istate\ source$  **and**  $validTrans = validTrans\ source$  **and**  $srcOf = srcOf\ source$  **and**  $tgtOf = tgtOf\ source$   
**and**  $\varphi = \varphi\ source$  **and**  $f = f\ source$  **and**  $\gamma = \gamma\ source$  **and**  $g = g\ source$  **and**  $T = T\ source$  **and**  $B = B\ source$   
**and**  $istate' = istate$  **and**  $validTrans' = nValidTrans$  **and**  $srcOf' = nSrcOf$  **and**  $tgtOf' = nTgtOf$   
**and**  $\varphi' = n\varphi$  **and**  $f' = nf$  **and**  $\gamma' = n\gamma$  **and**  $g' = ng$  **and**  $T' = nT$  **and**  $B' = nB$   
**and**  $translateState = \lambda s. istate(source := s)$   
**and**  $translateTrans = \lambda trn. LTrans(istate(source := srcOf\ source\ trn))\ source\ trn$   
**and**  $translateObs = \lambda obs. Some(LObs\ source\ obs)$   
**and**  $translateVal = Some\ o\ LVal\ source$   
 $\langle proof \rangle$

**end**

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

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

**sublocale** *BD-Security-TS-Network* **where**  $nodes = nodes'$   
 $\langle proof \rangle$

**end**

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

**locale** *BD-Security-TS-Network-New-Node-NoSource* = *Sub: BD-Security-TS-Network*  
**where**  $istate = istate$  **and**  $nodes = nodes$  **and**  $f = f$  **and**  $g = g$   
**for**  $istate :: 'nodeid \Rightarrow 'state$  **and**  $nodes :: 'nodeid\ set$   
**and**  $f :: 'nodeid \Rightarrow 'trans \Rightarrow 'val$  **and**  $g :: 'nodeid \Rightarrow 'trans \Rightarrow 'obs$   
 $+$   
**fixes**  $NID :: 'nodeid$   
**assumes**  $new-node: NID \notin nodes$   
**and**  $no-source: NID \neq source$   
**and**  $\varphi-NID-source$ :  
 $\bigwedge trn. \llbracket validTrans\ NID\ trn; Sub.lreach\ NID\ (srcOf\ NID\ trn); \varphi\ NID\ trn \rrbracket$   
 $\implies Sub.isCom\ NID\ trn \wedge tgtNodeOf\ NID\ trn = source \wedge source \in nodes$   
**begin**

**sublocale** *Node*: *BD-Security-TS* *istate* *NID* *validTrans* *NID* *srcOf* *NID* *tgtOf* *NID*  
 $\varphi$  *NID* *f* *NID*  $\gamma$  *NID* *g* *NID* *T* *NID* *B* *NID*  $\langle \text{proof} \rangle$

**sublocale** *BD-Security-TS-Network-Change-Nodes* **where** *nodes'* = *insert* *NID*  
*nodes*  
 $\langle \text{proof} \rangle$

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

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

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

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

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

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

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

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

```

obs = NID)
| isComO1 - = False

```

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

```

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

```

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

```

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

```

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

```

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

```

```

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

```

```

fun translateObs :: (('nodeid, 'obs) nobs, 'obs) cobs  $\Rightarrow$  ('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 ⇒ (('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 ∈ nodes ∧ nid2 ∈ 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 ⇒ (translateVal (invTranslateVal v)) = v  
⟨proof⟩

**lemma** map-translateVal-invTranslateVal[simp]:  
  list-all nValidV vl ⇒ map (translateVal o invTranslateVal) vl = vl  
⟨proof⟩

**fun** compValidV :: ('nodeid, 'val) nvalue, 'val) cvalue ⇒ bool **where**  
  compValidV (Value1 (LVal nid v)) = (Sub.nValidV (LVal nid v) ∧ (isComV nid v → tgtNodeOfV nid v ≠ 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) ∨ nValidV (CVal NID v2 nid1 v1))

**lemma** nValidV-compValidV: nValidV v ⇒ compValidV (invTranslateVal v)  
⟨proof⟩

**lemma** list-all-nValidV-compValidV: list-all nValidV vl ⇒ list-all compValidV (map invTranslateVal vl)  
⟨proof⟩

**lemma** compValidV-nValidV: compValidV v ⇒ nValidV (translateVal v)  
⟨proof⟩

**lemma** *list-all-compValidV-nValidV*: *list-all compValidV vl  $\implies$  list-all nValidV (map translateVal vl)*  
 <proof>

**lemma** *nValidV-subDecompV*: *list-all nValidV vl  $\implies$  list-all Sub.nValidV (subDecompV vl nodes)*  
 <proof>

**lemma** *validTrans-compValidV*:  
**assumes** *Comp.validTrans trn and Comp.reach (Comp.srcOf trn) and Comp. $\varphi$  trn*  
**shows** *compValidV (Comp.f trn)*  
 <proof>

**lemma** *validFrom-compValidV*: *Comp.reach s  $\implies$  Comp.validFrom s tr  $\implies$  list-all compValidV (Comp.V tr)*  
 <proof>

**lemma** *validFrom-istate-compValidV*: *Comp.validFrom Comp.icstate tr  $\implies$  list-all compValidV (Comp.V tr)*  
 <proof>

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

**sublocale** *Trans?*: *BD-Security-TS-Trans Comp.icstate Comp.validTrans Comp.srcOf Comp.tgtOf*  
*Comp. $\varphi$  Comp.f Comp. $\gamma$  Comp.g Comp.T Comp.B*  
*istate nValidTrans nSrcOf nTgtOf n $\varphi$  n $\gamma$  n $\gamma$  nT nB*  
*translateState translateTrans Some o translateObs Some o translateVal*  
 <proof>

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

**lemma** *secure-new-node*:  
**assumes** *Sub.secure and lsecure NID*  
**shows** *secure*  
 <proof>

**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  $\langle proof \rangle$ 
lemma finite-nodesRight: finite nodesRight  $\langle proof \rangle$ 

```

```

sublocale Left: BD-Security-TS-Network-Change-Nodes where  $nodes' = nodesLeft$ 
   $\langle proof \rangle$ 

```

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
   $\langle proof \rangle$ 

```

**end**

```

context BD-Security-TS-Network
begin

```

Putting it all together:

```

theorem network-secure:
assumes  $\forall nid \in nodes. lsecure\ nid$ 
shows secure
   $\langle proof \rangle$ 

```

**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 tgtNodeOf sync  $\varphi$  f
for istate :: 'nodeid  $\Rightarrow$  'state
and validTrans :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  bool
and srcOf :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'state

```

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

**inductive**  $compSec :: ('nodeid \Rightarrow 'sec\ list) \Rightarrow ('nodeid, 'sec, 'msec)\ merged\ sec\ list \Rightarrow bool$   
**where**

$Nil: compSec (\lambda-. []) []$   
 $| Local: [compSec\ sls\ sl; isComV\ nid\ s \longrightarrow tgtNodeOfV\ nid\ s \notin nodes; nid \in nodes] \implies compSec (sls(nid := s \# sls\ nid)) (LMSec\ nid\ s\ \# sl)$   
 $| Comm: [compSec\ sls\ sl; nid1 \in nodes; nid2 \in nodes; nid1 \neq nid2; comOfV\ nid1\ s1 = Send; tgtNodeOfV\ nid1\ s1 = nid2; comOfV\ nid2\ s2 = Recv; tgtNodeOfV\ nid2\ s2 = nid1; syncV\ nid1\ s1\ nid2\ s2] \implies compSec (sls(nid1 := s1 \# sls\ nid1, nid2 := s2 \# sls\ nid2)) (CMSec (mergeSec\ nid1\ s1\ nid2\ s2) \# sl)$

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

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

**sublocale**  $BD\text{-Security-TS}\ istate\ nValidTrans\ nSrcOf\ nTgtOf\ n\varphi\ nf\ n\gamma\ ng\ nT\ nB$   
 $\langle proof \rangle$

**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**  $decompV\text{-Cons-LVal}: decompV (LVal\ nid\ s \# sl) = (decompV\ sl)(nid := s \# decompV\ sl\ nid)$   
 $\langle proof \rangle$

**lemma**  $decompV\text{-Cons-CVal}: assumes\ nid1 \neq nid2$   
**shows**  $decompV (CVal\ nid1\ s1\ nid2\ s2 \# sl) = (decompV\ sl)(nid1 := s1 \# de-$

*compV sl nid1, nid2 := s2 # decompV sl nid2*  
 ⟨proof⟩

**lemma** *nValidV-compSec*:  
**assumes** *list-all nValidV sl*  
**shows** *compSec (decompV sl) (map translateSec sl)*  
 ⟨proof⟩

**lemma** *compSecE*:  
**assumes** *compSec sls sl*  
**obtains** *sl' where decompV sl' = sls and map translateSec sl' = sl and list-all nValidV sl'*  
 ⟨proof⟩

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

**theorem** *network-secure*:  
**assumes**  $\forall nid \in nodes. lsecure\ nid$   
**shows** *secure*  
 ⟨proof⟩

**end**

**context** *BD-Security-TS-Network*  
**begin**

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

**inductive** *srcSyncV* :: *'nodeid*  $\Rightarrow$  *'val list*  $\Rightarrow$  *'val list*  $\Rightarrow$  *bool*  
**for** *nid* :: *'nodeid*  
**where**  
*Nil*: *srcSyncV nid [] []*  
 | *Other*:  $\llbracket srcSyncV\ nid\ vSrc\ vNode; \neg isComV\ source\ v \vee tgtNodeOfV\ source\ v \neq nid \rrbracket$   
 $\implies srcSyncV\ nid\ (v \# vSrc)\ vNode$   
 | *Send*:  $\llbracket srcSyncV\ nid\ vSrc\ vNode; comOfV\ source\ vSrc = Send; comOfV\ nid\ vNode = Recv; tgtNodeOfV\ source\ vSrc = nid; tgtNodeOfV\ nid\ vNode = source; syncV\ source\ vSrc\ nid\ vNode \rrbracket \implies srcSyncV\ nid\ (vSrc \# vNode)\ (vNode \# vNode)$   
 | *Recv*:  $\llbracket srcSyncV\ nid\ vSrc\ vNode; comOfV\ source\ vSrc = Recv; comOfV\ nid\ vNode = Send; tgtNodeOfV\ source\ vSrc = nid; tgtNodeOfV\ nid\ vNode = source; \rrbracket$

$\llbracket \text{syncV } nid \ vNode \ source \ vSrc \rrbracket \Longrightarrow \text{srcSyncV } nid \ (vSrc \ \# \ vlSrc) \ (vNode \ \# \ vlNode)$

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

**where**

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

**lemma**  $\text{srcSyncV-binSyncV}$ :

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

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

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

$\langle \text{proof} \rangle$

**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**  $\text{BD-Security-TS-Network-Preserve-Source-Security} = \text{Net?} : \text{BD-Security-TS-Network}$   
 $+$

**assumes**  $\text{source-in-nodes}: \text{source} \in \text{nodes}$

**and**  $\text{source-secure}: \text{lsecure } \text{source}$

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

$\llbracket \text{B } \text{source} \ (lV \ \text{source} \ \text{tr}) \ \text{vl1}; \text{srcSyncV } \text{nid} \ (lV \ \text{source} \ \text{tr}) \ \text{vl}';$

$l\text{ValidFrom } \text{source} \ (\text{istate } \text{source}) \ \text{tr}; \text{never } (T \ \text{source}) \ \text{tr};$

$\text{nid} \in \text{nodes}; \text{nid} \neq \text{source} \rrbracket$

$\Longrightarrow (\exists \text{vl1}'. \text{B } \text{nid} \ \text{vl}' \ \text{vl1}' \wedge \text{srcSyncV } \text{nid} \ \text{vl1} \ \text{vl1}')$

**begin**

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

**fun**  $nf'$  **where**

$nf' (LTrans\ s\ nid\ trn) = f\ source\ trn$   
 $| nf' (CTrans\ s\ nid1\ trn1\ nid2\ trn2) = (if\ nid1 = source\ then\ f\ source\ trn1\ else\ f\ source\ trn2)$

**fun**  $translateVal$  **where**

$translateVal (LVal\ nid\ v) = v$   
 $| translateVal (CVal\ nid1\ v1\ nid2\ v2) = (if\ nid1 = source\ then\ v1\ else\ v2)$

**definition**  $isProjectionOf$  **where**

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

**lemma**  $nValidV-tgtNodeOf$ :

**assumes**  $list-all\ nValidV\ vl'$

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

$\langle proof \rangle$

**lemma**  $lValidFrom-source-tgtNodeOfV$ :

**assumes**  $lValidFrom\ source\ s\ tr$

**and**  $lreach\ source\ s$

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

(**is**  $?goal\ tr$ )

$\langle proof \rangle$

**lemma**  $merge-projection$ :

**assumes**  $isProjectionOf\ p\ vl$

**and**  $list-all\ (\lambda v.\ isComV\ source\ v \longrightarrow tgtNodeOfV\ source\ v \neq source)$   $vl$

**obtains**  $vl'$  **where**  $\forall\ nid \in nodes'.\ decompV\ vl'\ nid = p\ nid$

**and**  $decompV\ vl'\ source = vl$

**and**  $map\ translateVal\ vl' = vl$

**and**  $list-all\ nValidV\ vl'$

$\langle proof \rangle$

**lemma**  $translateVal-decompV$ :

**assumes**  $validFrom\ s\ tr$

**and**  $reach\ s$

**shows**  $map\ translateVal\ (V\ tr) = decompV\ (V\ tr)\ source$

$\langle proof \rangle$

**lemma**  $srcSyncV-decompV$ :

**assumes**  $tr: validFrom\ s\ tr$

**and**  $s: reach\ s$

**and**  $nid \in nodes$  **and**  $nid \neq source$

**shows**  $srcSyncV\ nid\ (decompV\ (V\ tr)\ source)\ (decompV\ (V\ tr)\ nid)$   
 $\langle proof \rangle$

**sublocale**  $BD\text{-}Security\text{-}TS\text{-}Trans\ istate\ nValidTrans\ nSrcOf\ nTgtOf\ n\varphi\ nf\ n\gamma\ ng$   
 $nT\ nB$   
 $istate\ nValidTrans\ nSrcOf\ nTgtOf\ n\varphi\ nf'\ n\gamma\ ng\ nT\ B$   
 $source$   
 $id\ id\ Some\ Some\ o\ translateVal$   
 $\langle proof \rangle$

**theorem**  $preserve\text{-}source\text{-}secure$ :  
**assumes**  $\forall\ nid \in nodes'.\ lsecure\ nid$   
**shows**  $secure$   
 $\langle proof \rangle$

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

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

**lemma**  $srcSyncV\text{-}projectSrcV$ :  
**assumes**  $nid \in nodes - \{source\}$   
**shows**  $srcSyncV\ nid\ vlSrc\ vln \longleftrightarrow vln = projectSrcV\ nid\ vlSrc$   
 $\langle proof \rangle$

**end**

**locale**  $BD\text{-}Security\text{-}TS\text{-}Network\text{-}Preserve\text{-}Source\text{-}Security\text{-}getTgtV = Net?: BD\text{-}Security\text{-}TS\text{-}Network\text{-}getTgtV$   
 $+$   
**assumes**  $source\text{-}in\text{-}nodes: source \in nodes$   
**and**  $source\text{-}secure: lsecure\ source$   
**and**  $B\text{-}source\text{-}in\text{-}B\text{-}sinks: \bigwedge\ nid\ tr\ vl\ vl1.$   
 $\llbracket B\ source\ vl\ vl1; vl = lV\ source\ tr; lValidFrom\ source\ (istate\ source)\ tr; never\ (T\ source)\ tr;$   
 $nid \in nodes; nid \neq source \rrbracket$

```

 $\implies B \text{ nid } (\text{projectSrcV nid vl}) (\text{projectSrcV nid vl1})$ 
begin

sublocale BD-Security-TS-Network-Preserve-Source-Security
   $\langle \text{proof} \rangle$ 

end

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

locale BD-Security-TS-Network' = TS-Network istate validTrans srcOf tgtOf nodes
comOf tgtNodeOf sync
for
  istate :: ('nodeid, 'state) nstate and validTrans :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  bool
and
  srcOf :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'state and tgtOf :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'state
and
  nodes :: 'nodeid set
and
  comOf :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  com
and
  tgtNodeOf :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'nodeid
and
  sync :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  bool
+
fixes
   $\varphi$  :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  bool and  $f$  :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'val
and
   $\gamma$  :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  bool and  $g$  :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  'obs
and
   $T$  :: 'nodeid  $\Rightarrow$  'trans  $\Rightarrow$  bool and  $B$  :: 'nodeid  $\Rightarrow$  'val list  $\Rightarrow$  'val list  $\Rightarrow$  bool
and
  source :: 'nodeid
assumes
  g-comOf:  $\bigwedge \text{nid trn1 trn2.}$ 
   $\llbracket \text{validTrans nid trn1; lreach nid (srcOf nid trn1); } \gamma \text{ nid trn1;}$ 
   $\text{validTrans nid trn2; lreach nid (srcOf nid trn2); } \gamma \text{ nid trn2;}$ 
   $g \text{ nid trn2} = g \text{ nid trn1} \rrbracket \implies \text{comOf nid trn2} = \text{comOf nid trn1}$ 
and
  f-comOf:  $\bigwedge \text{nid trn1 trn2.}$ 
   $\llbracket \text{validTrans nid trn1; lreach nid (srcOf nid trn1); } \varphi \text{ nid trn1;}$ 
   $\text{validTrans nid trn2; lreach nid (srcOf nid trn2); } \varphi \text{ nid trn2;}$ 
   $f \text{ nid trn2} = f \text{ nid trn1} \rrbracket \implies \text{comOf nid trn2} = \text{comOf nid trn1}$ 
and
  g-tgtNodeOf:  $\bigwedge \text{nid trn1 trn2.}$ 
   $\llbracket \text{validTrans nid trn1; lreach nid (srcOf nid trn1); } \gamma \text{ nid trn1;}$ 
   $\text{validTrans nid trn2; lreach nid (srcOf nid trn2); } \gamma \text{ nid trn2;}$ 
   $g \text{ nid trn2} = g \text{ nid trn1} \rrbracket \implies \text{tgtNodeOf nid trn2} = \text{tgtNodeOf nid trn1}$ 
and

```

$f\text{-tgtNodeOf}: \bigwedge \text{nid trn1 trn2}.$   
 $\llbracket \text{validTrans nid trn1}; \text{leach nid (srcOf nid trn1)}; \varphi \text{ nid trn1};$   
 $\text{validTrans nid trn2}; \text{leach nid (srcOf nid trn2)}; \varphi \text{ nid trn2};$   
 $f \text{ nid trn2} = f \text{ nid trn1} \rrbracket \implies \text{tgtNodeOf nid trn2} = \text{tgtNodeOf nid trn1}$

**and**

$\text{sync-}\varphi_1\text{-}\varphi_2:$   
 $\bigwedge \text{nid1 trn1 nid2 trn2}.$   
 $\text{validTrans nid1 trn1} \implies \text{leach nid1 (srcOf nid1 trn1)} \implies$   
 $\text{validTrans nid2 trn2} \implies \text{leach nid2 (srcOf nid2 trn2)} \implies$   
 $\text{comOf nid1 trn1} = \text{Send} \implies \text{tgtNodeOf nid1 trn1} = \text{nid2} \implies$   
 $\text{comOf nid2 trn2} = \text{Recv} \implies \text{tgtNodeOf nid2 trn2} = \text{nid1} \implies$   
 $\text{sync nid1 trn1 nid2 trn2} \implies \varphi \text{ nid1 trn1} \longleftrightarrow \varphi \text{ nid2 trn2}$

**and**

$\text{sync-}\varphi\text{-}\gamma:$   
 $\bigwedge \text{nid1 trn1 nid2 trn2}.$   
 $\text{validTrans nid1 trn1} \implies \text{leach nid1 (srcOf nid1 trn1)} \implies$   
 $\text{validTrans nid2 trn2} \implies \text{leach nid2 (srcOf nid2 trn2)} \implies$   
 $\text{comOf nid1 trn1} = \text{Send} \implies \text{tgtNodeOf nid1 trn1} = \text{nid2} \implies$   
 $\text{comOf nid2 trn2} = \text{Recv} \implies \text{tgtNodeOf nid2 trn2} = \text{nid1} \implies$   
 $(\gamma \text{ nid1 trn1} \implies \gamma \text{ nid2 trn2} \implies$   
 $\exists \text{trn1}' \text{trn2}'.$   
 $\text{validTrans nid1 trn1}' \wedge \text{leach nid1 (srcOf nid1 trn1}') \wedge \gamma \text{ nid1 trn1}' \wedge g$   
 $\text{nid1 trn1}' = g \text{ nid1 trn1} \wedge$   
 $\text{validTrans nid2 trn2}' \wedge \text{leach nid2 (srcOf nid2 trn2}') \wedge \gamma \text{ nid2 trn2}' \wedge g$   
 $\text{nid2 trn2}' = g \text{ nid2 trn2} \wedge$   
 $\text{sync nid1 trn1}' \text{nid2 trn2}') \implies$   
 $(\varphi \text{ nid1 trn1} \implies \varphi \text{ nid2 trn2} \implies$   
 $\exists \text{trn1}' \text{trn2}'.$   
 $\text{validTrans nid1 trn1}' \wedge \text{leach nid1 (srcOf nid1 trn1}') \wedge \varphi \text{ nid1 trn1}' \wedge f$   
 $\text{nid1 trn1}' = f \text{ nid1 trn1} \wedge$   
 $\text{validTrans nid2 trn2}' \wedge \text{leach nid2 (srcOf nid2 trn2}') \wedge \varphi \text{ nid2 trn2}' \wedge f$   
 $\text{nid2 trn2}' = f \text{ nid2 trn2} \wedge$   
 $\text{sync nid1 trn1}' \text{nid2 trn2}') \implies$   
 $\implies$   
 $\text{sync nid1 trn1 nid2 trn2}$

**and**

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

**and**

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

**begin**

**definition**  $\text{reachableO nid obs} = (\exists \text{trn}.$   $\text{validTrans nid trn} \wedge \text{leach nid (srcOf nid}$   
 $\text{trn}) \wedge \gamma \text{ nid trn} \wedge g \text{ nid trn} = \text{obs})$

**definition**  $\text{reachableV nid sec} = (\exists \text{trn}.$   $\text{validTrans nid trn} \wedge \text{leach nid (srcOf nid}$   
 $\text{trn}) \wedge \varphi \text{ nid trn} \wedge f \text{ nid trn} = \text{sec})$

**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**  $syncV\ nid1\ sec1\ nid2\ sec2 =$

$(\exists\ trn1\ trn2.\ validTrans\ nid1\ trn1 \wedge lreach\ nid1\ (srcOf\ nid1\ trn1) \wedge \varphi\ nid1\ trn1$   
 $\wedge\ f\ nid1\ trn1 = sec1 \wedge$

$validTrans\ nid2\ trn2 \wedge lreach\ nid2\ (srcOf\ nid2\ trn2) \wedge \varphi\ nid2\ trn2 \wedge$   
 $f\ nid2\ trn2 = sec2 \wedge$

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

**lemmas**  $comp\text{-}O\text{-}V\text{-}defs = comOfO\text{-}def\ tgtNodeOfO\text{-}def\ comOfV\text{-}def\ tgtNodeOfV\text{-}def$   
 $syncO\text{-}def\ syncV\text{-}def$   
 $reachableO\text{-}def\ reachableV\text{-}def$

**lemma**  $reachableV\text{-}D$ :

**assumes**  $reachableV\ nid\ sec$

**shows**  $validTrans\ nid\ (invV\ nid\ sec)$  **and**  $lreach\ nid\ (srcOf\ nid\ (invV\ nid\ sec))$

**and**  $\varphi\ nid\ (invV\ nid\ sec)$  **and**  $f\ nid\ (invV\ nid\ sec) = sec$

$\langle proof \rangle$

**lemma**  $reachableO\text{-}D$ :

**assumes**  $reachableO\ nid\ obs$

**shows**  $validTrans\ nid\ (invO\ nid\ obs)$  **and**  $lreach\ nid\ (srcOf\ nid\ (invO\ nid\ obs))$

**and**  $\gamma\ nid\ (invO\ nid\ obs)$  **and**  $g\ nid\ (invO\ nid\ obs) = obs$

$\langle proof \rangle$

**sublocale**  $BD\text{-}Security\text{-}TS\text{-}Network$

**where**  $comOfV = comOfV$  **and**  $tgtNodeOfV = tgtNodeOfV$  **and**  $syncV = syncV$

**and**  $comOfO = comOfO$  **and**  $tgtNodeOfO = tgtNodeOfO$  **and**  $syncO = syncO$

$\langle proof \rangle$

**lemma**  $comOf\text{-}invV$ :

**assumes**  $validTrans\ nid\ trn$  **and**  $lreach\ nid\ (srcOf\ nid\ trn)$  **and**  $\varphi\ nid\ trn$

**shows**  $comOf\ nid\ (invV\ nid\ (f\ nid\ trn)) = comOf\ nid\ trn$

*<proof>*

**lemma** *comOfV-SendE*:

**assumes** *comOfV* *nid* *v* = *Send*

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

**and** *comOf* *nid* *trn* = *Send*

*<proof>*

**lemma** *comOfV-RecvE*:

**assumes** *comOfV* *nid* *v* = *Recv*

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

**and** *comOf* *nid* *trn* = *Recv*

*<proof>*

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

*secComp* [] = True

| *secComp* (LVal *nid* *s* # *sl*) =

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

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

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

$\in$  *nodes*))

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

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

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

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

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

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

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

*sync* *nid1* *trn1* *nid2* *trn2*))

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

*<proof>*

**lemma** *nB-secComp*:

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

*<proof>*

**end**

**end**

## 6 Combining independent secret sources

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

```

theory Independent-Secrets
imports Bounded-Deducibility-Security.BD-Security-TS
begin

locale Abstract-BD-Security-Two-Secrets =
  One: Abstract-BD-Security validSystemTrace V1 O1 B1 TT1
+ Two: Abstract-BD-Security validSystemTrace V2 O2 B2 TT2
for
  validSystemTrace :: 'traces  $\Rightarrow$  bool
and
  V1 :: 'traces  $\Rightarrow$  'values1
and
  O1 :: 'traces  $\Rightarrow$  'observations1
and
  B1 :: 'values1  $\Rightarrow$  'values1  $\Rightarrow$  bool
and
  TT1 :: 'traces  $\Rightarrow$  bool
and
  V2 :: 'traces  $\Rightarrow$  'values2
and
  O2 :: 'traces  $\Rightarrow$  'observations2
and
  B2 :: 'values2  $\Rightarrow$  'values2  $\Rightarrow$  bool
and
  TT2 :: 'traces  $\Rightarrow$  bool
+
fixes
  O :: 'traces  $\Rightarrow$  'observations
assumes
  O1-O: O1 tr = O1 tr'  $\Longrightarrow$  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**  $B\ vl\ vl' = (B1\ (fst\ vl)\ (fst\ vl') \wedge B2\ (snd\ vl)\ (snd\ vl'))$   
**definition**  $TT\ tr = (TT1\ tr \wedge TT2\ tr)$

**sublocale** *Abstract-BD-Security validSystemTrace*  $V\ O\ B\ TT\ \langle proof \rangle$

**theorem** *two-secure*:  
**assumes** *One.secure* **and** *Two.secure*  
**shows** *secure*  
 $\langle proof \rangle$

**end**

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

*One: BD-Security-TS* *istate validTrans srcOf tgtOf*  $\varphi1\ f1\ \gamma1\ g1\ T1\ B1$   
+ *Two: BD-Security-TS* *istate validTrans srcOf tgtOf*  $\varphi2\ f2\ \gamma2\ g2\ T2\ B2$   
**for** *istate* :: 'state **and** *validTrans* :: 'trans  $\Rightarrow$  bool  
**and** *srcOf* :: 'trans  $\Rightarrow$  'state **and** *tgtOf* :: 'trans  $\Rightarrow$  'state  
**and**  $\varphi1$  :: 'trans  $\Rightarrow$  bool **and**  $f1$  :: 'trans  $\Rightarrow$  'val1  
**and**  $\gamma1$  :: 'trans  $\Rightarrow$  bool **and**  $g1$  :: 'trans  $\Rightarrow$  'obs1  
**and**  $T1$  :: 'trans  $\Rightarrow$  bool **and**  $B1$  :: 'val1 list  $\Rightarrow$  'val1 list  $\Rightarrow$  bool  
**and**  $\varphi2$  :: 'trans  $\Rightarrow$  bool **and**  $f2$  :: 'trans  $\Rightarrow$  'val2  
**and**  $\gamma2$  :: 'trans  $\Rightarrow$  bool **and**  $g2$  :: 'trans  $\Rightarrow$  'obs2  
**and**  $T2$  :: 'trans  $\Rightarrow$  bool **and**  $B2$  :: 'val2 list  $\Rightarrow$  'val2 list  $\Rightarrow$  bool  
+  
**fixes**  $\gamma$  :: 'trans  $\Rightarrow$  bool **and**  $g$  :: 'trans  $\Rightarrow$  'obs  
**assumes**  $\gamma$ - $\gamma12$ :  $\bigwedge tr\ trn. One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow \gamma\ trn \Longrightarrow \gamma1\ trn$   
 $\wedge\ \gamma2\ trn$   
**and**  $O1$ - $\gamma$ :  $\bigwedge tr\ tr'\ trn\ trn'. One.O\ tr = One.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma1\ trn \Longrightarrow \gamma1\ trn' \Longrightarrow g1\ trn = g1\ trn' \Longrightarrow \gamma\ trn = \gamma\ trn'$   
**and**  $O1$ - $g$ :  $\bigwedge tr\ tr'\ trn\ trn'. One.O\ tr = One.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma1\ trn \Longrightarrow \gamma1\ trn' \Longrightarrow g1\ trn = g1\ trn' \Longrightarrow \gamma\ trn \Longrightarrow \gamma\ trn' \Longrightarrow g\ trn = g\ trn'$   
**and**  $O2$ - $\gamma$ :  $\bigwedge tr\ tr'\ trn\ trn'. Two.O\ tr = Two.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma2\ trn \Longrightarrow \gamma2\ trn' \Longrightarrow g2\ trn = g2\ trn' \Longrightarrow \gamma\ trn = \gamma\ trn'$   
**and**  $O2$ - $g$ :  $\bigwedge tr\ tr'\ trn\ trn'. Two.O\ tr = Two.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma2\ trn \Longrightarrow \gamma2\ trn' \Longrightarrow g2\ trn = g2\ trn' \Longrightarrow \gamma\ trn \Longrightarrow \gamma\ trn' \Longrightarrow g\ trn = g\ trn'$   
**and**  $\varphi2$ - $\gamma1$ :  $\bigwedge tr\ trn. One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow \varphi2\ trn \Longrightarrow \gamma1\ trn$   
**and**  $\gamma1$ - $\varphi2$ :  $\bigwedge tr\ tr'\ trn\ trn'. One.O\ tr = One.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma1\ trn \Longrightarrow \gamma1\ trn' \Longrightarrow g1\ trn = g1\ trn' \Longrightarrow \varphi2\ trn = \varphi2\ trn'$   
**and**  $g1$ - $f2$ :  $\bigwedge tr\ tr'\ trn\ trn'. One.O\ tr = One.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma1\ trn \Longrightarrow \gamma1\ trn' \Longrightarrow g1\ trn = g1\ trn' \Longrightarrow \varphi2\ trn \Longrightarrow \varphi2\ trn' \Longrightarrow f2\ trn = f2\ trn'$   
**and**  $\varphi1$ - $\gamma2$ :  $\bigwedge tr\ trn. One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow \varphi1\ trn \Longrightarrow \gamma2\ trn$   
**and**  $\gamma2$ - $\varphi1$ :  $\bigwedge tr\ tr'\ trn\ trn'. Two.O\ tr = Two.O\ tr' \Longrightarrow One.validFrom\ istate\ (tr\ \#\#\ trn) \Longrightarrow One.validFrom\ istate\ (tr'\ \#\#\ trn') \Longrightarrow \gamma2\ trn \Longrightarrow \gamma2\ trn' \Longrightarrow g2\ trn = g2\ trn' \Longrightarrow \varphi1\ trn = \varphi1\ trn'$

$##\ trn) \implies One.validFrom\ ystate\ (tr'\ ##\ trn') \implies \gamma2\ trn \implies \gamma2\ trn' \implies g2$   
 $trn = g2\ trn' \implies \varphi1\ trn = \varphi1\ trn'$   
**and**  $g2-f1: \bigwedge tr\ tr'\ trn\ trn'.\ Two.O\ tr = Two.O\ tr' \implies One.validFrom\ ystate\ (tr$   
 $##\ trn) \implies One.validFrom\ ystate\ (tr'\ ##\ trn') \implies \gamma2\ trn \implies \gamma2\ trn' \implies g2$   
 $trn = g2\ trn' \implies \varphi1\ trn \implies \varphi1\ trn' \implies f1\ trn = f1\ trn'$   
**and**  $T2-\gamma1: \bigwedge tr\ trn.\ One.validFrom\ ystate\ (tr\ ##\ trn) \implies never\ T2\ tr \implies T2$   
 $trn \implies \gamma1\ trn$   
**and**  $\gamma1-T2: \bigwedge tr\ tr'\ trn\ trn'.\ One.O\ tr = One.O\ tr' \implies One.validFrom\ ystate\ (tr$   
 $##\ trn) \implies One.validFrom\ ystate\ (tr'\ ##\ trn') \implies \gamma1\ trn \implies \gamma1\ trn' \implies g1$   
 $trn = g1\ trn' \implies T2\ trn = T2\ trn'$   
**begin**

**definition**  $O\ tr = map\ g\ (filter\ \gamma\ tr)$

**lemma**  $O-Nil-never: O\ tr = [] \iff never\ \gamma\ tr$  *<proof>*

**lemma**  $Nil-O-never: [] = O\ tr \iff never\ \gamma\ tr$  *<proof>*

**lemma**  $O-append: O\ (tr\ @\ tr') = O\ tr\ @\ O\ tr'$  *<proof>*

**lemma**  $never-\gamma12-never-\gamma: One.validFrom\ ystate\ (tr\ @\ tr') \implies never\ \gamma1\ tr' \vee$   
 $never\ \gamma2\ tr' \implies never\ \gamma\ tr'$   
*<proof>*

**lemma**  $never-\gamma1-never-\varphi2: One.validFrom\ ystate\ (tr\ @\ tr') \implies never\ \gamma1\ tr' \implies$   
 $never\ \varphi2\ tr'$   
*<proof>*

**lemma**  $never-\gamma2-never-\varphi1: One.validFrom\ ystate\ (tr\ @\ tr') \implies never\ \gamma2\ tr' \implies$   
 $never\ \varphi1\ tr'$   
*<proof>*

**lemma**  $never-\gamma1-never-T2: One.validFrom\ ystate\ (tr\ @\ tr') \implies never\ T2\ tr \implies$   
 $never\ \gamma1\ tr' \implies never\ T2\ tr'$   
*<proof>*

**sublocale**  $Abstract-BD-Security-Two-Secrets\ One.validFrom\ ystate\ One.V\ One.O$   
 $B1\ never\ T1\ Two.V\ Two.O\ B2\ never\ T2\ O$   
*<proof>*

**end**

**end**

## References

- [1] T. Bauerei, A. Pesenti Gritti, A. Popescu, and F. Raimondi. Cosmed: A confidentiality-verified social media platform. In J. C. Blanchette and S. Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceed-*

- ings, volume 9807 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2016.
- [2] T. Bauerei, A. Pesenti Gritti, A. Popescu, and F. Raimondi. Cosmedis: A distributed social media platform with formally verified confidentiality guarantees. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 729–748. IEEE Computer Society, 2017.
  - [3] T. Bauerei, A. Pesenti Gritti, A. Popescu, and F. Raimondi. Cosmed: A confidentiality-verified social media platform. *J. Autom. Reason.*, 61(1-4):113–139, 2018.
  - [4] T. Bauereiss and A. Popescu. CoSMed: A confidentiality-verified social media platform. In M. Eberl, G. Klein, A. Lochbihler, T. Nipkow, L. Paulson, and R. Thiemann, editors, *Archive of Formal Proofs*, 2021.
  - [5] T. Bauereiss and A. Popescu. CoSMedis: A confidentiality-verified distributed social media platform. In M. Eberl, G. Klein, A. Lochbihler, T. Nipkow, L. Paulson, and R. Thiemann, editors, *Archive of Formal Proofs*, 2021.
  - [6] S. Kanav, P. Lammich, and A. Popescu. A conference management system with verified document confidentiality. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 2014.
  - [7] A. Popescu, T. Bauereiss, and P. Lammich. Bounded-Deducibility security (invited paper). In L. Cohen and C. Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPICs*, pages 3:1–3:20. Schloss Dagstuhl - Leibniz-Zentrum fr 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.