Compositional BD Security

Thomas Bauereiss   Andrei Popescu

March 19, 2025

Abstract

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

Contents

1 Introduction

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

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

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

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

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

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

2 Binary compositionality theorem

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

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

lemma *list2-induct*[*case-names NilNil Cons1 Cons2*]:
assumes *NN*: P [] []
and *CN*: $\bigwedge x\ xs\ ys.\ P\ xs\ ys \Longrightarrow P\ (x\ \#\ xs)\ ys$
and *NC*: $\bigwedge xs\ y\ ys.\ P\ xs\ ys \Longrightarrow P\ xs\ (y\ \#\ ys)$
shows $P\ xs\ ys$
⟨*proof*⟩

lemma *list22-induct*[*case-names NilNil ConsNil NilCons ConsCons*]:
assumes *NN*: P [] []
and *CN*: $\bigwedge x\ xs.\ P\ xs\ [] \Longrightarrow P\ (x\ \#\ xs)\ []$
and *NC*: $\bigwedge y\ ys.\ P\ []\ ys \Longrightarrow P\ []\ (y\ \#\ ys)$
and *CC*: $\bigwedge x\ xs\ y\ ys.$
  $P\ xs\ ys \Longrightarrow$
  $(\bigwedge\ ys'.\ length\ ys' \leq Suc\ (length\ ys) \Longrightarrow P\ xs\ ys') \Longrightarrow$
  $(\bigwedge\ xs'.\ length\ xs' \leq Suc\ (length\ xs) \Longrightarrow P\ xs'\ ys) \Longrightarrow$
  $P\ (x\ \#\ xs)\ (y\ \#\ ys)$
shows $P\ xs\ ys$
⟨*proof*⟩

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

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

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

end

locale *Abstract-BD-Security-Comp* =

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

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

validSystemTraces :: *'traces* ⇒ *bool*
and
V :: *'traces* ⇒ *'values* **and** *O* :: *'traces* ⇒ *'observations*
and
TT :: *'traces* ⇒ *bool*
and
B :: *'values* ⇒ *'values* ⇒ *bool*
+
fixes
comp :: *'traces1* ⇒ *'traces2* ⇒ *'traces* ⇒ *bool*
and
compO :: *'observations1* ⇒ *'observations2* ⇒ *'observations* ⇒ *bool*
and
compV :: *'values1* ⇒ *'values2* ⇒ *'values* ⇒ *bool*
assumes
validSystemTraces:
⋀ *tr*. *validSystemTraces tr* ⟹
(∃ *tr1 tr2*. *validSystemTraces1 tr1* ∧ *validSystemTraces2 tr2* ∧ *comp tr1 tr2 tr*)
and
V-comp:
⋀ *tr1 tr2 tr*.
validSystemTraces1 tr1 ⟹ *validSystemTraces2 tr2* ⟹ *comp tr1 tr2 tr*
⟹ *compV* (*V1 tr1*) (*V2 tr2*) (*V tr*)
and
O-comp:
⋀ *tr1 tr2 tr*.
validSystemTraces1 tr1 ⟹ *validSystemTraces2 tr2* ⟹ *comp tr1 tr2 tr*

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

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

theorem *secure1-secure2-secure*:
assumes *s1*: *secure1* **and** *s2*: *secure2*
shows *secure*
⟨*proof*⟩

end

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

locale *BD-Security-TS-Comp* =
One: *BD-Security-TS istate1 validTrans1 srcOf1 tgtOf1* φ*1 f1* γ*1 g1 T1 B1* +
Two: *BD-Security-TS istate2 validTrans2 srcOf2 tgtOf2* φ*2 f2* γ*2 g2 T2 B2*
for
istate1 :: *'state1* **and** *validTrans1* :: *'trans1* ⇒ *bool*
and
srcOf1 :: *'trans1* ⇒ *'state1* **and** *tgtOf1* :: *'trans1* ⇒ *'state1*
and
φ*1* :: *'trans1* ⇒ *bool* **and** *f1* :: *'trans1* ⇒ *'value1*
and

$\gamma 1$ :: *'trans1* ⇒ *bool* **and** *g1* :: *'trans1* ⇒ *'obs1*
and
T1 :: *'trans1* ⇒ *bool* **and** *B1* :: *'value1 list* ⇒ *'value1 list* ⇒ *bool*
and

istate2 :: *'state2* **and** *validTrans2* :: *'trans2* ⇒ *bool*
and
srcOf2 :: *'trans2* ⇒ *'state2* **and** *tgtOf2* :: *'trans2* ⇒ *'state2*
and
$\varphi 2$ :: *'trans2* ⇒ *bool* **and** *f2* :: *'trans2* ⇒ *'value2*
and
$\gamma 2$ :: *'trans2* ⇒ *bool* **and** *g2* :: *'trans2* ⇒ *'obs2*
and
T2 :: *'trans2* ⇒ *bool* **and** *B2* :: *'value2 list* ⇒ *'value2 list* ⇒ *bool*
+
fixes
isCom1 :: *'trans1* ⇒ *bool* **and** *isCom2* :: *'trans2* ⇒ *bool*
and
sync :: *'trans1* ⇒ *'trans2* ⇒ *bool*
and
isComV1 :: *'value1* ⇒ *bool* **and** *isComV2* :: *'value2* ⇒ *bool*
and
syncV :: *'value1* ⇒ *'value2* ⇒ *bool*
and
isComO1 :: *'obs1* ⇒ *bool* **and** *isComO2* :: *'obs2* ⇒ *bool*
and
syncO :: *'obs1* ⇒ *'obs2* ⇒ *bool*

assumes
isCom1-isComV1: $\bigwedge$ *trn1*. *validTrans1 trn1* ⟹ *One.reach* (*srcOf1 trn1*) ⟹ $\varphi 1$ *trn1* ⟹ *isCom1 trn1* ⟷ *isComV1* (*f1 trn1*)
and
isCom1-isComO1: $\bigwedge$ *trn1*. *validTrans1 trn1* ⟹ *One.reach* (*srcOf1 trn1*) ⟹ $\gamma 1$ *trn1* ⟹ *isCom1 trn1* ⟷ *isComO1* (*g1 trn1*)
and
isCom2-isComV2: $\bigwedge$ *trn2*. *validTrans2 trn2* ⟹ *Two.reach* (*srcOf2 trn2*) ⟹ $\varphi 2$ *trn2* ⟹ *isCom2 trn2* ⟷ *isComV2* (*f2 trn2*)
and
isCom2-isComO2: $\bigwedge$ *trn2*. *validTrans2 trn2* ⟹ *Two.reach* (*srcOf2 trn2*) ⟹ $\gamma 2$ *trn2* ⟹ *isCom2 trn2* ⟷ *isComO2* (*g2 trn2*)
and
sync-syncV:
$\bigwedge$ *trn1 trn2*.
validTrans1 trn1 ⟹ *One.reach* (*srcOf1 trn1*) ⟹
validTrans2 trn2 ⟹ *Two.reach* (*srcOf2 trn2*) ⟹
isCom1 trn1 ⟹ *isCom2 trn2* ⟹ $\varphi 1$ *trn1* ⟹ $\varphi 2$ *trn2* ⟹
sync trn1 trn2 ⟹ *syncV* (*f1 trn1*) (*f2 trn2*)
and
sync-syncO:

$\bigwedge$ *trn1 trn2.*
validTrans1 trn1 $\Longrightarrow$ *One.reach* (*srcOf1 trn1*) $\Longrightarrow$
validTrans2 trn2 $\Longrightarrow$ *Two.reach* (*srcOf2 trn2*) $\Longrightarrow$
isCom1 trn1 $\Longrightarrow$ *isCom2 trn2* $\Longrightarrow$ γ*1 trn1* $\Longrightarrow$ γ*2 trn2* $\Longrightarrow$
sync trn1 trn2 $\Longrightarrow$ *syncO* (*g1 trn1*) (*g2 trn2*)
and
sync-φ*1-*φ*2*:
$\bigwedge$ *trn1 trn2.*
validTrans1 trn1 $\Longrightarrow$ *One.reach* (*srcOf1 trn1*) $\Longrightarrow$
validTrans2 trn2 $\Longrightarrow$ *Two.reach* (*srcOf2 trn2*) $\Longrightarrow$
isCom1 trn1 $\Longrightarrow$ *isCom2 trn2* $\Longrightarrow$
sync trn1 trn2 $\Longrightarrow$ φ*1 trn1* $\longleftrightarrow$ φ*2 trn2*
and
sync-φ*-*γ:
$\bigwedge$ *trn1 trn2.*
validTrans1 trn1 $\Longrightarrow$ *One.reach* (*srcOf1 trn1*) $\Longrightarrow$
validTrans2 trn2 $\Longrightarrow$ *Two.reach* (*srcOf2 trn2*) $\Longrightarrow$
isCom1 trn1 $\Longrightarrow$ *isCom2 trn2* $\Longrightarrow$
γ*1 trn1* $\Longrightarrow$ γ*2 trn2* $\Longrightarrow$
syncO (*g1 trn1*) (*g2 trn2*) $\Longrightarrow$
(φ*1 trn1* $\Longrightarrow$ φ*2 trn2* $\Longrightarrow$ *syncV* (*f1 trn1*) (*f2 trn2*))
$\Longrightarrow$
sync trn1 trn2
and
isCom1-γ*1*: $\bigwedge$ *trn1. validTrans1 trn1* $\Longrightarrow$ *One.reach* (*srcOf1 trn1*) $\Longrightarrow$ *isCom1 trn1* $\Longrightarrow$ γ*1 trn1*
and
isCom2-γ*2*: $\bigwedge$ *trn2. validTrans2 trn2* $\Longrightarrow$ *Two.reach* (*srcOf2 trn2*) $\Longrightarrow$ *isCom2 trn2* $\Longrightarrow$ γ*2 trn2*
and
isCom2-V2: $\bigwedge$ *trn2. validTrans2 trn2* $\Longrightarrow$ *Two.reach* (*srcOf2 trn2*) $\Longrightarrow$ φ*2 trn2* $\Longrightarrow$ *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-*γ*1-*γ*2*:
$\bigwedge$ *trn1 trn2.*
validTrans1 trn1 $\Longrightarrow$ *One.reach* (*srcOf1 trn1*) $\Longrightarrow$
validTrans2 trn2 $\Longrightarrow$ *Two.reach* (*srcOf2 trn2*) $\Longrightarrow$
isCom1 trn1 $\Longrightarrow$ *isCom2 trn2* $\Longrightarrow$
sync trn1 trn2 $\Longrightarrow$ γ*1 trn1* $\longleftrightarrow$ γ*2 trn2*
$\langle$*proof*$\rangle$

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

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

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

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

fun φ :: ('state1, 'trans1, 'state2, 'trans2) ctrans ⇒ bool where
 φ (Trans1 s2 trn1) = φ1 trn1
|φ (Trans2 s1 trn2) = φ2 trn2
|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 ⟹ ¬ *isComV1 v1*
⟹ *compV* (*v1* # *vl1*) *vl2* (*Value1 v1* # *vl*)
|*Step2* [*intro*]:
compV vl1 vl2 vl ⟹ ¬ *isComV2 v2*
⟹ *compV vl1* (*v2* # *vl2*) (*Value2 v2* # *vl*)
|*Com*[*intro*]:
compV vl1 vl2 vl ⟹ *isComV1 v1* ⟹ *isComV2 v2* ⟹ *syncV v1 v2*
⟹ *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* = [] ⟹ *Two.V tr2* = [] ⟹ *vl* = [] ⟹ *P*
and *Step1*:
⋀*vll1 vll2 vll v1*.
vl1 = *v1* # *vll1* ⟹
Two.V tr2 = *vll2* ⟹
vl = *Value1 v1* # *vll* ⟹
compV vll1 vll2 vll ⟹ ¬ *isComV1 v1* ⟹ *P*
and *Com*:
⋀*vll1 vll2 vll v1 v2*.
vl1 = *v1* # *vll1* ⟹
Two.V tr2 = *v2* # *vll2* ⟹
vl = *CValue v1 v2* # *vll* ⟹
compV vll1 vll2 vll ⟹
isComV1 v1 ⟹ *isComV2 v2* ⟹ *syncV v1 v2* ⟹ *P*
shows *P*
⟨*proof*⟩

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

definition *B* :: ('*value1*,'*value2*) *cvalue list* ⇒ ('*value1*,'*value2*) *cvalue list* ⇒ *bool*
where
B vl vl' ≡ ∀ *vl1 vl2*. *compV vl1 vl2 vl* ⟶

(∃ *vl1' vl2'. compV vl1' vl2' vl' ∧ B1 vl1 vl1' ∧ B2 vl2 vl2'*)

inductive *ccomp* ::
'state1 ⇒ 'state2 ⇒ 'trans1 trace ⇒ 'trans2 trace ⇒
(*'state1*, *'trans1*, *'state2*, *'trans2*) *ctrans trace ⇒ bool*
where
Nil[*simp*,*intro*!]: *ccomp s1 s2* [] [] []
|
Step1[*intro*]:
ccomp (*tgtOf1 trn1*) *s2 tr1 tr2 tr* ⟹ ¬ *isCom1 trn1* ⟹
ccomp (*srcOf1 trn1*) *s2* (*trn1* # *tr1*) *tr2* (*Trans1 s2 trn1* # *tr*)
|
Step2[*intro*]:
ccomp s1 (*tgtOf2 trn2*) *tr1 tr2 tr* ⟹ ¬ *isCom2 trn2* ⟹
ccomp s1 (*srcOf2 trn2*) *tr1* (*trn2* # *tr2*) (*Trans2 s1 trn2* # *tr*)
|
Com[*intro*]:
ccomp (*tgtOf1 trn1*) (*tgtOf2 trn2*) *tr1 tr2 tr* ⟹
isCom1 trn1 ⟹ *isCom2 trn2* ⟹ *sync trn1 trn2* ⟹
ccomp (*srcOf1 trn1*) *s2* (*trn1* # *tr1*) (*trn2* # *tr2*) (*CTrans trn1 trn2* # *tr*)

definition *comp* **where** *comp* ≡ *ccomp istate1 istate2*

end

sublocale *BD-Security-TS-Comp* ⊆ *BD-Security-TS icstate validTrans srcOf tgtOf φ f γ g T B* ⟨*proof*⟩

context *BD-Security-TS-Comp*
begin

lemma *valid*:
assumes *valid tr* **and** *srcOf* (*hd tr*) = (*s1*,*s2*)
shows
∃ *tr1 tr2*.
One.validFrom s1 tr1 ∧ *Two.validFrom s2 tr2* ∧
ccomp s1 s2 tr1 tr2 tr
⟨*proof*⟩

lemma *validFrom*:
assumes *validFrom icstate tr*
shows ∃ *tr1 tr2*. *One.validFrom istate1 tr1* ∧ *Two.validFrom istate2 tr2* ∧ *comp tr1 tr2 tr*
⟨*proof*⟩

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* ∧ *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* ∧ *never T2 tr2*
⟨*proof*⟩

lemma *B*:
assumes *compV vl1 vl2 vl* **and** *B vl vl'*
shows ∃ *vl1' vl2'*. *compV vl1' vl2' vl'* ∧ *B1 vl1 vl1'* ∧ *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 *compO* (*One.O tr1*) (*Two.O tr2*) *obl*
shows ∃ *tr. validFrom* (*s1*,*s2*) *tr* ∧ *O tr* = *obl* ∧ *V tr* = *vl*
⟨*proof*⟩

lemma *pullback-O-V*:
assumes *One.validFrom istate1 tr1* **and** *Two.validFrom istate2 tr2*
and *compV* (*One.V tr1*) (*Two.V tr2*) *vl*
and *compO* (*One.O tr1*) (*Two.O tr2*) *ol*
shows ∃ *tr. validFrom icstate tr* ∧ *O tr* = *ol* ∧ *V tr* = *vl*
⟨*proof*⟩

end

sublocale *BD-Security-TS-Comp* ⊆ *K?* : *Abstract-BD-Security-Comp* **where**
validSystemTraces1 = *One.validFrom istate1* **and** *V1* = *One.V* **and** *O1* = *One.O*
and *TT1* = *never T1* **and** *B1* = *B1* **and**
validSystemTraces2 = *Two.validFrom istate2* **and** *V2* = *Two.V* **and** *O2* = *Two.O*
and *TT2* = *never T2* **and** *B2* = *B2* **and**
validSystemTraces = *validFrom icstate* **and** *V* = *V* **and** *O* = *O*
and *TT* = *never T* **and** *B* = *B* **and**
comp = *comp* **and** *compO* = *compO* **and** *compV* = *compV*
⟨*proof*⟩

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

theorem *secure1* ⟹ *secure2* ⟹ *secure*
⟨*proof*⟩

end

end

3 Trivial security properties

Here we formalize some cases when BD Security holds trivially.

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

definition *B-id* :: *'value* ⇒ *'value* ⇒ *bool*

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

context *Abstract-BD-Security*
begin

lemma *B-id-secure*:
assumes $\bigwedge tr\ vl\ vl1.\ B\ (V\ tr)\ vl1 \Longrightarrow validSystemTrace\ tr \Longrightarrow B\text{-}id\ (V\ tr)\ vl1$
shows *secure*
⟨*proof*⟩

lemma *O-const-secure*:
assumes $\bigwedge tr.\ validSystemTrace\ tr \Longrightarrow O\ tr = ol$
and $\bigwedge tr\ vl\ vl1.\ B\ (V\ tr)\ vl1 \Longrightarrow validSystemTrace\ tr \Longrightarrow (\exists\, tr1.\ validSystemTrace\ tr1 \wedge V\ tr1 = vl1)$
shows *secure*
⟨*proof*⟩

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

definition $V\text{-}compatible :: {}'values \Rightarrow {}'values \Rightarrow bool$ **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**
$validObs\ obs \equiv (\exists\, tr.\ validSystemTrace\ tr \wedge O\ tr = obs)$

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

lemma *OV-total-secure*:
assumes OV: $\bigwedge obs\ vl.\ validObs\ obs \Longrightarrow validVal\ vl \Longrightarrow OV\text{-}compatible\ obs\ vl \Longrightarrow (\exists\, tr.\ validSystemTrace\ tr \wedge O\ tr = obs \wedge V\ tr = vl)$
and BV: $\bigwedge vl\ vl1.\ B\ vl\ vl1 \Longrightarrow validVal\ vl \Longrightarrow V\text{-}compatible\ vl\ vl1 \wedge validVal\ vl1$
shows *secure*
⟨*proof*⟩

lemma *unconstrained-secure*:
assumes $\bigwedge tr.\ validSystemTrace\ tr$
and BV: $\bigwedge vl\ vl1.\ B\ vl\ vl1 \Longrightarrow validVal\ vl \Longrightarrow V\text{-}compatible\ vl\ vl1 \wedge validVal\ vl1$
shows *secure*
⟨*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
  $vST\text{-}vST'$: $validSystemTrace\ tr \Longrightarrow validSystemTrace'\ (translateTrace\ tr)$
and
  $vST'\text{-}vST$: $validSystemTrace'\ tr' \Longrightarrow (\exists\, tr.\ validSystemTrace\ tr \wedge translateTrace\ tr = tr')$
and
  $V'\text{-}V$: $validSystemTrace\ tr \Longrightarrow V'\ (translateTrace\ tr) = translateVal\ (V\ tr)$
and
  $O'\text{-}O$: $validSystemTrace\ tr \Longrightarrow O'\ (translateTrace\ tr) = translateObs\ (O\ tr)$

and
  B'-B: $B'\ vl'\ vl1' \Longrightarrow validSystemTrace\ tr \Longrightarrow TT\ tr \Longrightarrow translateVal\ (V\ tr) =$ vl'
  $\Longrightarrow (\exists\, vl1.\ translateVal\ vl1 = vl1' \wedge B\ (V\ tr)\ vl1)$
and
  TT'-TT: $TT'\ (translateTrace\ tr) \Longrightarrow validSystemTrace\ tr \Longrightarrow TT\ tr$
begin

lemma *translate-secure*:
assumes *Orig.secure*
shows *Prime.secure*
⟨*proof*⟩

end

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

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

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

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

definition *translateV* :: *′val list* ⇒ *′val′ list*
where $translateV\ vl = these\ (map\ translateVal\ vl)$

lemma *validFrom-validFrom′*:
assumes *Orig.validFrom s tr*
and *Orig.reach s*
shows *Prime.validFrom* (*translateState s*) (*translateTrace tr*)
⟨*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 $Prime.V\ (translateTrace\ tr) = translateV\ (Orig.V\ tr)$

$\langle proof \rangle$

lemma *O′-O*:
assumes *Orig.validFrom s tr*
and *Orig.reach s*
shows *Prime.O* (*translateTrace tr*) = *translateO* (*Orig.O tr*)
$\langle proof \rangle$

lemma *TT′-TT*:
assumes *never T′* (*translateTrace tr*)
and *Orig.validFrom s tr*
and *Orig.reach s*
shows *never T tr*
$\langle proof \rangle$

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*
$\langle proof \rangle$

theorem *Orig.secure* $\Longrightarrow$ *Prime.secure* $\langle proof \rangle$

end

locale *BD-Security-TS-Weaken-Observations* =
  Orig: *BD-Security-TS* **where** *g* = *g* **for** *g* :: *′trans* ⇒ *′obs*
\+ **fixes** *translateObs* :: *′obs* ⇒ *′obs′ option*
begin

definition γ' :: *′trans* ⇒ *bool*
where γ' *trn* ≡ γ *trn* ∧ *translateObs* (*g trn*) ≠ *None*

definition *g′* :: *′trans* ⇒ *′obs′*
where *g′ trn* ≡ *the* (*translateObs* (*g trn*))

sublocale *Prime?*: *BD-Security-TS istate validTrans srcOf tgtOf* φ *f* γ' *g′ T B*
$\langle proof \rangle$

sublocale *BD-Security-TS-Trans istate validTrans srcOf tgtOf* φ *f* γ *g T B*
istate validTrans srcOf tgtOf φ *f* γ' *g′ T B*
id id translateObs Some
$\langle proof \rangle$

theorem *Orig.secure* $\Longrightarrow$ *Prime.secure* $\langle proof \rangle$

end

end

5 N-ary compositionality theorem

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

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

Definition of n-ary system composition:

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

locale *TS-Network* =
fixes
istate :: (*'nodeid*, *'state*) *nstate* **and** *validTrans* :: *'nodeid* ⇒ *'trans* ⇒ *bool*
and
srcOf :: *'nodeid* ⇒ *'trans* ⇒ *'state* **and** *tgtOf* :: *'nodeid* ⇒ *'trans* ⇒ *'state*
and
nodes :: *'nodeid set*
and
comOf :: *'nodeid* ⇒ *'trans* ⇒ *com*
and
tgtNodeOf :: *'nodeid* ⇒ *'trans* ⇒ *'nodeid*
and
sync :: *'nodeid* ⇒ *'trans* ⇒ *'nodeid* ⇒ *'trans* ⇒ *bool*
assumes
finite-nodes: *finite nodes*
and
isCom-tgtNodeOf: ⋀*nid trn*.
⟦*validTrans nid trn*; *comOf nid trn* = *Send* ∨ *comOf nid trn* = *Recv*;
Transition-System.reach (*istate nid*) (*validTrans nid*) (*srcOf nid*) (*tgtOf nid*) (*srcOf nid trn*)⟧
⟹ *tgtNodeOf nid trn* ≠ *nid*
begin

abbreviation *isCom* :: *'nodeid* ⇒ *'trans* ⇒ *bool*
where *isCom nid trn* ≡ (*comOf nid trn* = *Send* ∨ *comOf nid trn* = *Recv*) ∧ *tgtNodeOf nid trn* ∈ *nodes*

abbreviation *lreach* :: *'nodeid* ⇒ *'state* ⇒ *bool*
where *lreach nid s* ≡ *Transition-System.reach* (*istate nid*) (*validTrans nid*) (*srcOf nid*) (*tgtOf nid*) *s*

Two types of valid transitions in the network:

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

fun *nValidTrans* :: (*'nodeid*, *'state*, *'trans*) *ntrans* ⇒ *bool* **where**
Local: *nValidTrans* (*LTrans s nid trn*) =
(*validTrans nid trn* ∧ *srcOf nid trn* = *s nid* ∧ *nid* ∈ *nodes* ∧ ¬*isCom nid trn*)
| *Comm*: *nValidTrans* (*CTrans s nid1 trn1 nid2 trn2*) =
(*validTrans nid1 trn1* ∧ *srcOf nid1 trn1* = *s nid1* ∧ *comOf nid1 trn1* = *Send* ∧ *tgtNodeOf nid1 trn1* = *nid2* ∧
validTrans nid2 trn2 ∧ *srcOf nid2 trn2* = *s nid2* ∧ *comOf nid2 trn2* = *Recv* ∧ *tgtNodeOf nid2 trn2* = *nid1* ∧
nid1 ∈ *nodes* ∧ *nid2* ∈ *nodes* ∧ *nid1* ≠ *nid2* ∧
sync nid1 trn1 nid2 trn2)

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

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

sublocale *Transition-System istate nValidTrans nSrcOf nTgtOf* ⟨*proof*⟩

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

lemma *nSrcOfTrFrom-nSrcOf-hd*:
tr ≠ [] ⟹ *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*:
$tr \neq [] \Longrightarrow nTgtOfTrFrom\ s\ tr = nTgtOf\ (last\ tr)$
⟨*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* ⇒ ('*nodeid*, '*state*, '*trans*) *ntrans list* ⇒ *bool*
where
Nil: *comp s* []
| *Local*: ⋀*s trn s' tr nid*.
⟦*comp s tr*; *tgtOf nid trn* = *s nid*; *s'* = *s*(*nid* := *srcOf nid trn*); *nid* ∈ *nodes*; ¬*isCom nid trn*⟧
⟹ *comp s'* (*LTrans s' nid trn* # *tr*)
| *Comm*: ⋀*s trn1 trn2 s' tr nid1 nid2*.
⟦*comp s tr*; *tgtOf nid1 trn1* = *s nid1*; *tgtOf nid2 trn2* = *s nid2*;
s' = *s*(*nid1* := *srcOf nid1 trn1*, *nid2* := *srcOf nid2 trn2*);
nid1 ∈ *nodes*; *nid2* ∈ *nodes*; *nid1* ≠ *nid2*;
comOf nid1 trn1 = *Send*; *tgtNodeOf nid1 trn1* = *nid2*;
comOf nid2 trn2 = *Recv*; *tgtNodeOf nid2 trn2* = *nid1*;
sync nid1 trn1 nid2 trn2⟧
⟹ *comp s'* (*CTrans s' nid1 trn1 nid2 trn2* # *tr*)

abbreviation *lValidFrom* :: '*nodeid* ⇒ '*state* ⇒ '*trans list* ⇒ *bool* **where**
lValidFrom nid ≡ *Transition-System.validFrom* (*validTrans nid*) (*srcOf nid*) (*tgtOf nid*)

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

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

nid
⟨*proof*⟩

lemma *validFrom-comp*: $\mathit{validFrom}\ s\ tr \Longrightarrow \mathit{comp}\ s\ tr$
⟨*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*:
$\mathit{validFrom}\ s\ tr \longleftrightarrow \mathit{comp}\ s\ tr \wedge (\forall\, \mathit{nid}.\ \mathit{lValidFrom}\ \mathit{nid}\ (s\ \mathit{nid})\ (\mathit{decomp}\ tr\ \mathit{nid}))$
⟨*proof*⟩

end

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

lemma *nValidTransE*: $\mathit{nValidTrans}\ trn \Longrightarrow P$ ⟨*proof*⟩
lemma *validE*: $\mathit{valid}\ tr \Longrightarrow P$ ⟨*proof*⟩
lemma *validFrom-iff-Nil*: $\mathit{validFrom}\ s\ tr \longleftrightarrow tr = []$ ⟨*proof*⟩
lemma *reach-istate*: $\mathit{reach}\ s \Longrightarrow s = \mathit{istate}$ ⟨*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* ⇒ '*trans* ⇒ *bool*
and
srcOf :: '*nodeid* ⇒ '*trans* ⇒ '*state* **and** *tgtOf* :: '*nodeid* ⇒ '*trans* ⇒ '*state*
and
nodes :: '*nodeid set*
and
comOf :: '*nodeid* ⇒ '*trans* ⇒ *com*
and
tgtNodeOf :: '*nodeid* ⇒ '*trans* ⇒ '*nodeid*
and
sync :: '*nodeid* ⇒ '*trans* ⇒ '*nodeid* ⇒ '*trans* ⇒ *bool*
+

fixes
φ :: $'nodeid \Rightarrow 'trans \Rightarrow bool$ **and** f :: $'nodeid \Rightarrow 'trans \Rightarrow 'val$
and
γ :: $'nodeid \Rightarrow 'trans \Rightarrow bool$ **and** g :: $'nodeid \Rightarrow 'trans \Rightarrow 'obs$
and
T :: $'nodeid \Rightarrow 'trans \Rightarrow bool$ **and** B :: $'nodeid \Rightarrow 'val\ list \Rightarrow 'val\ list \Rightarrow bool$
and
comOfV :: $'nodeid \Rightarrow 'val \Rightarrow com$
and
tgtNodeOfV :: $'nodeid \Rightarrow 'val \Rightarrow 'nodeid$
and
syncV :: $'nodeid \Rightarrow 'val \Rightarrow 'nodeid \Rightarrow 'val \Rightarrow bool$
and
comOfO :: $'nodeid \Rightarrow 'obs \Rightarrow com$
and
tgtNodeOfO :: $'nodeid \Rightarrow 'obs \Rightarrow 'nodeid$
and
syncO :: $'nodeid \Rightarrow 'obs \Rightarrow 'nodeid \Rightarrow 'obs \Rightarrow bool$

and
source :: $'nodeid$

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

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

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

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

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

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

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

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

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

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

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

definition nB :: ('$nodeid$, 'val) $nvalue\ list$ ⇒ ('$nodeid$, 'val) $nvalue\ list$ ⇒ $bool$ **where**
$nB\ vl\ vl' \equiv (\forall nid \in nodes.\ B\ nid\ (decompV\ vl\ nid)\ (decompV\ vl'\ nid)) \land$
$(list\text{-}all\ nValidV\ vl \longrightarrow list\text{-}all\ nValidV\ vl')$

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

lemma *decompV-subDecompV*[*simp*]: *nid* ∈ *nds* ⟹ *decompV* (*subDecompV vl nds*) *nid* = *decompV vl nid*
⟨*proof*⟩

sublocale *BD-Security-TS istate nValidTrans nSrcOf nTgtOf nφ nf nγ ng nT nB*
⟨*proof*⟩

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

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

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

abbreviation *lsecure* :: '*nodeid* ⇒ *bool* **where**
lsecure nid ≡ *Abstract-BD-Security.secure* (*lValidFrom nid* (*istate nid*)) (*lV nid*) (*lO nid*) (*B nid*) (*lTT nid*)

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

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* ⟹ *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* ⟹ *P* ⟨*proof*⟩
lemma *list-all-nValidV-Nil*: *list-all nValidV vl* ⟹ *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*
φ *source f source* γ *source g source T source B source*
⟨*proof*⟩

lemma [*simp*]: *decompV (map (LVal source) vl) source* = *vl*
⟨*proof*⟩

lemma [*simp*]: *list-all nValidV vl′* ⟹ *map (LVal source) (decompV vl′ source)* = *vl′*
⟨*proof*⟩

lemma *Node-validFrom-nValidV*:
Node.validFrom s tr ⟹ *Node.reach s* ⟹ *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*
⟨*proof*⟩

end

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

locale *BD-Security-TS-Network-Change-Nodes* = *Orig*: *BD-Security-TS-Network* +
fixes *nodes'*
assumes *finite-nodes'*: *finite nodes'*
and φ*-source'*:
$\bigwedge$*nid trn.* ⟦*validTrans nid trn*; *Orig.lreach nid* (*srcOf nid trn*); φ *nid trn*; *nid* ≠ *source*; *nid* ∈ *nodes'*⟧
⟹ *Orig.isCom nid trn* ∧ *tgtNodeOf nid trn* = *source* ∧ *source* ∈ *nodes'*
begin

sublocale *BD-Security-TS-Network* **where** *nodes* = *nodes'*
⟨*proof*⟩

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* ⇒ *'state* **and** *nodes* :: *'nodeid set*
and f :: *'nodeid* ⇒ *'trans* ⇒ *'val* **and** g :: *'nodeid* ⇒ *'trans* ⇒ *'obs*

+
fixes *NID* :: *'nodeid*
assumes *new-node*: *NID* ∉ *nodes*
and *no-source*: *NID* ≠ *source*
and φ*-NID-source*:
$\bigwedge$*trn.* ⟦*validTrans NID trn*; *Sub.lreach NID* (*srcOf NID trn*); φ *NID trn*⟧
⟹ *Sub.isCom NID trn* ∧ *tgtNodeOf NID trn* = *source* ∧ *source* ∈ *nodes*
begin

sublocale *Node*: *BD-Security-TS istate NID validTrans NID srcOf NID tgtOf NID* φ *NID f NID* γ *NID g NID T NID B NID* ⟨*proof*⟩

sublocale *BD-Security-TS-Network-Change-Nodes* **where** *nodes′ = insert NID nodes*
⟨*proof*⟩

fun *isCom1* :: (*′nodeid*,*′state*,*′trans*) *ntrans* ⇒ *bool* **where**
isCom1 (*LTrans s nid trn*) = (*nid* ∈ *nodes* ∧ *isCom nid trn* ∧ *tgtNodeOf nid trn* = *NID*)
| *isCom1* - = *False*

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

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

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

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

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

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

fun *isComO1* :: (*′nodeid*,*′obs*) *nobs* ⇒ *bool* **where**
isComO1 (*LObs nid obs*) = (*nid* ∈ *nodes* ∧ *isComO nid obs* ∧ *tgtNodeOfO nid*

obs = NID)
| *isComO1* - = *False*

definition *isComO2 obs* = (∃ *nid*. *nid* ∈ *nodes* ∧ *isComO NID obs* ∧ *tgtNodeOfO NID obs* = *nid*)

fun *SyncO* :: (*'nodeid*,*'obs*) *nobs* ⇒ *'obs* ⇒ *bool* **where**
SyncO (*LObs nid obs1*) *obs2* = (*tgtNodeOfO nid obs1* = *NID* ∧ *tgtNodeOfO NID obs2* = *nid* ∧
((*syncO nid obs1 NID obs2* ∧ *comOfO nid obs1* = *Send* ∧ *comOfO NID obs2* = *Recv*)
∨ (*syncO NID obs2 nid obs1* ∧ *comOfO NID obs2* = *Send* ∧ *comOfO nid obs1* = *Recv*)))
| *SyncO* - - = *False*

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

sublocale *Comp*: *BD-Security-TS-Comp istate Sub.nValidTrans Sub.nSrcOf Sub.nTgtOf Sub.nφ Sub.nf Sub.nγ Sub.ng Sub.nT Sub.nB*
istate NID validTrans NID srcOf NID tgtOf NID φ NID f NID γ NID g NID T NID B NID
isCom1 isCom2 Sync isComV1 isComV2 SyncV isComO1 isComO2 SyncO
⟨*proof*⟩

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

fun *translateState* :: ((*'nodeid* ⇒ *'state*) × *'state*) ⇒ (*'nodeid* ⇒ *'state*) **where**
translateState (*sSub*, *sNode*) = (*sSub*(*NID* := *sNode*))

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

fun *translateObs* :: ((*'nodeid*, *'obs*) *nobs*, *'obs*) *cobs* ⇒ (*'nodeid*, *'obs*) *nobs* **where**
translateObs (*Obs1 obs*) = *obs*
| *translateObs* (*Obs2 obs*) = (*LObs NID obs*)
| *translateObs* (*cobs.CObs* (*LObs nid1 obs1*) *obs2*) =

(*if comOfO nid1 obs1 = Send then CObs nid1 obs1 NID obs2 else CObs NID obs2 nid1 obs1*)
| *translateObs - = undefined*

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

fun *invTranslateVal* :: ('*nodeid*, '*val*) *nvalue* ⇒ (('*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* $\Longrightarrow$ *list-all nValidV* (*map translateVal vl*)
⟨*proof*⟩

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

lemma *validTrans-compValidV*:
assumes *Comp.validTrans trn* **and** *Comp.reach* (*Comp.srcOf trn*) **and** *Comp.*φ *trn*
shows *compValidV* (*Comp.f trn*)
⟨*proof*⟩

lemma *validFrom-compValidV*: *Comp.reach s* $\Longrightarrow$ *Comp.validFrom s tr* $\Longrightarrow$ *list-all compValidV* (*Comp.V tr*)
⟨*proof*⟩

lemma *validFrom-istate-compValidV*: *Comp.validFrom Comp.icstate tr* $\Longrightarrow$ *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.φ *Comp.f Comp.*γ *Comp.g Comp.T Comp.B*
istate nValidTrans nSrcOf nTgtOf nφ *nf n*γ *ng 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* ∩ *nodesRight* = {}
and
nodes-nodesLeftRight: *nodes* = *nodesLeft* ∪ *nodesRight*
and
no-source-right: *source* ∉ *nodesRight*
begin

lemma *finite-nodesLeft*: *finite nodesLeft* ⟨*proof*⟩
lemma *finite-nodesRight*: *finite nodesRight* ⟨*proof*⟩

sublocale *Left*: *BD-Security-TS-Network-Change-Nodes* **where** *nodes'* = *nodesLeft*
⟨*proof*⟩

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 ∀ *nid* ∈ *nodesRight*. *lsecure nid*
shows *secure*
⟨*proof*⟩

end

context *BD-Security-TS-Network*
begin

Putting it all together:

theorem *network-secure*:
assumes ∀ *nid* ∈ *nodes*. *lsecure nid*
shows *secure*
⟨*proof*⟩

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* φ *f*
for *istate* :: *'nodeid* ⇒ *'state*
and *validTrans* :: *'nodeid* ⇒ *'trans* ⇒ *bool*
and *srcOf* :: *'nodeid* ⇒ *'trans* ⇒ *'state*

and *tgtOf* :: *'nodeid* ⇒ *'trans* ⇒ *'state*
and *nodes* :: *'nodeid set*
and *comOf* :: *'nodeid* ⇒ *'trans* ⇒ *com*
and *tgtNodeOf* :: *'nodeid* ⇒ *'trans* ⇒ *'nodeid*
and *sync* :: *'nodeid* ⇒ *'trans* ⇒ *'nodeid* ⇒ *'trans* ⇒ *bool*
and *φ* :: *'nodeid* ⇒ *'trans* ⇒ *bool*
and *f* :: *'nodeid* ⇒ *'trans* ⇒ *'sec* +
fixes *mergeSec* :: *'nodeid* ⇒ *'sec* ⇒ *'nodeid* ⇒ *'sec* ⇒ *'msec*
begin

inductive *compSec* :: (*'nodeid* ⇒ *'sec list*) ⇒ (*'nodeid*, *'sec*, *'msec*) *merged-sec list* ⇒ *bool*
where
  Nil: *compSec* (λ-. []) []
| *Local*: ⟦*compSec sls sl*; *isComV nid s* ⟶ *tgtNodeOfV nid s* ∉ *nodes*; *nid* ∈ *nodes*⟧
    ⟹ *compSec* (*sls*(*nid* := *s* # *sls nid*)) (*LMSec nid s* # *sl*)
| *Comm*: ⟦*compSec sls sl*; *nid1* ∈ *nodes*; *nid2* ∈ *nodes*; *nid1* ≠ *nid2*;
    comOfV nid1 s1 = *Send*; *tgtNodeOfV nid1 s1* = *nid2*;
    comOfV nid2 s2 = *Recv*; *tgtNodeOfV nid2 s2* = *nid1*;
    syncV nid1 s1 nid2 s2⟧
    ⟹ *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* ⇒ (*'nodeid*, *'sec*, *'msec*) *merged-sec list* ⇒ *bool* **where**
nB sl sl' ≡ ∀ *sls*. *compSec sls sl* ⟶
  (∃ *sls'*. *compSec sls' sl'* ∧ (∀ *nid* ∈ *nodes*. *B nid* (*sls nid*) (*sls' nid*)))

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

sublocale *BD-Security-TS istate nValidTrans nSrcOf nTgtOf nφ nf nγ ng nT nB*
⟨*proof*⟩

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

lemma *decompV-Cons-LVal*: *decompV* (*LVal nid s* # *sl*) = (*decompV sl*)(*nid* := *s* # *decompV sl nid*)
⟨*proof*⟩

lemma *decompV-Cons-CVal*:
assumes *nid1* ≠ *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 ∀ *nid* ∈ *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* ⇒ *′val list* ⇒ *′val list* ⇒ *bool*
for *nid* :: *′nodeid*
where
Nil: *srcSyncV nid* [] []
| *Other*: ⟦*srcSyncV nid vlSrc vlNode*; ¬*isComV source v* ∨ *tgtNodeOfV source v* ≠ *nid*⟧
⟹ *srcSyncV nid* (*v* # *vlSrc*) *vlNode*
| *Send*: ⟦*srcSyncV nid vlSrc vlNode*; *comOfV source vSrc* = *Send*; *comOfV nid vNode* = *Recv*;
tgtNodeOfV source vSrc = *nid*; *tgtNodeOfV nid vNode* = *source*;
syncV source vSrc nid vNode⟧ ⟹ *srcSyncV nid* (*vSrc* # *vlSrc*) (*vNode* # *vlNode*)
| *Recv*: ⟦*srcSyncV nid vlSrc vlNode*; *comOfV source vSrc* = *Recv*; *comOfV nid vNode* = *Send*;
tgtNodeOfV source vSrc = *nid*; *tgtNodeOfV nid vNode* = *source*;

syncV nid vNode source vSrc⟧ ⟹ *srcSyncV nid* (*vSrc* # *vlSrc*) (*vNode* # *vlNode*)

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

inductive *binSyncV* :: *'nodeid* ⇒ *'nodeid* ⇒ *'val list* ⇒ *'val list* ⇒ *bool*
for *nid1 nid2* :: *'nodeid*
where
Nil: *binSyncV nid1 nid2* [] []
| *Int1*: ⟦*binSyncV nid1 nid2 vl1 vl2*; ¬*isComV nid1 v* ∨ *tgtNodeOfV nid1 v* ≠ *nid2*⟧ ⟹ *binSyncV nid1 nid2* (*v* # *vl1*) *vl2*
| *Int2*: ⟦*binSyncV nid1 nid2 vl1 vl2*; ¬*isComV nid2 v* ∨ *tgtNodeOfV nid2 v* ≠ *nid1*⟧ ⟹ *binSyncV nid1 nid2 vl1* (*v* # *vl2*)
| *Send*: ⟦*binSyncV nid1 nid2 vl1 vl2*; *comOfV nid1 v1* = *Send*; *comOfV nid2 v2* = *Recv*;
tgtNodeOfV nid1 v1 = *nid2*; *tgtNodeOfV nid2 v2* = *nid1*;
syncV nid1 v1 nid2 v2⟧ ⟹ *binSyncV nid1 nid2* (*v1* # *vl1*) (*v2* # *vl2*)
| *Recv*: ⟦*binSyncV nid1 nid2 vl1 vl2*; *comOfV nid1 v1* = *Recv*; *comOfV nid2 v2* = *Send*;
tgtNodeOfV nid1 v1 = *nid2*; *tgtNodeOfV nid2 v2* = *nid1*;
syncV nid2 v2 nid1 v1⟧ ⟹ *binSyncV nid1 nid2* (*v1* # *vl1*) (*v2* # *vl2*)

lemma *srcSyncV-binSyncV*:
assumes *source* ∈ *nodes* **and** *nid2* ∈ *nodes*
shows *srcSyncV nid2 vl1 vl2* ⟷ (*binSyncV source nid2 vl1 vl2* ∧
list-all (λ*v*. *isComV nid2 v* ∧ *tgtNodeOfV nid2 v* = *source*) *vl2*)
(**is** *?l* ⟷ *?r*)
⟨*proof*⟩

end

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

locale *BD-Security-TS-Network-Preserve-Source-Security* = *Net?*: *BD-Security-TS-Network* +
assumes *source-in-nodes*: *source* ∈ *nodes*
and *source-secure*: *lsecure source*
and *B-source-in-B-sinks*: ⋀*nid tr vl' vl1*.
⟦*B source* (*lV source tr*) *vl1*; *srcSyncV nid* (*lV source tr*) *vl'*;
lValidFrom source (*istate source*) *tr*; *never* (*T source*) *tr*;
nid ∈ *nodes*; *nid* ≠ *source*⟧
⟹ (∃ *vl1'*. *B nid vl' vl1'* ∧ *srcSyncV nid vl1 vl1'*)

begin

abbreviation *nodes′* ≡ *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 = (∀ *nid* ∈ *nodes′*. *srcSyncV nid vl* (*p nid*))

lemma *nValidV-tgtNodeOf*:
assumes *list-all nValidV vl′*
shows *list-all* (λ*v*. *isComV source v* ⟶ *tgtNodeOfV source v* ≠ *source*) (*decompV vl′ source*)
⟨*proof*⟩

lemma *lValidFrom-source-tgtNodeOfV*:
assumes *lValidFrom source s tr*
and *lreach source s*
shows *list-all* (λ*v*. *isComV source v* ⟶ *tgtNodeOfV source v* ≠ *source*) (*lV source tr*)
(**is** *?goal tr*)
⟨*proof*⟩

lemma *merge-projection*:
assumes *isProjectionOf p vl*
and *list-all* (λ*v*. *isComV source v* ⟶ *tgtNodeOfV source v* ≠ *source*) *vl*
obtains *vl′* **where** ∀ *nid* ∈ *nodes′*. *decompV vl′ nid* = *p nid*
and *decompV vl′ source* = *vl*
and *map translateVal vl′* = *vl*
and *list-all nValidV vl′*
⟨*proof*⟩

lemma *translateVal-decompV*:
assumes *validFrom s tr*
and *reach s*
shows *map translateVal* (*V tr*) = *decompV* (*V tr*) *source*
⟨*proof*⟩

lemma *srcSyncV-decompV*:
assumes *tr*: *validFrom s tr*
and *s*: *reach s*
and *nid* ∈ *nodes* **and** *nid* ≠ *source*

shows *srcSyncV nid (decompV (V tr) source) (decompV (V tr) nid)*
⟨*proof*⟩

sublocale *BD-Security-TS-Trans istate nValidTrans nSrcOf nTgtOf nφ nf nγ ng nT nB*
istate nValidTrans nSrcOf nTgtOf nφ nf′ nγ ng nT B source
id id Some Some o translateVal
⟨*proof*⟩

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

end

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

locale *BD-Security-TS-Network-getTgtV* = *BD-Security-TS-Network* +
fixes *getTgtV*
assumes *getTgtV-Send*: *comOfV source vSrc* = *Send* ⟹ *tgtNodeOfV source vSrc* = *nid* ⟹ *nid* ≠ *source* ⟹ (*syncV source vSrc nid vn* ⟷ *vn* = *getTgtV vSrc*) ∧ *tgtNodeOfV nid* (*getTgtV vSrc*) = *source* ∧ *comOfV nid* (*getTgtV vSrc*) = *Recv*
and *getTgtV-Recv*: *comOfV source vSrc* = *Recv* ⟹ *tgtNodeOfV source vSrc* = *nid* ⟹ *nid* ≠ *source* ⟹ (*syncV nid vn source vSrc* ⟷ *vn* = *getTgtV vSrc*) ∧ *tgtNodeOfV nid* (*getTgtV vSrc*) = *source* ∧ *comOfV nid* (*getTgtV vSrc*) = *Send*
begin

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

lemma *srcSyncV-projectSrcV*:
assumes *nid* ∈ *nodes* − {*source*}
shows *srcSyncV nid vlSrc vln* ⟷ *vln* = *projectSrcV nid vlSrc*
⟨*proof*⟩

end

locale *BD-Security-TS-Network-Preserve-Source-Security-getTgtV* = *Net?*: *BD-Security-TS-Network-getTgtV* +
assumes *source-in-nodes*: *source* ∈ *nodes*
and *source-secure*: *lsecure source*
and *B-source-in-B-sinks*: ⋀*nid tr vl vl1*.
⟦*B source vl vl1*; *vl* = *lV source tr*; *lValidFrom source* (*istate source*) *tr*; *never* (*T source*) *tr*;
nid ∈ *nodes*; *nid* ≠ *source*⟧

$\Longrightarrow$ *B nid* (*projectSrcV nid vl*) (*projectSrcV nid vl1*)
begin

sublocale *BD-Security-TS-Network-Preserve-Source-Security*
⟨*proof*⟩

end

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

locale *BD-Security-TS-Network′* = *TS-Network istate validTrans srcOf tgtOf nodes comOf tgtNodeOf sync*
for
  istate :: (*′nodeid*, *′state*) *nstate* **and** *validTrans* :: *′nodeid* ⇒ *′trans* ⇒ *bool*
 and
  srcOf :: *′nodeid* ⇒ *′trans* ⇒ *′state* **and** *tgtOf* :: *′nodeid* ⇒ *′trans* ⇒ *′state*
 and
  nodes :: *′nodeid set*
 and
  comOf :: *′nodeid* ⇒ *′trans* ⇒ *com*
 and
  tgtNodeOf :: *′nodeid* ⇒ *′trans* ⇒ *′nodeid*
 and
  sync :: *′nodeid* ⇒ *′trans* ⇒ *′nodeid* ⇒ *′trans* ⇒ *bool*
+
fixes
  φ :: *′nodeid* ⇒ *′trans* ⇒ *bool* **and** *f* :: *′nodeid* ⇒ *′trans* ⇒ *′val*
 and
  γ :: *′nodeid* ⇒ *′trans* ⇒ *bool* **and** *g* :: *′nodeid* ⇒ *′trans* ⇒ *′obs*
 and
  T :: *′nodeid* ⇒ *′trans* ⇒ *bool* **and** *B* :: *′nodeid* ⇒ *′val list* ⇒ *′val list* ⇒ *bool*
 and
  source :: *′nodeid*
assumes
 g-comOf: ⋀*nid trn1 trn2*.
  ⟦*validTrans nid trn1*; *lreach nid* (*srcOf nid trn1*); γ *nid trn1*;
   validTrans nid trn2; *lreach nid* (*srcOf nid trn2*); γ *nid trn2*;
   g nid trn2 = *g nid trn1*⟧ $\Longrightarrow$ *comOf nid trn2* = *comOf nid trn1*
and
 f-comOf: ⋀*nid trn1 trn2*.
  ⟦*validTrans nid trn1*; *lreach nid* (*srcOf nid trn1*); φ *nid trn1*;
   validTrans nid trn2; *lreach nid* (*srcOf nid trn2*); φ *nid trn2*;
   f nid trn2 = *f nid trn1*⟧ $\Longrightarrow$ *comOf nid trn2* = *comOf nid trn1*
and
 g-tgtNodeOf: ⋀*nid trn1 trn2*.
  ⟦*validTrans nid trn1*; *lreach nid* (*srcOf nid trn1*); γ *nid trn1*;
   validTrans nid trn2; *lreach nid* (*srcOf nid trn2*); γ *nid trn2*;
   g nid trn2 = *g nid trn1*⟧ $\Longrightarrow$ *tgtNodeOf nid trn2* = *tgtNodeOf nid trn1*
and

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

definition $reachableO\ nid\ obs = (\exists\, trn.\ validTrans\ nid\ trn \wedge lreach\ nid\ (srcOf\ nid\ trn) \wedge \gamma\ nid\ trn \wedge g\ nid\ trn = obs)$
definition $reachableV\ nid\ sec = (\exists\, trn.\ validTrans\ nid\ trn \wedge lreach\ nid\ (srcOf\ nid\ trn) \wedge \varphi\ nid\ trn \wedge f\ nid\ trn = sec)$

definition *invO nid obs* = *inv-into* {*trn. validTrans nid trn* ∧ *lreach nid* (*srcOf nid trn*) ∧ γ *nid trn*} (*g nid*) *obs*
definition *invV nid sec* = *inv-into* {*trn. validTrans nid trn* ∧ *lreach nid* (*srcOf nid trn*) ∧ φ *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* =
(∃ *trn1 trn2*. *validTrans nid1 trn1* ∧ *lreach nid1* (*srcOf nid1 trn1*) ∧ γ *nid1 trn1* ∧ *g nid1 trn1* = *obs1* ∧
validTrans nid2 trn2 ∧ *lreach nid2* (*srcOf nid2 trn2*) ∧ γ *nid2 trn2* ∧ *g nid2 trn2* = *obs2* ∧
sync nid1 trn1 nid2 trn2)
definition *syncV nid1 sec1 nid2 sec2* =
(∃ *trn1 trn2*. *validTrans nid1 trn1* ∧ *lreach nid1* (*srcOf nid1 trn1*) ∧ φ *nid1 trn1* ∧ *f nid1 trn1* = *sec1* ∧
validTrans nid2 trn2 ∧ *lreach nid2* (*srcOf nid2 trn2*) ∧ φ *nid2 trn2* ∧ *f nid2 trn2* = *sec2* ∧
sync nid1 trn1 nid2 trn2)

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

lemma *reachableV-D*:
assumes *reachableV nid sec*
shows *validTrans nid* (*invV nid sec*) **and** *lreach nid* (*srcOf nid* (*invV nid sec*))
and φ *nid* (*invV nid sec*) **and** *f nid* (*invV nid sec*) = *sec*
⟨*proof*⟩

lemma *reachableO-D*:
assumes *reachableO nid obs*
shows *validTrans nid* (*invO nid obs*) **and** *lreach nid* (*srcOf nid* (*invO nid obs*))
and γ *nid* (*invO nid obs*) **and** *g nid* (*invO nid obs*) = *obs*
⟨*proof*⟩

sublocale *BD-Security-TS-Network*
where *comOfV* = *comOfV* **and** *tgtNodeOfV* = *tgtNodeOfV* **and** *syncV* = *syncV*
and *comOfO* = *comOfO* **and** *tgtNodeOfO* = *tgtNodeOfO* **and** *syncO* = *syncO*
⟨*proof*⟩

lemma *comOf-invV*:
assumes *validTrans nid trn* **and** *lreach nid* (*srcOf nid trn*) **and** φ *nid trn*
shows *comOf nid* (*invV nid* (*f nid trn*)) = *comOf nid trn*

⟨*proof*⟩

lemma *comOfV-SendE*:
assumes *comOfV nid v = Send*
obtains *trn* **where** *validTrans nid trn* **and** *lreach nid (srcOf nid trn)* **and** *φ nid trn* **and** *f nid trn = v*
and *comOf nid trn = Send*
⟨*proof*⟩

lemma *comOfV-RecvE*:
assumes *comOfV nid v = Recv*
obtains *trn* **where** *validTrans nid trn* **and** *lreach nid (srcOf nid trn)* **and** *φ nid trn* **and** *f nid trn = v*
and *comOf nid trn = Recv*
⟨*proof*⟩

fun *secComp* :: *('nodeid, 'val) nvalue list ⇒ bool* **where**
secComp [] = True
| *secComp (LVal nid s # sl) =*
(secComp sl ∧ nid ∈ nodes ∧
¬(∃ trn. validTrans nid trn ∧ lreach nid (srcOf nid trn) ∧ φ nid trn ∧ f nid trn = s ∧
(comOf nid trn = Send ∨ comOf nid trn = Recv) ∧ tgtNodeOf nid trn ∈ nodes))
| *secComp (CVal nid1 s1 nid2 s2 # sl) =*
(secComp sl ∧ nid1 ∈ nodes ∧ nid2 ∈ nodes ∧ nid1 ≠ nid2 ∧
(∃ trn1 trn2. validTrans nid1 trn1 ∧ lreach nid1 (srcOf nid1 trn1) ∧ φ nid1 trn1 ∧ f nid1 trn1 = s1 ∧
validTrans nid2 trn2 ∧ lreach nid2 (srcOf nid2 trn2) ∧ φ nid2 trn2 ∧ f nid2 trn2 = s2 ∧
comOf nid1 trn1 = Send ∧ tgtNodeOf nid1 trn1 = nid2 ∧
comOf nid2 trn2 = Recv ∧ tgtNodeOf nid2 trn2 = nid1 ∧
sync nid1 trn1 nid2 trn2))

lemma *syncedSecs-iff-nValidV*: *secComp sl ⟷ list-all nValidV sl*
⟨*proof*⟩

lemma *nB-secComp*:
nB sl sl1 ⟷ (∀ nid ∈ nodes. B nid (decompV sl nid) (decompV sl1 nid)) ∧
(secComp sl ⟶ 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* ⇒ *bool*
and
V1 :: *'traces* ⇒ *'values1*
and
O1 :: *'traces* ⇒ *'observations1*
and
B1 :: *'values1* ⇒ *'values1* ⇒ *bool*
and
TT1 :: *'traces* ⇒ *bool*
and
V2 :: *'traces* ⇒ *'values2*
and
O2 :: *'traces* ⇒ *'observations2*
and
B2 :: *'values2* ⇒ *'values2* ⇒ *bool*
and
TT2 :: *'traces* ⇒ *bool*
+
fixes
O :: *'traces* ⇒ *'observations*
assumes
O1-O: *O1 tr* = *O1 tr'* ⟹ *validSystemTrace tr* ⟹ *validSystemTrace tr'* ⟹ *O tr* = *O tr'*
and
O2-O: *O2 tr* = *O2 tr'* ⟹ *validSystemTrace tr* ⟹ *validSystemTrace tr'* ⟹ *O tr* = *O tr'*
and
O1-V2: *O1 tr* = *O1 tr'* ⟹ *validSystemTrace tr* ⟹ *validSystemTrace tr'* ⟹ *B1* (*V1 tr*) (*V1 tr'*) ⟹ *V2 tr* = *V2 tr'*
and
O2-V1: *O2 tr* = *O2 tr'* ⟹ *validSystemTrace tr* ⟹ *validSystemTrace tr'* ⟹ *B2* (*V2 tr*) (*V2 tr'*) ⟹ *V1 tr* = *V1 tr'*
and
O1-TT2: *O1 tr* = *O1 tr'* ⟹ *validSystemTrace tr* ⟹ *validSystemTrace tr'* ⟹ *B1* (*V1 tr*) (*V1 tr'*) ⟹ *TT2 tr* = *TT2 tr'*
begin

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

sublocale *Abstract-BD-Security validSystemTrace V O B TT* ⟨*proof*⟩

theorem *two-secure*:
assumes *One.secure* **and** *Two.secure*
shows *secure*
⟨*proof*⟩

end

locale *BD-Security-TS-Two-Secrets* =
One: *BD-Security-TS istate validTrans srcOf tgtOf φ1 f1 γ1 g1 T1 B1*
+ *Two*: *BD-Security-TS istate validTrans srcOf tgtOf φ2 f2 γ2 g2 T2 B2*
for *istate* :: ′*state* **and** *validTrans* :: ′*trans* ⇒ *bool*
and *srcOf* :: ′*trans* ⇒ ′*state* **and** *tgtOf* :: ′*trans* ⇒ ′*state*
and *φ1* :: ′*trans* ⇒ *bool* **and** *f1* :: ′*trans* ⇒ ′*val1*
and *γ1* :: ′*trans* ⇒ *bool* **and** *g1* :: ′*trans* ⇒ ′*obs1*
and *T1* :: ′*trans* ⇒ *bool* **and** *B1* :: ′*val1 list* ⇒ ′*val1 list* ⇒ *bool*
and *φ2* :: ′*trans* ⇒ *bool* **and** *f2* :: ′*trans* ⇒ ′*val2*
and *γ2* :: ′*trans* ⇒ *bool* **and** *g2* :: ′*trans* ⇒ ′*obs2*
and *T2* :: ′*trans* ⇒ *bool* **and** *B2* :: ′*val2 list* ⇒ ′*val2 list* ⇒ *bool*
+
fixes *γ* :: ′*trans* ⇒ *bool* **and** *g* :: ′*trans* ⇒ ′*obs*
assumes *γ-γ12*: ⋀*tr trn. One.validFrom istate* (*tr* ## *trn*) ⟹ *γ trn* ⟹ *γ1 trn* ∧ *γ2 trn*
and *O1-γ*: ⋀*tr tr′ trn trn′. One.O tr* = *One.O tr′* ⟹ *One.validFrom istate* (*tr* ## *trn*) ⟹ *One.validFrom istate* (*tr′* ## *trn′*) ⟹ *γ1 trn* ⟹ *γ1 trn′* ⟹ *g1 trn* = *g1 trn′* ⟹ *γ trn* = *γ trn′*
and *O1-g*: ⋀*tr tr′ trn trn′. One.O tr* = *One.O tr′* ⟹ *One.validFrom istate* (*tr* ## *trn*) ⟹ *One.validFrom istate* (*tr′* ## *trn′*) ⟹ *γ1 trn* ⟹ *γ1 trn′* ⟹ *g1 trn* = *g1 trn′* ⟹ *γ trn* ⟹ *γ trn′* ⟹ *g trn* = *g trn′*
and *O2-γ*: ⋀*tr tr′ trn trn′. Two.O tr* = *Two.O tr′* ⟹ *One.validFrom istate* (*tr* ## *trn*) ⟹ *One.validFrom istate* (*tr′* ## *trn′*) ⟹ *γ2 trn* ⟹ *γ2 trn′* ⟹ *g2 trn* = *g2 trn′* ⟹ *γ trn* = *γ trn′*
and *O2-g*: ⋀*tr tr′ trn trn′. Two.O tr* = *Two.O tr′* ⟹ *One.validFrom istate* (*tr* ## *trn*) ⟹ *One.validFrom istate* (*tr′* ## *trn′*) ⟹ *γ2 trn* ⟹ *γ2 trn′* ⟹ *g2 trn* = *g2 trn′* ⟹ *γ trn* ⟹ *γ trn′* ⟹ *g trn* = *g trn′*
and *φ2-γ1*: ⋀*tr trn. One.validFrom istate* (*tr* ## *trn*) ⟹ *φ2 trn* ⟹ *γ1 trn*
and *γ1-φ2*: ⋀*tr tr′ trn trn′. One.O tr* = *One.O tr′* ⟹ *One.validFrom istate* (*tr* ## *trn*) ⟹ *One.validFrom istate* (*tr′* ## *trn′*) ⟹ *γ1 trn* ⟹ *γ1 trn′* ⟹ *g1 trn* = *g1 trn′* ⟹ *φ2 trn* = *φ2 trn′*
and *g1-f2*: ⋀*tr tr′ trn trn′. One.O tr* = *One.O tr′* ⟹ *One.validFrom istate* (*tr* ## *trn*) ⟹ *One.validFrom istate* (*tr′* ## *trn′*) ⟹ *γ1 trn* ⟹ *γ1 trn′* ⟹ *g1 trn* = *g1 trn′* ⟹ *φ2 trn* ⟹ *φ2 trn′* ⟹ *f2 trn* = *f2 trn′*
and *φ1-γ2*: ⋀*tr trn. One.validFrom istate* (*tr* ## *trn*) ⟹ *φ1 trn* ⟹ *γ2 trn*
and *γ2-φ1*: ⋀*tr tr′ trn trn′. Two.O tr* = *Two.O tr′* ⟹ *One.validFrom istate* (*tr*

trn) $\Longrightarrow$ *One.validFrom istate* (*tr′ ## trn′*) $\Longrightarrow$ γ*2 trn* $\Longrightarrow$ γ*2 trn′* $\Longrightarrow$ *g2 trn* = *g2 trn′* $\Longrightarrow$ φ*1 trn* = φ*1 trn′*
and *g2-f1*: $\bigwedge$*tr tr′ trn trn′*. *Two.O tr* = *Two.O tr′* $\Longrightarrow$ *One.validFrom istate* (*tr ## trn*) $\Longrightarrow$ *One.validFrom istate* (*tr′ ## trn′*) $\Longrightarrow$ γ*2 trn* $\Longrightarrow$ γ*2 trn′* $\Longrightarrow$ *g2 trn* = *g2 trn′* $\Longrightarrow$ φ*1 trn* $\Longrightarrow$ φ*1 trn′* $\Longrightarrow$ *f1 trn* = *f1 trn′*
and *T2-*γ*1*: $\bigwedge$*tr trn*. *One.validFrom istate* (*tr ## trn*) $\Longrightarrow$ *never T2 tr* $\Longrightarrow$ *T2 trn* $\Longrightarrow$ γ*1 trn*
and γ*1-T2*: $\bigwedge$*tr tr′ trn trn′*. *One.O tr* = *One.O tr′* $\Longrightarrow$ *One.validFrom istate* (*tr ## trn*) $\Longrightarrow$ *One.validFrom istate* (*tr′ ## trn′*) $\Longrightarrow$ γ*1 trn* $\Longrightarrow$ γ*1 trn′* $\Longrightarrow$ *g1 trn* = *g1 trn′* $\Longrightarrow$ *T2 trn* = *T2 trn′*
begin

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

lemma *O-Nil-never*: *O tr* = [] $\longleftrightarrow$ *never* γ *tr* ⟨*proof*⟩
lemma *Nil-O-never*: [] = *O tr* $\longleftrightarrow$ *never* γ *tr* ⟨*proof*⟩
lemma *O-append*: *O* (*tr* @ *tr′*) = *O tr* @ *O tr′* ⟨*proof*⟩

lemma *never-*γ*12-never-*γ: *One.validFrom istate* (*tr* @ *tr′*) $\Longrightarrow$ *never* γ*1 tr′* $\vee$ *never* γ*2 tr′* $\Longrightarrow$ *never* γ *tr′*
⟨*proof*⟩

lemma *never-*γ*1-never-*φ*2*: *One.validFrom istate* (*tr* @ *tr′*) $\Longrightarrow$ *never* γ*1 tr′* $\Longrightarrow$ *never* φ*2 tr′*
⟨*proof*⟩

lemma *never-*γ*2-never-*φ*1*: *One.validFrom istate* (*tr* @ *tr′*) $\Longrightarrow$ *never* γ*2 tr′* $\Longrightarrow$ *never* φ*1 tr′*
⟨*proof*⟩

lemma *never-*γ*1-never-T2*: *One.validFrom istate* (*tr* @ *tr′*) $\Longrightarrow$ *never T2 tr* $\Longrightarrow$ *never* γ*1 tr′* $\Longrightarrow$ *never T2 tr′*
⟨*proof*⟩

sublocale *Abstract-BD-Security-Two-Secrets One.validFrom istate One.V One.O B1 never T1 Two.V Two.O B2 never T2 O*
⟨*proof*⟩

end

end

References

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

ings, volume 9807 of *Lecture Notes in Computer Science*, pages 87–106. Springer, 2016.

[2] T. Bauereiß, A. Pesenti Gritti, A. Popescu, and F. Raimondi. Cosmedis: A distributed social media platform with formally verified confidentiality guarantees. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, pages 729–748. IEEE Computer Society, 2017.

[3] T. Bauereiß, A. Pesenti Gritti, A. Popescu, and F. Raimondi. Cosmed: A confidentiality-verified social media platform. *J. Autom. Reason.*, 61(1-4):113–139, 2018.

[4] T. Bauereiss and A. Popescu. CoSMed: A confidentiality-verified social media platform. In M. Eberl, G. Klein, A. Lochbihler, T. Nipkow, L. Paulson, and R. Thiemann, editors, *Archive of Formal Proofs*, 2021.

[5] T. Bauereiss and A. Popescu. CoSMeDis: A confidentiality-verified distributed social media platform. In M. Eberl, G. Klein, A. Lochbihler, T. Nipkow, L. Paulson, and R. Thiemann, editors, *Archive of Formal Proofs*, 2021.

[6] S. Kanav, P. Lammich, and A. Popescu. A conference management system with verified document confidentiality. In A. Biere and R. Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 167–183. Springer, 2014.

[7] A. Popescu, T. Bauereiss, and P. Lammich. Bounded-Deducibility security (invited paper). In L. Cohen and C. Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 3:1–3:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[8] A. Popescu, P. Lammich, and T. Bauereiss. Bounded-deducibility security. In G. Klein, T. Nipkow, and L. Paulson, editors, *Archive of Formal Proofs*, 2014.

[9] A. Popescu, P. Lammich, and T. Bauereiss. CoCon: A confidentiality-verified conference management system. In M. Eberl, G. Klein, A. Lochbihler, T. Nipkow, L. Paulson, and R. Thiemann, editors, *Archive of Formal Proofs*, 2021.

[10] A. Popescu, P. Lammich, and P. Hou. Cocon: A conference management system with formally verified document confidentiality. *J. Autom. Reason.*, 65(2):321–356, 2021.