Network Security Policy Verification

Cornelius Diekmann

March 17, 2025

Abstract. We present a unified theory for verifying network security policies. A security policy is represented as directed graph. To check high-level security goals, security invariants over the policy are expressed. We cover monotonic security invariants, i.e. prohibiting more does not harm security. We provide the following contributions for the security invariant theory. (*i*) Secure auto-completion of scenario-specific knowledge, which eases usability. (*ii*) Security violations can be repaired by tightening the policy iff the security invariants hold for the deny-all policy. (*iii*) An algorithm to compute a security policy. (*iv*) A formalization of stateful connection semantics in network security mechanisms. (*v*) An algorithm to compute a secure stateful implementation of a policy. (*vi*) An executable implementation of all the theory. (*vii*) Examples, ranging from an aircraft cabin data network to the analysis of a large real-world firewall.

For a detailed description, see [2, 3, 1].

Acknowledgements. This entry contains contributions by Lars Hupel and would not have made it into the AFP without him. I want to thank the Isabelle group Munich for always providing valuable help. I would like to express my deep gratitude to my supervisor, Georg Carle, for supporting this topic and facilitating further research possibilities in this field.

Contents

theory *TopoS-Vertices*
imports *Main*
HOL−Library.Char-ord
HOL−Library.List-Lexorder
begin

1 A type for vertices

This theory makes extensive use of graphs. We define a typeclass *vertex* for the vertices we will use in our theory. The vertices will correspond to network or policy entities.

Later, we will conduct some proves by providing counterexamples. Therefore, we say that the type of a vertex has at least three pairwise distinct members.

For example, the types *string*, *nat*, *bool* × *bool* and many other fulfill this assumption. The type *bool* alone does not fulfill this assumption, because it only has two elements.

This is only a constraint over the type, of course, a policy with less than three entities can also be verified.

TL;DR: We define *'a vertex*, which is as good as *'a*.

```
class vertex =
  fixes vertex-1 :: 'a
  fixes vertex-2 :: 'a
  fixes vertex-3 :: 'a
  assumes distinct-vertices: distinct [vertex-1, vertex-2, vertex-3]
begin
  lemma distinct-vertices12[simp]: vertex-1 ≠ vertex-2 ⟨proof⟩
  lemma distinct-vertices13[simp]: vertex-1 ≠ vertex-3 ⟨proof⟩
  lemma distinct-vertices23[simp]: vertex-2 ≠ vertex-3 ⟨proof⟩

  lemmas distinct-vertices-sym = distinct-vertices12[symmetric] distinct-vertices13[symmetric]
        distinct-vertices23[symmetric]
  declare distinct-vertices-sym[simp]
end
```

Numbers, chars and strings are good candidates for vertices.

```
instantiation nat::vertex
begin
  definition vertex-1-nat ::nat where vertex-1 ≡ (1::nat)
  definition vertex-2-nat ::nat where vertex-2 ≡ (2::nat)
  definition vertex-3-nat ::nat where vertex-3 ≡ (3::nat)
instance ⟨proof⟩
end
value vertex-1::nat

instantiation int::vertex
begin
  definition vertex-1-int ::int where vertex-1 ≡ (1::int)
  definition vertex-2-int ::int where vertex-2 ≡ (2::int)
  definition vertex-3-int ::int where vertex-3 ≡ (3::int)
instance ⟨proof⟩
end

instantiation char::vertex
```

begin
  definition *vertex-1-char* ::*char* **where** *vertex-1* ≡ *CHR ''A''*
  definition *vertex-2-char* ::*char* **where** *vertex-2* ≡ *CHR ''B''*
  definition *vertex-3-char* ::*char* **where** *vertex-3* ≡ *CHR ''C''*
instance ⟨*proof*⟩
end
value *vertex-1*::*char*

instantiation *list* :: (*vertex*) *vertex*
begin
  definition *vertex-1-list* **where** *vertex-1* ≡ []
  definition *vertex-2-list* **where** *vertex-2* ≡ [*vertex-1*]
  definition *vertex-3-list* **where** *vertex-3* ≡ [*vertex-1*, *vertex-1*]
instance ⟨*proof*⟩
end

— for the ML graphviz visualizer
⟨*ML*⟩

end
theory *TopoS-Interface*
imports *Main Lib/FiniteGraph TopoS-Vertices Lib/TopoS-Util*
begin

2 Security Invariants

A good documentation of this formalization is available in [3].

We define security invariants over a graph. The graph corresponds to the network's access control structure.

  record (*'v*::*vertex*, *'a*) *TopoS-Params* =
    node-properties :: *'v*::*vertex* ⇒ *'a option*

A Security Invariant is defined as locale.

We successively define more and more locales with more and more assumptions. This clearly depicts which assumptions are necessary to use certain features of a Security Invariant. In addition, it makes instance proofs of Security Invariants easier, since the lemmas obtained by an (easy, few assumptions) instance proof can be used for the complicated (more assumptions) instance proofs.

A security Invariant consists of one function: *sinvar*. Essentially, it is a predicate over the policy (depicted as graph G and a host attribute mapping (nP)).

A Security Invariant where the offending flows (flows that invalidate the policy) can be defined and calculated. No assumptions are necessary for this step.

  locale *SecurityInvariant-withOffendingFlows* =
    fixes *sinvar*::(*'v*::*vertex*) *graph* ⇒ (*'v*::*vertex* ⇒ *'a*) ⇒ *bool* — policy ⇒ host attribute mapping ⇒ bool
    begin

— Offending Flows definitions:
definition *is-offending-flows*::$('v \times 'v)$ *set* $\Rightarrow$ $'v$ *graph* $\Rightarrow$ $('v \Rightarrow 'a) \Rightarrow$ *bool* **where**
is-offending-flows f G nP $\equiv \neg$ *sinvar G nP* $\wedge$ *sinvar* (*delete-edges G f*) *nP*

— Above definition is not minimal:
definition *is-offending-flows-min-set*::$('v \times 'v)$ *set* $\Rightarrow$ $'v$ *graph* $\Rightarrow$ $('v \Rightarrow 'a) \Rightarrow$ *bool* **where**
is-offending-flows-min-set f G nP $\equiv$ *is-offending-flows f G nP* $\wedge$
$(\forall\ (e1,\ e2) \in f.\ \neg$ *sinvar* (*add-edge e1 e2* (*delete-edges G f*)) *nP*)

— The set of all offending flows.
definition *set-offending-flows*::$'v$ *graph* $\Rightarrow$ $('v \Rightarrow 'a) \Rightarrow ('v \times 'v)$ *set set* **where**
set-offending-flows G nP = $\{F.\ F \subseteq$ (*edges G*) $\wedge$ *is-offending-flows-min-set F G nP*$\}$

Some of the *set-offending-flows* definition

lemma *offending-not-empty*: $[\![$ $F \in$ *set-offending-flows G nP* $]\!] \Longrightarrow F \neq \{\}$
⟨*proof*⟩
lemma *empty-offending-contra*:
$[\![$ $F \in$ *set-offending-flows G nP*; $F = \{\}]\!] \Longrightarrow$ *False*
⟨*proof*⟩
lemma *offending-notevalD*: $F \in$ *set-offending-flows G nP* $\Longrightarrow \neg$ *sinvar G nP*
⟨*proof*⟩
lemma *sinvar-no-offending*: *sinvar G nP* $\Longrightarrow$ *set-offending-flows G nP* $= \{\}$
⟨*proof*⟩
theorem *removing-offending-flows-makes-invariant-hold*:
$\forall F \in$ *set-offending-flows G nP*. *sinvar* (*delete-edges G F*) *nP*
⟨*proof*⟩
corollary *valid-without-offending-flows*:
$[\![$ $F \in$ *set-offending-flows G nP* $]\!] \Longrightarrow$ *sinvar* (*delete-edges G F*) *nP*
⟨*proof*⟩

lemma *set-offending-flows-simp*:
$[\![$ *wf-graph G* $]\!] \Longrightarrow$
set-offending-flows G nP $= \{F.\ F \subseteq$ *edges G* $\wedge$
$(\neg$ *sinvar G nP* $\wedge$ *sinvar* (|*nodes* = *nodes G*, *edges* = *edges G* − *F*|) *nP*) $\wedge$
$(\forall (e1,\ e2) \in F.\ \neg$ *sinvar* (|*nodes* = *nodes G*, *edges* = $\{(e1,\ e2)\} \cup$ (*edges G* − *F*)|) *nP*)$\}$
⟨*proof*⟩

end

print-locale! *SecurityInvariant-withOffendingFlows*

The locale *SecurityInvariant-withOffendingFlows* has no assumptions about the security invariant *sinvar*. Undesirable things may happen: The offending flows can be empty, even for a violated invariant.

We provide an example, the security invariant λ- -. *False*. As host attributes, we simply use the identity function *id*.

lemma *SecurityInvariant-withOffendingFlows.set-offending-flows* (λ- -. *False*) (| *nodes* = {''*v1*''}, *edges*={} |) *id* = {}
lemma *SecurityInvariant-withOffendingFlows.set-offending-flows* (λ- -. *False*)
(| *nodes* = {''*v1*'', ''*v2*''}, *edges* = {(''*v1*'', ''*v2*'')} |) *id* = {}

In general, there exists a *sinvar* such that the invariant does not hold and no offending flows exits.

lemma $\exists$ *sinvar*. $\neg$ *sinvar G nP* $\wedge$ *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G nP* = {}
⟨*proof*⟩

Thus, we introduce usefulness properties that prohibits such useless invariants.

We summarize them in an invariant. It requires the following:

1. The offending flows are always defined.
2. The invariant is monotonic, i.e. prohibiting more is more secure.
3. And, the (non-minimal) offending flows are monotonic, i.e. prohibiting more solves more security issues.

Later, we will show that is suffices to show that the invariant is monotonic. The other two properties can be derived.

locale *SecurityInvariant-preliminaries* = *SecurityInvariant-withOffendingFlows sinvar*
for *sinvar*
+
assumes
defined-offending:
⟦ *wf-graph G*; $\neg$ *sinvar G nP* ⟧ $\Longrightarrow$ *set-offending-flows G nP* $\neq$ {}
and
mono-sinvar:
⟦ *wf-graph* ⦇ *nodes* = *N*, *edges* = *E* ⦈; $E' \subseteq E$; *sinvar* ⦇ *nodes* = *N*, *edges* = *E* ⦈ *nP* ⟧ $\Longrightarrow$
sinvar ⦇ *nodes* = *N*, *edges* = *E'* ⦈ *nP*
and *mono-offending*:
⟦ *wf-graph G*; *is-offending-flows ff G nP* ⟧ $\Longrightarrow$ *is-offending-flows* (*ff* $\cup$ *f'*) *G nP*
begin

To instantiate a *SecurityInvariant-preliminaries*, here are some hints: Have a look at the *TopoS-withOffendingFlows.thy* file. There is a definition of *sinvar-mono*. It impplies *mono-sinvar* and *mono-offending apply*(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-sinvar-mono*[*OF sinvar-mono*]) *apply*(*fact SecurityInvariant-withOffendingFlows.sinvar-mono-imp-is-offending-flows-mono*[*OF sinvar-mono*])

In addition, *SecurityInvariant-withOffendingFlows.mono-imp-set-offending-flows-not-empty*[*OF sinvar-mono*] gives a nice proof rule for *defined-offending*

Basically, *sinvar-mono*. implies almost all assumptions here and is equal to *mono-sinvar*.

end

2.1 Security Invariants with secure auto-completion of host attribute mappings

We will now add a new artifact to the Security Invariant. It is a secure default host attribute, we will use the symbol $\bot$.

The newly introduced Boolean *receiver-violation* tells whether a security violation happens at the sender's or the receiver's side.

The details can be looked up in [3].

locale *SecurityInvariant* = *SecurityInvariant-preliminaries sinvar*

for *sinvar*::('v::vertex) graph ⇒ ('v::vertex ⇒ 'a) ⇒ bool*
+
fixes *default-node-properties* :: *'a* (‹⊥›)
and *receiver-violation* :: *bool*
assumes
— default value can never fix a security violation.
— Idea: Assume there is a violation, then there is some offending flow. *receiver-violation* defines whether the violation happens at the sender's or the receiver's side. We call the place of the violation the *offending host*. We replace the host attribute of the offending host with the default attribute. Giving an offending host, a *secure* default attribute does not change whether the invariant holds. I.e. this reconfiguration does not remove information, thus preserves all security critical information. Thought experiment preliminaries: Can a default configuration ever solve an existing security violation? NO! Thought experiment 1: admin forgot to configure host, hence it is handled by default configuration value ... Thought experiment 2: new node (attacker) is added to the network. What is its default configuration value ...

default-secure:
$\llbracket$ *wf-graph G*; ¬ *sinvar G nP*; *F* ∈ *set-offending-flows G nP* $\rrbracket \Longrightarrow$
(¬ *receiver-violation* ⟶ *i* ∈ *fst* ‘ *F* ⟶ ¬ *sinvar G* (*nP*(*i* := ⊥))) ∧
(*receiver-violation* ⟶ *i* ∈ *snd* ‘ *F* ⟶ ¬ *sinvar G* (*nP*(*i* := ⊥)))
and
default-unique:
otherbot ≠ ⊥ ⟹
∃ (*G*::('v::vertex) graph) *nP i F*. *wf-graph G* ∧ ¬ *sinvar G nP* ∧ *F* ∈ *set-offending-flows G nP* ∧
sinvar (*delete-edges G F*) *nP* ∧
(¬ *receiver-violation* ⟶ *i* ∈ *fst* ‘ *F* ∧ *sinvar G* (*nP*(*i* := *otherbot*))) ∧
(*receiver-violation* ⟶ *i* ∈ *snd* ‘ *F* ∧ *sinvar G* (*nP*(*i* := *otherbot*)))
begin
— Removes option type, replaces with default host attribute
fun *node-props* :: ('v, 'a) *TopoS-Params* ⇒ ('v ⇒ 'a) **where**
node-props P = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ ⊥))

definition *node-props-formaldef* :: ('v, 'a) *TopoS-Params* ⇒ ('v ⇒ 'a) **where**
node-props-formaldef P ≡
(λ *i*. (*if i* ∈ *dom* (*node-properties P*) *then the* (*node-properties P i*) *else* ⊥))

lemma *node-props-eq-node-props-formaldef*: *node-props-formaldef* = *node-props*
⟨*proof*⟩

Checking whether a security invariant holds.

1. check that the policy *G* is syntactically valid
2. check the security invariant *sinvar*

definition *eval*::'v graph ⇒ ('v, 'a) *TopoS-Params* ⇒ *bool* **where**
eval G P ≡ *wf-graph G* ∧ *sinvar G* (*node-props P*)

lemma *unique-common-math-notation*:
assumes ∀ *G nP i F*. *wf-graph* (*G*::('v::vertex) graph) ∧ ¬ *sinvar G nP* ∧ *F* ∈ *set-offending-flows G nP* ∧
sinvar (*delete-edges G F*) *nP* ∧
(¬ *receiver-violation* ⟶ *i* ∈ *fst* ‘ *F* ⟶ ¬ *sinvar G* (*nP*(*i* := *otherbot*))) ∧

$(\textit{receiver-violation} \longrightarrow i \in \textit{snd} \text{ ` } F \longrightarrow \neg \textit{ sinvar } G \ (nP(i := \textit{otherbot})))$
shows *otherbot* $= \bot$
$\langle proof \rangle$
end

print-locale! *SecurityInvariant*

2.2 Information Flow Security and Access Control

receiver-violation defines the offending host. Thus, it defines when the violation happens. We found that this coincides with the invariant's security strategy.

ACS If the violation happens at the sender, we have an access control strategy (*ACS*). I.e. the sender does not have the appropriate rights to initiate the connection.

IFS If the violation happens at the receiver, we have an information flow security strategy (*IFS*) I.e. the receiver lacks the appropriate security level to retrieve the (confidential) information. The violations happens only when the receiver reads the data.

We refine our *SecurityInvariant* locale.

2.3 Information Flow Security Strategy (IFS)

locale *SecurityInvariant-IFS* = *SecurityInvariant-preliminaries sinvar*
for *sinvar*::('*v*::*vertex*) *graph* $\Rightarrow$ ('*v*::*vertex* $\Rightarrow$ '*a*) $\Rightarrow$ *bool*
+
fixes *default-node-properties* :: '*a* (‹$\bot$›)
assumes *default-secure-IFS*:
$[\![$ *wf-graph G*; $f \in$ *set-offending-flows G nP* $]\!] \Longrightarrow$
$\forall i \in$ *snd*‘ *f*. $\neg$ *sinvar G* $(nP(i := \bot))$
and
— If some otherbot fulfills *default-secure*, it must be $\bot$ Hence, $\bot$ is uniquely defined
default-unique-IFS:
$(\forall$ *G f nP i*. *wf-graph G* $\wedge$ $f \in$ *set-offending-flows G nP* $\wedge$ $i \in$ *snd*‘ *f*
$\longrightarrow \neg$ *sinvar G* $(nP(i := \textit{otherbot}))) \Longrightarrow$ *otherbot* $= \bot$
begin
lemma *default-unique-EX-notation*: *otherbot* $\neq \bot \Longrightarrow$
$\exists$ *G nP i f*. *wf-graph G* $\wedge \neg$ *sinvar G nP* $\wedge$ $f \in$ *set-offending-flows G nP* $\wedge$
sinvar (*delete-edges G f*) *nP* $\wedge$
$(i \in$ *snd*‘ *f* $\wedge$ *sinvar G* $(nP(i := \textit{otherbot})))$
$\langle proof \rangle$
end

sublocale *SecurityInvariant-IFS* $\subseteq$ *SecurityInvariant* **where** *receiver-violation=True*
$\langle proof \rangle$

locale *SecurityInvariant-IFS-otherDirectrion* = *SecurityInvariant* **where** *receiver-violation=True*
sublocale *SecurityInvariant-IFS-otherDirectrion* $\subseteq$ *SecurityInvariant-IFS*
$\langle proof \rangle$

lemma *default-uniqueness-by-counterexample-IFS*:

assumes $(\forall\, G\ F\ nP\ i.\ \textit{wf-graph}\ G \wedge F \in \textit{SecurityInvariant-withOffendingFlows.set-offending-flows}$ $\textit{sinvar}\ G\ nP \wedge i \in \textit{snd}\text{`}\ F$
$\longrightarrow \neg\ \textit{sinvar}\ G\ (nP(i := \textit{otherbot})))$
and $\textit{otherbot} \neq \textit{default-value} \Longrightarrow$
$\exists\, G\ nP\ i\ F.\ \textit{wf-graph}\ G \wedge \neg\ \textit{sinvar}\ G\ nP \wedge F \in (\textit{SecurityInvariant-withOffendingFlows.set-offending-flows}$ $\textit{sinvar}\ G\ nP) \wedge$
$\textit{sinvar}\ (\textit{delete-edges}\ G\ F)\ nP \wedge$
$i \in \textit{snd}\ \text{`}\ F \wedge \textit{sinvar}\ G\ (nP(i := \textit{otherbot}))$
shows $\textit{otherbot} = \textit{default-value}$
⟨*proof*⟩

2.4 Access Control Strategy (ACS)

locale *SecurityInvariant-ACS* = *SecurityInvariant-preliminaries sinvar*
for $\textit{sinvar}{::}('v{::}\textit{vertex})\ \textit{graph} \Rightarrow ('v{::}\textit{vertex} \Rightarrow 'a) \Rightarrow \textit{bool}$
+
fixes $\textit{default-node-properties} :: 'a\ (\langle\bot\rangle)$
assumes *default-secure-ACS*:
$[\![\ \textit{wf-graph}\ G;\ f \in \textit{set-offending-flows}\ G\ nP\ ]\!] \Longrightarrow$
$\forall\, i \in \textit{fst}\text{`}\ f.\ \neg\ \textit{sinvar}\ G\ (nP(i := \bot))$
and
default-unique-ACS:
$(\forall\, G\ f\ nP\ i.\ \textit{wf-graph}\ G \wedge f \in \textit{set-offending-flows}\ G\ nP \wedge i \in \textit{fst}\text{`}\ f$
$\longrightarrow \neg\ \textit{sinvar}\ G\ (nP(i := \textit{otherbot}))) \Longrightarrow \textit{otherbot} = \bot$
begin
lemma *default-unique-EX-notation*: $\textit{otherbot} \neq \bot \Longrightarrow$
$\exists\ G\ nP\ i\ f.\ \textit{wf-graph}\ G \wedge \neg\ \textit{sinvar}\ G\ nP \wedge f \in \textit{set-offending-flows}\ G\ nP \wedge$
$\textit{sinvar}\ (\textit{delete-edges}\ G\ f)\ nP \wedge$
$(i \in \textit{fst}\text{`}\ f \wedge \textit{sinvar}\ G\ (nP(i := \textit{otherbot})))$
⟨*proof*⟩
end

sublocale *SecurityInvariant-ACS* ⊆ *SecurityInvariant* **where** *receiver-violation=False*
⟨*proof*⟩

locale *SecurityInvariant-ACS-otherDirectrion* = *SecurityInvariant* **where** *receiver-violation=False*
sublocale *SecurityInvariant-ACS-otherDirectrion* ⊆ *SecurityInvariant-ACS*
⟨*proof*⟩

lemma *default-uniqueness-by-counterexample-ACS*:
assumes $(\forall\, G\ F\ nP\ i.\ \textit{wf-graph}\ G \wedge F \in \textit{SecurityInvariant-withOffendingFlows.set-offending-flows}$ $\textit{sinvar}\ G\ nP \wedge i \in \textit{fst}\ \text{`}\ F$
$\longrightarrow \neg\ \textit{sinvar}\ G\ (nP(i := \textit{otherbot})))$
and $\textit{otherbot} \neq \textit{default-value} \Longrightarrow$
$\exists\, G\ nP\ i\ F.\ \textit{wf-graph}\ G \wedge \neg\ \textit{sinvar}\ G\ nP \wedge F \in (\textit{SecurityInvariant-withOffendingFlows.set-offending-flows}$ $\textit{sinvar}\ G\ nP) \wedge$
$\textit{sinvar}\ (\textit{delete-edges}\ G\ F)\ nP \wedge$
$i \in \textit{fst}\ \text{`}\ F \wedge \textit{sinvar}\ G\ (nP(i := \textit{otherbot}))$
shows $\textit{otherbot} = \textit{default-value}$
⟨*proof*⟩

The sublocale relationships tell that the simplified *SecurityInvariant-ACS* and *SecurityInvari-*

ant-IFS assumptions suffice to do tho generic SecurityInvariant assumptions.

end
theory *TopoS-withOffendingFlows*
imports *TopoS-Interface*
begin

3 *SecurityInvariant* Instantiation Helpers

The security invariant locales are set up hierarchically to ease instantiation proofs. The first locale, *SecurityInvariant-withOffendingFlows* has no assumptions, thus instantiations is for free. The first step focuses on monotonicity,

context *SecurityInvariant-withOffendingFlows*
begin

We define the monotonicity of *sinvar*:

$\bigwedge nP\ N\ E'\ E.\ [\![wf\text{-}graph\ (\!|nodes = N,\ edges = E|\!);\ E' \subseteq E;\ sinvar\ (\!|nodes = N,\ edges = E|\!)\ nP]\!] \Longrightarrow sinvar\ (\!|nodes = N,\ edges = E'|\!)\ nP$

Having a valid invariant, removing edges retains the validity. I.e. prohibiting more, is more or equally secure.

definition *sinvar-mono* :: *bool* **where**
  sinvar-mono $\longleftrightarrow$ ($\forall$ *nP N E' E*.
    wf-graph (| *nodes* = *N*, *edges* = *E* |) $\wedge$
    $E' \subseteq E$ $\wedge$
    sinvar (| *nodes* = *N*, *edges* = *E* |) *nP* $\longrightarrow$ *sinvar* (| *nodes* = *N*, *edges* = *E'* |) *nP*)

If one can show *sinvar-mono*, then the instantiation of the *SecurityInvariant-preliminaries* locale is tremendously simplified.

lemma *sinvar-mono-I-proofrule-simple*:
[[($\forall$ *G nP*. *sinvar G nP* = ($\forall$ (*e1*, *e2*) $\in$ *edges G*. *P e1 e2 nP*))]] $\Longrightarrow$ *sinvar-mono*
⟨*proof*⟩

lemma *sinvar-mono-I-proofrule*:
[[($\forall$ *nP* (*G*:: *'v graph*). *sinvar G nP* = ($\forall$ (*e1*, *e2*) $\in$ *edges G*. *P e1 e2 nP G*));
  ($\forall$ *nP e1 e2 N E' E*.
    wf-graph (| *nodes* = *N*, *edges* = *E* |) $\wedge$
    (*e1*,*e2*) $\in$ *E* $\wedge$
    $E' \subseteq E$ $\wedge$
   P e1 e2 nP (|*nodes* = *N*, *edges* = *E*|) $\longrightarrow$ *P e1 e2 nP* (|*nodes* = *N*, *edges* = *E'*|))]] $\Longrightarrow$ *sinvar-mono*
⟨*proof*⟩

Invariant violations do not disappear if we add more flows.

lemma *sinvar-mono-imp-negative-mono*:
sinvar-mono $\Longrightarrow$ *wf-graph* (| *nodes* = *N*, *edges* = *E* |) $\Longrightarrow$ $E' \subseteq E$ $\Longrightarrow$
$\neg$ *sinvar* (| *nodes* = *N*, *edges* = *E'* |) *nP* $\Longrightarrow$ $\neg$ *sinvar* (| *nodes* = *N*, *edges* = *E* |) *nP*
⟨*proof*⟩

corollary *sinvar-mono-imp-negative-delete-edge-mono*:
  sinvar-mono $\Longrightarrow$ *wf-graph G* $\Longrightarrow$ $X \subseteq Y$ $\Longrightarrow$ $\neg$ *sinvar* (*delete-edges G Y*) *nP* $\Longrightarrow$ $\neg$ *sinvar* (*delete-edges G X*) *nP*
⟨*proof*⟩

lemma *sinvar-mono-imp-is-offending-flows-mono*:
assumes *mono*: *sinvar-mono*
and *wfG*: *wf-graph G*
shows *is-offending-flows FF G nP* $\Longrightarrow$ *is-offending-flows* (*FF* $\cup$ *F*) *G nP*
⟨*proof*⟩

lemma *sinvar-mono-imp-sinvar-mono*:
sinvar-mono $\Longrightarrow$ *wf-graph* (| *nodes* = *N*, *edges* = *E* |) $\Longrightarrow$ $E' \subseteq E$ $\Longrightarrow$ *sinvar* (| *nodes* = *N*, *edges* = *E* |) *nP* $\Longrightarrow$
sinvar (| *nodes* = *N*, *edges* = *E'* |) *nP*
⟨*proof*⟩

end

3.1 Offending Flows Not Empty Helper Lemmata

context *SecurityInvariant-withOffendingFlows*
begin

Give an over-approximation of offending flows (e.g. all edges) and get back a minimal set

fun *minimalize-offending-overapprox* :: ('v × 'v) *list* ⇒ ('v × 'v) *list* ⇒
'v *graph* ⇒ ('v ⇒ 'a) ⇒ ('v × 'v) *list* **where**
minimalize-offending-overapprox [] *keep* - - = *keep* |
minimalize-offending-overapprox (*f*#*fs*) *keep G nP* = (*if sinvar* (*delete-edges-list G* (*fs*@*keep*)) *nP* *then*
minimalize-offending-overapprox fs keep G nP
else
minimalize-offending-overapprox fs (*f*#*keep*) *G nP*
)

The graph we check in *minimalize-offending-overapprox*, *G* (−) (*fs* $\cup$ *keep*) is the graph from the *offending-flows-min-set* condition. We add *f* and remove it.

lemma *minimalize-offending-overapprox-subset*:
set (*minimalize-offending-overapprox ff keeps G nP*) $\subseteq$ *set ff* $\cup$ *set keeps*
⟨*proof*⟩

lemma *not-model-mono-imp-addedge-mono*:
assumes *mono*: *sinvar-mono*
and *vG*: *wf-graph G* **and** *ain*: (*a1*,*a2*) $\in$ *edges G* **and** *xy*: $X \subseteq Y$ **and** *ns*: ¬ *sinvar* (*add-edge a1 a2* (*delete-edges G* (*Y*))) *nP*
shows ¬ *sinvar* (*add-edge a1 a2* (*delete-edges G X*)) *nP*
⟨*proof*⟩

theorem *is-offending-flows-min-set-minimalize-offending-overapprox*:

assumes *mono*: *sinvar-mono*
and *vG*: *wf-graph G* **and** *iO*: *is-offending-flows* (*set ff*) *G nP* **and** *sF*: *set ff* ⊆ *edges G* **and** *dF*: *distinct ff*
shows *is-offending-flows-min-set* (*set* (*minimalize-offending-overapprox ff* [] *G nP*)) *G nP*
(**is** *is-offending-flows-min-set ?minset G nP*)
⟨*proof*⟩

corollary *mono-imp-set-offending-flows-not-empty*:
assumes *mono-sinvar*: *sinvar-mono*
and *vG*: *wf-graph G* **and** *iO*: *is-offending-flows* (*set ff*) *G nP* **and** *sS*: *set ff* ⊆ *edges G* **and** *dF*: *distinct ff*
shows
set-offending-flows G nP ≠ {}
⟨*proof*⟩

To show that *set-offending-flows* is not empty, the previous corollary ⟦*sinvar-mono*; *wf-graph ?G*; *is-offending-flows* (*set ?ff*) *?G ?nP*; *set ?ff* ⊆ *edges ?G*; *distinct ?ff*⟧ ⟹ *set-offending-flows ?G ?nP* ≠ {} is very useful. Just select *set ff* = *edges G*.

If there exists a security violations, there a means to fix it if and only if the network in which nobody communicates with anyone fulfills the security requirement

theorem *valid-empty-edges-iff-exists-offending-flows*:
assumes *mono*: *sinvar-mono* **and** *wfG*: *wf-graph G* **and** *noteval*: ¬ *sinvar G nP*
shows *sinvar* (| *nodes* = *nodes G*, *edges* = {} |) *nP* ⟷ *set-offending-flows G nP* ≠ {}
⟨*proof*⟩

minimalize-offending-overapprox not only computes a set where *is-offending-flows-min-set* holds, but it also returns a subset of the input.

lemma *minimalize-offending-overapprox-keeps-keeps*: (*set keeps*) ⊆ *set* (*minimalize-offending-overapprox ff keeps G nP*)
⟨*proof*⟩

lemma *minimalize-offending-overapprox-subseteq-input*: *set* (*minimalize-offending-overapprox ff keeps G nP*) ⊆ (*set ff*) ∪ (*set keeps*)
⟨*proof*⟩

end

context *SecurityInvariant-preliminaries*
begin

sinvar-mono naturally holds in *SecurityInvariant-preliminaries*

lemma *sinvar-monoI*: *sinvar-mono*
⟨*proof*⟩

Note: due to monotonicity, the minimality also holds for arbitrary subsets

lemma assumes *wf-graph G* **and** *is-offending-flows-min-set F G nP* **and** *F* ⊆ *edges G* **and** *E* ⊆ *F* **and** *E* ≠ {}
shows ¬ *sinvar* (| *nodes* = *nodes G*, *edges* = ((*edges G*) − *F*) ∪ *E* |) *nP*
⟨*proof*⟩

The algorithm *minimalize-offending-overapprox* is correct

lemma *minimalize-offending-overapprox-sound*:
$\llbracket$ *wf-graph G*; *is-offending-flows* (*set ff*) *G nP*; *set ff* $\subseteq$ *edges G*; *distinct ff* $\rrbracket$
$\Longrightarrow$ *is-offending-flows-min-set* (*set* (*minimalize-offending-overapprox ff* [] *G nP*)) *G nP*
⟨*proof*⟩

If ¬ *sinvar G nP* Given a list ff, (ff is distinct and a subset of G's edges) such that *sinvar* (*V*, *E* − *ff*) *nP* *minimalize-offending-overapprox* minimizes ff such that we get an offending flows Note: choosing ff = edges G is a good choice!

theorem *minimalize-offending-overapprox-gives-back-an-offending-flow*:
$\llbracket$ *wf-graph G*; *is-offending-flows* (*set ff*) *G nP*; *set ff* $\subseteq$ *edges G*; *distinct ff* $\rrbracket$
$\Longrightarrow$
(*set* (*minimalize-offending-overapprox ff* [] *G nP*)) $\in$ *set-offending-flows G nP*
⟨*proof*⟩

end

A version which acts on configured security invariants. I.e. there is no type *'a* for the host attributes in it.

fun *minimalize-offending-overapprox* :: (*'v graph* ⇒ *bool*) ⇒ (*'v* × *'v*) *list* ⇒ (*'v* × *'v*) *list* ⇒ *'v graph* ⇒(*'v* × *'v*) *list* **where**
minimalize-offending-overapprox - [] *keep* - = *keep* |
minimalize-offending-overapprox m (*f*#*fs*) *keep G* = (*if m* (*delete-edges-list G* (*fs*@*keep*)) *then*
minimalize-offending-overapprox m fs keep G
else
minimalize-offending-overapprox m fs (*f*#*keep*) *G*
)

lemma *minimalize-offending-overapprox-boundnP*:
shows *minimalize-offending-overapprox* (λ*G*. *m G nP*) *fs keeps G* =
SecurityInvariant-withOffendingFlows.minimalize-offending-overapprox m fs keeps G nP
⟨*proof*⟩

context *SecurityInvariant-withOffendingFlows*
begin

If there is a violation and there are no offending flows, there does not exist a possibility to fix the violation by tightening the policy. $\llbracket$*sinvar-mono*; *wf-graph ?G*; ¬ *sinvar ?G ?nP*$\rrbracket$ $\Longrightarrow$ *sinvar* (\|*nodes* = *nodes ?G*, *edges* = {}\|) *?nP* = (*set-offending-flows ?G ?nP* ≠ {}) already hints this.

lemma *mono-imp-emptyoffending-eq-nevervalid*:
$\llbracket$ *sinvar-mono*; *wf-graph G*; ¬ *sinvar G nP*; *set-offending-flows G nP* = {}$\rrbracket$ $\Longrightarrow$
¬ (∃ *F* ⊆ *edges G*. *sinvar* (*delete-edges G F*) *nP*)
⟨*proof*⟩
end

3.2 Monotonicity of offending flows

context *SecurityInvariant-preliminaries*
begin

If there is some F' in the offending flows of a small graph and you have a bigger graph, you can extend F' by some *Fadd* and minimality in F is preserved

lemma *minimality-offending-flows-mono-edges-graph-extend*:
$\llbracket$ *wf-graph* (| *nodes* = V, *edges* = E |); $E' \subseteq E$; *Fadd* $\cap E' = \{\}$; $F' \in$ *set-offending-flows* (|*nodes* = V, *edges* = E'|) *nP* $\rrbracket \Longrightarrow$
($\forall$ (*e1*, *e2*)$\in F'$. $\neg$ *sinvar* (*add-edge e1 e2* (*delete-edges* (|*nodes* = V, *edges* = E |) ($F' \cup$ *Fadd*))) *nP*)
⟨*proof*⟩

The minimality condition of the offending flows also holds if we increase the graph.

corollary *minimality-offending-flows-mono-edges-graph*:
$\llbracket$ *wf-graph* (| *nodes* = V, *edges* = E |);
$E' \subseteq E$;
$F \in$ *set-offending-flows* (|*nodes* = V, *edges* = E'|) *nP* $\rrbracket \Longrightarrow$
$\forall$ (*e1*, *e2*)$\in F$. $\neg$ *sinvar* (*add-edge e1 e2* (*delete-edges* (|*nodes* = V, *edges* = E |) F)) *nP*
⟨*proof*⟩

all sets in the set of offending flows are monotonic, hence, for a larger graph, they can be extended to match the smaller graph. I.e. everything is monotonic.

theorem *mono-extend-set-offending-flows*: $\llbracket$ *wf-graph* (| *nodes* = V, *edges* = E |); $E' \subseteq E$; $F' \in$ *set-offending-flows* (| *nodes* = V, *edges* = E' |) *nP* $\rrbracket \Longrightarrow$
$\exists$ $F \in$ *set-offending-flows* (| *nodes* = V, *edges* = E |) *nP*. $F' \subseteq F$
⟨*proof*⟩

The offending flows are monotonic.

corollary *offending-flows-union-mono*: $\llbracket$ *wf-graph* (| *nodes* = V, *edges* = E |); $E' \subseteq E$ $\rrbracket \Longrightarrow$
$\bigcup$ (*set-offending-flows* (| *nodes* = V, *edges* = E' |) *nP*) $\subseteq \bigcup$ (*set-offending-flows* (| *nodes* = V, *edges* = E |) *nP*)
⟨*proof*⟩

lemma *set-offending-flows-insert-contains-new*:
$\llbracket$ *wf-graph* (| *nodes* = V, *edges* = *insert e E* |); *set-offending-flows* (|*nodes* = V, *edges* = E|) *nP* = $\{\}$; *set-offending-flows* (|*nodes* = V, *edges* = *insert e E*|) *nP* $\neq \{\}$ $\rrbracket \Longrightarrow$
$\{e\} \in$ *set-offending-flows* (|*nodes* = V, *edges* = *insert e E*|) *nP*
⟨*proof*⟩

end

value *Pow* {*1*::*int*, *2*, *3*} $\cup$ {{*8*}, {*9*}}
value $\bigcup$ *x*$\in$*Pow* {*1*::*int*, *2*, *3*}. $\bigcup$ *y* $\in$ {{*8*::*int*}, {*9*}}. {*x* $\cup$ *y*}

— combines powerset of A with B
definition *pow-combine* :: *'x set* $\Rightarrow$ *'x set set* $\Rightarrow$ *'x set set* **where**
pow-combine A B $\equiv$ ($\bigcup$ $X \in$ *Pow A*. $\bigcup$ $Y \in B$. $\{X \cup Y\}$) $\cup$ *Pow A*

value *pow-combine* {*1*::*int*,*2*} {{*5*::*int*, *6*}, {*8*}}
value *pow-combine* {*1*::*int*,*2*} {}

lemma *pow-combine-mono*:
fixes S :: *'a set set*
and X :: *'a set*
and Y :: *'a set*
assumes *a1*: $\forall\ F \in S.\ F \subseteq X$
shows $\forall\ F \in$ *pow-combine* $Y\ S.\ F \subseteq Y \cup X$
⟨*proof*⟩

lemma $S \subseteq$ *pow-combine* $X\ S$ ⟨*proof*⟩
lemma *Pow* $X \subseteq$ *pow-combine* $X\ S$ ⟨*proof*⟩

lemma *rule-pow-combine-fixfst*: $B \subseteq C \Longrightarrow$ *pow-combine* $A\ B \subseteq$ *pow-combine* $A\ C$
⟨*proof*⟩

value *pow-combine* {*1*::*int*,*2*} {{*5*::*int*, *6*}, {*1*}} ⊆ *pow-combine* {*1*::*int*,*2*} {{*5*::*int*, *6*}, {*8*}}

lemma *rule-pow-combine-fixfst-Union*: $\bigcup\ B \subseteq \bigcup\ C \Longrightarrow \bigcup$ (*pow-combine* $A\ B$) $\subseteq \bigcup$ (*pow-combine* $A\ C$)
⟨*proof*⟩

context *SecurityInvariant-preliminaries*
begin

lemma *offending-partition-subset-empty*:
assumes *a1*:$\forall\ F \in$ (*set-offending-flows* (\|*nodes* = V, *edges* = $E \cup X$\|) *nP*). $F \subseteq X$
and *wfGEX*: *wf-graph* (\|*nodes* = V, *edges* = $E \cup X$\|)
and *disj*: $E \cap X = \{\}$
shows (*set-offending-flows* (\|*nodes* = V, *edges* = E\|) *nP*) = {}
⟨*proof*⟩

corollary *partitioned-offending-subseteq-pow-combine*:
assumes *wfGEX*: *wf-graph* (\|*nodes* = V, *edges* = $E \cup X$\|)
and *disj*: $E \cap X = \{\}$
and *partitioned-offending*: $\forall\ F \in$ (*set-offending-flows* (\|*nodes* = V, *edges* = $E \cup X$\|) *nP*). $F \subseteq X$
shows (*set-offending-flows* (\|*nodes* = V, *edges* = $E \cup X$\|) *nP*) ⊆ *pow-combine* X (*set-offending-flows* (\|*nodes* = V, *edges* = E\|) *nP*)
⟨*proof*⟩
end

context *SecurityInvariant-preliminaries*
begin

Knowing that the $\bigcup$ *offending is* $\subseteq X$, removing something from the graphs's edges, it also disappears from the offending flows.

lemma *Un-set-offending-flows-bound-minus*:

assumes *wfG*: *wf-graph* $(\!|$ *nodes* = *V*, *edges* = *E* $|\!)$
and *Foffending*: $\bigcup$(*set-offending-flows* $(\!|$*nodes* = *V*, *edges* = *E*$|\!)$ *nP*) ⊆ *X*
shows $\bigcup$(*set-offending-flows* $(\!|$*nodes* = *V*, *edges* = *E* − {*f*}$|\!)$ *nP*) ⊆ *X* − {*f*}
⟨*proof*⟩

If the offending flows are bound by some X, the we can remove all finite $E'$from the graph's edges and the offending flows from the smaller graph are bound by $X - E'$.

lemma *Un-set-offending-flows-bound-minus-subseteq*:
assumes *wfG*: *wf-graph* $(\!|$ *nodes* = *V*, *edges* = *E* $|\!)$
and *Foffending*: $\bigcup$ (*set-offending-flows* $(\!|$*nodes* = *V*, *edges* = *E*$|\!)$ *nP*) ⊆ *X*
shows $\bigcup$ (*set-offending-flows* $(\!|$*nodes* = *V*, *edges* = *E* − *E′*$|\!)$ *nP*) ⊆ *X* − *E′*
⟨*proof*⟩

corollary *Un-set-offending-flows-bound-minus-subseteq′*:
⟦ *wf-graph* $(\!|$ *nodes* = *V*, *edges* = *E* $|\!)$;
$\bigcup$ (*set-offending-flows* $(\!|$ *nodes* = *V*, *edges* = *E* $|\!)$ *nP*) ⊆ *X* ⟧ ⟹
$\bigcup$ (*set-offending-flows* $(\!|$ *nodes* = *V*, *edges* = *E* − *E′* $|\!)$ *nP*) ⊆ *X* − *E′*
⟨*proof*⟩

end

end
theory *TopoS-ENF*
imports *Main TopoS-Interface Lib/TopoS-Util TopoS-withOffendingFlows*
begin

4 Special Structures of Security Invariants

Security Invariants may have a common structure: If the function *sinvar* is predicate which starts with ∀ (v_1, v_2) ∈ *edges G*. ..., we call this the all edges normal form (ENF). We found that this form has some nice properties. Also, locale instantiation is easier in ENF with the help of the following lemmata.

4.1 Simple Edges Normal Form (ENF)

context *SecurityInvariant-withOffendingFlows*
begin

definition *sinvar-all-edges-normal-form* :: (′*a* ⇒ ′*a* ⇒ *bool*) ⇒ *bool* **where**
sinvar-all-edges-normal-form P ≡ ∀ *G nP*. *sinvar G nP* = (∀ (*e1*, *e2*)∈ *edges G*. *P* (*nP e1*) (*nP e2*))

reflexivity is needed for convenience. If a security invariant is not reflexive, that means that all nodes with the default parameter ⊥ are not allowed to communicate with each other. Non-reflexivity is possible, but requires more work.

definition *ENF-refl* :: (′*a* ⇒ ′*a* ⇒ *bool*) ⇒ *bool* **where**
ENF-refl P ≡ *sinvar-all-edges-normal-form P* ∧ (∀ *p1*. *P p1 p1*)

lemma *monotonicity-sinvar-mono*: *sinvar-all-edges-normal-form P* ⟹ *sinvar-mono*
⟨*proof*⟩

end

4.1.1 Offending Flows

context *SecurityInvariant-withOffendingFlows*
begin

The insight: for all edges in the members of the offending flows, ¬ *P* holds.

lemma *ENF-offending-imp-not-P*:
assumes *sinvar-all-edges-normal-form P F* ∈ *set-offending-flows G nP* (*e1*, *e2*) ∈ *F*
shows ¬ *P* (*nP e1*) (*nP e2*)
⟨*proof*⟩

Hence, the members of *set-offending-flows* must look as follows.

lemma *ENF-offending-set-P-representation*:
assumes *sinvar-all-edges-normal-form P F* ∈ *set-offending-flows G nP*
shows *F* = {(*e1*,*e2*). (*e1*, *e2*) ∈ *edges G* ∧ ¬ *P* (*nP e1*) (*nP e2*)} (**is** *F* = *?E*)
⟨*proof*⟩

We can show left to right of the desired representation of *set-offending-flows*

lemma *ENF-offending-subseteq-lhs*:
assumes *sinvar-all-edges-normal-form P*
shows *set-offending-flows G nP* ⊆ { {(*e1*,*e2*). (*e1*, *e2*) ∈ *edges G* ∧ ¬ *P* (*nP e1*) (*nP e2*)} }
⟨*proof*⟩

if *set-offending-flows* is not empty, we have the other direction.

lemma *ENF-offending-not-empty-imp-ENF-offending-subseteq-rhs*:
assumes *sinvar-all-edges-normal-form P set-offending-flows G nP* ≠ {}
shows { {(*e1*,*e2*) ∈ *edges G*. ¬ *P* (*nP e1*) (*nP e2*)} } ⊆ *set-offending-flows G nP*
⟨*proof*⟩

lemma *ENF-notevalmodel-imp-offending-not-empty*:
sinvar-all-edges-normal-form P ⟹ ¬ *sinvar G nP* ⟹ *set-offending-flows G nP* ≠ {}

⟨*proof*⟩

lemma *ENF-offending-case1*:
⟦ *sinvar-all-edges-normal-form P*; ¬ *sinvar G nP* ⟧ ⟹
{ {(*e1*,*e2*). (*e1*, *e2*) ∈ (*edges G*) ∧ ¬ *P* (*nP e1*) (*nP e2*)} } = *set-offending-flows G nP*
⟨*proof*⟩

lemma *ENF-offending-case2*:
⟦ *sinvar-all-edges-normal-form P*; *sinvar G nP* ⟧ ⟹
{} = *set-offending-flows G nP*
⟨*proof*⟩

theorem *ENF-offending-set*:
⟦ *sinvar-all-edges-normal-form P* ⟧ ⟹
set-offending-flows G nP = (*if sinvar G nP then*
{}
else
{ {(*e1*,*e2*). (*e1*, *e2*) ∈ *edges G* ∧ ¬ *P* (*nP e1*) (*nP e2*)} })

⟨*proof*⟩
end

4.1.2 Lemmata

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENF-offending-members*:
⟦ ¬ *sinvar G nP*; *sinvar-all-edges-normal-form P*; *f* ∈ *set-offending-flows G nP*⟧ ⟹
f ⊆ (*edges G*) ∧ (∀ (*e1*, *e2*)∈ *f*. ¬ *P* (*nP e1*) (*nP e2*))
⟨*proof*⟩

4.1.3 Instance Helper

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENF-refl-not-offedning*:
⟦ ¬ *sinvar G nP*; *f* ∈ *set-offending-flows G nP*;
ENF-refl P⟧ ⟹
∀(*e1*,*e2*) ∈ *f*. *e1* ≠ *e2*
⟨*proof*⟩

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENF-default-update-fst*:
fixes *default-node-properties* :: ′*a* (‹⊥›)
assumes *modelInv*: ¬ *sinvar G nP*
and *ENFdef*: *sinvar-all-edges-normal-form P*
and *secdef*: ∀ (*nP*::′*v* ⇒ ′*a*) *e1 e2*. ¬ (*P* (*nP e1*) (*nP e2*)) ⟶ ¬ (*P* ⊥ (*nP e2*))
shows
¬ (∀ (*e1*, *e2*) ∈ *edges G*. *P* ((*nP*(*i* := ⊥)) *e1*) (*nP e2*))
⟨*proof*⟩

lemma (**in** *SecurityInvariant-withOffendingFlows*)
fixes *default-node-properties* :: ′*a* (‹⊥›)
shows ¬ *sinvar G nP* ⟹ *sinvar-all-edges-normal-form P* ⟹
(∀ (*nP*::′*v* ⇒ ′*a*) *e1 e2*. ¬ (*P* (*nP e1*) (*nP e2*)) ⟶ ¬ (*P* ⊥ (*nP e2*))) ⟹
(∀ (*nP*::′*v* ⇒ ′*a*) *e1 e2*. ¬ (*P* (*nP e1*) (*nP e2*)) ⟶ ¬ (*P* (*nP e1*) ⊥)) ⟹
(∀ (*nP*::′*v* ⇒ ′*a*) *e1 e2*. ¬ *P* ⊥ ⊥)
⟹ ¬ *sinvar G* (*nP*(*i* := ⊥))
⟨*proof*⟩

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENF-fsts-refl-instance*:
fixes *default-node-properties* :: ′*a* (‹⊥›)
assumes *a-enf-refl*: *ENF-refl P*
and *a3*: ∀ (*nP*::′*v* ⇒ ′*a*) *e1 e2*. ¬ (*P* (*nP e1*) (*nP e2*)) ⟶ ¬ (*P* ⊥ (*nP e2*))
and *a-offending*: *f* ∈ *set-offending-flows G nP*
and *a-i-fsts*: *i* ∈ *fst* ‘ *f*
shows
¬ *sinvar G* (*nP*(*i* := ⊥))
⟨*proof*⟩

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENF-snds-refl-instance*:
fixes *default-node-properties* :: ′*a* (‹⊥›)
assumes *a-enf-refl*: *ENF-refl P*
and *a3*: ∀ (*nP*::′*v* ⇒ ′*a*) *e1 e2*. ¬ (*P* (*nP e1*) (*nP e2*)) ⟶ ¬ (*P* (*nP e1*) ⊥)
and *a-offending*: *f* ∈ *set-offending-flows G nP*

and *a-i-snds*: $i \in snd$ ‘ f
shows
¬ *sinvar G* (*nP*(*i* := ⊥))
⟨*proof*⟩

4.2 edges normal form ENF with sender and receiver names

definition (**in** *SecurityInvariant-withOffendingFlows*) *sinvar-all-edges-normal-form-sr* :: (′*a* ⇒ ′*v* ⇒ ′*a* ⇒ ′*v* ⇒ *bool*) ⇒ *bool* **where**
sinvar-all-edges-normal-form-sr P ≡ ∀ *G nP*. *sinvar G nP* = (∀ (*s*, *r*)∈ *edges G*. *P* (*nP s*) *s* (*nP r*) *r*)

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENFsr-monotonicity-sinvar-mono*: ⟦ *sinvar-all-edges-normal-form-s* *P* ⟧ ⟹
sinvar-mono
⟨*proof*⟩

4.2.1 Offending Flows:

theorem (**in** *SecurityInvariant-withOffendingFlows*) *ENFsr-offending-set*:
assumes *ENFsr*: *sinvar-all-edges-normal-form-sr P*
shows *set-offending-flows G nP* = (*if sinvar G nP then*
{}
else
{ {(*s*,*r*). (*s*, *r*) ∈ *edges G* ∧ ¬ *P* (*nP s*) *s* (*nP r*) *r*} }) (**is** *?A* = *?B*)
⟨*proof*⟩

4.3 edges normal form not refl ENFnrSR

definition (**in** *SecurityInvariant-withOffendingFlows*) *sinvar-all-edges-normal-form-not-refl-SR* :: (′*a* ⇒ ′*v* ⇒ ′*a* ⇒ ′*v* ⇒ *bool*) ⇒ *bool* **where**
sinvar-all-edges-normal-form-not-refl-SR P ≡
∀ *G nP*. *sinvar G nP* = (∀ (*s*, *r*) ∈ *edges G*. *s* ≠ *r* ⟶ *P* (*nP s*) *s* (*nP r*) *r*)

we derive everything from the ENFnrSR form

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENFnrSR-to-ENFsr*:
sinvar-all-edges-normal-form-not-refl-SR P ⟹ *sinvar-all-edges-normal-form-sr* (λ *p1 v1 p2 v2*. *v1* ≠ *v2* ⟶ *P p1 v1 p2 v2*)
⟨*proof*⟩

4.3.1 Offending Flows

theorem (**in** *SecurityInvariant-withOffendingFlows*) *ENFnrSR-offending-set*:
⟦ *sinvar-all-edges-normal-form-not-refl-SR P* ⟧ ⟹
set-offending-flows G nP = (*if sinvar G nP then*
{}
else
{ {(*e1*,*e2*). (*e1*, *e2*) ∈ *edges G* ∧ *e1* ≠ *e2* ∧ ¬ *P* (*nP e1*) *e1* (*nP e2*) *e2*} })
⟨*proof*⟩

4.3.2 Instance helper

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENFnrSR-fsts-weakrefl-instance*:
fixes *default-node-properties* :: ′*a* (‹⊥›)

assumes *a-enf*: *sinvar-all-edges-normal-form-not-refl-SR P*
and *a-weakrefl*: $\forall$ *s r*. *P* $\bot$ *s* $\bot$ *r*
and *a-botdefault*: $\forall$ *s r*. (*nP r*) $\neq \bot \longrightarrow \neg$ *P* (*nP s*) *s* (*nP r*) *r* $\longrightarrow \neg$ *P* $\bot$ *s* (*nP r*) *r*
and *a-alltobot*: $\forall$ *s r*. *P* (*nP s*) *s* $\bot$ *r*
and *a-offending*: *f* $\in$ *set-offending-flows G nP*
and *a-i-fsts*: *i* $\in$ *fst' f*
shows
$\neg$ *sinvar G* (*nP*(*i* := $\bot$))
⟨*proof*⟩

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENFnrSR-snds-weakrefl-instance*:
fixes *default-node-properties* :: *'a* (‹$\bot$›)
assumes *a-enf*: *sinvar-all-edges-normal-form-not-refl-SR P*
and *a-weakrefl*: $\forall$ *s r*. *P* $\bot$ *s* $\bot$ *r*
and *a-botdefault*: $\forall$ *s r*. (*nP s*) $\neq \bot \longrightarrow \neg$ *P* (*nP s*) *s* (*nP r*) *r* $\longrightarrow \neg$ *P* (*nP s*) *s* $\bot$ *r*
and *a-bottoall*: $\forall$ *s r*. *P* $\bot$ *s* (*nP r*) *r*
and *a-offending*: *f* $\in$ *set-offending-flows G nP*
and *a-i-snds*: *i* $\in$ *snd' f*
shows
$\neg$ *sinvar G* (*nP*(*i* := $\bot$))
⟨*proof*⟩

4.4 edges normal form not refl ENFnr

definition (**in** *SecurityInvariant-withOffendingFlows*) *sinvar-all-edges-normal-form-not-refl* :: (*'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool*) $\Rightarrow$ *bool* **where**
sinvar-all-edges-normal-form-not-refl P $\equiv \forall$ *G nP*. *sinvar G nP* = ($\forall$ (*e1*, *e2*) $\in$ *edges G*. *e1* $\neq$ *e2* $\longrightarrow$ *P* (*nP e1*) (*nP e2*))

we derive everything from the ENFnrSR form

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENFnr-to-ENFnrSR*:
sinvar-all-edges-normal-form-not-refl P $\Longrightarrow$ *sinvar-all-edges-normal-form-not-refl-SR* (λ *v1* - *v2* -. *P v1 v2*)
⟨*proof*⟩

4.4.1 Offending Flows

theorem (**in** *SecurityInvariant-withOffendingFlows*) *ENFnr-offending-set*:
⟦ *sinvar-all-edges-normal-form-not-refl P* ⟧ $\Longrightarrow$
set-offending-flows G nP = (*if sinvar G nP then*
{}
else
{ {(*e1*,*e2*). (*e1*, *e2*) $\in$ *edges G* $\wedge$ *e1* $\neq$ *e2* $\wedge \neg$ *P* (*nP e1*) (*nP e2*)} })
⟨*proof*⟩

4.4.2 Instance helper

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENFnr-fsts-weakrefl-instance*:
fixes *default-node-properties* :: *'a* (‹$\bot$›)
assumes *a-enf*: *sinvar-all-edges-normal-form-not-refl P*
and *a-botdefault*: $\forall$ *e1 e2*. *e2* $\neq \bot \longrightarrow \neg$ *P e1 e2* $\longrightarrow \neg$ *P* $\bot$ *e2*

and *a-alltobot*: ∀ *e1*. *P e1* ⊥
and *a-offending*: *f* ∈ *set-offending-flows G nP*
and *a-i-fsts*: *i* ∈ *fst' f*
shows
¬ *sinvar G* (*nP*(*i* := ⊥))
⟨*proof*⟩

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENFnr-snds-weakrefl-instance*:
fixes *default-node-properties* :: *'a* (‹⊥›)
assumes *a-enf*: *sinvar-all-edges-normal-form-not-refl P*
and *a-botdefault*: ∀ *e1 e2*. ¬ *P e1 e2* ⟶ ¬ *P e1* ⊥
and *a-bottoall*: ∀ *e2*. *P* ⊥ *e2*
and *a-offending*: *f* ∈ *set-offending-flows G nP*
and *a-i-snds*: *i* ∈ *snd' f*
shows
¬ *sinvar G* (*nP*(*i* := ⊥))
⟨*proof*⟩

lemma (**in** *SecurityInvariant-withOffendingFlows*) *ENF-weakrefl-instance-FALSE*:
fixes *default-node-properties* :: *'a* (‹⊥›)
assumes *a-wfG*: *wf-graph G*
and *a-not-eval*: ¬ *sinvar G nP*
and *a-enf*: *sinvar-all-edges-normal-form P*
and *a-weakrefl*: *P* ⊥ ⊥
and *a-botisolated*: ⋀ *e2*. *e2* ≠ ⊥ ⟹ ¬ *P* ⊥ *e2*
and *a-botdefault*: ⋀ *e1 e2*. *e1* ≠ ⊥ ⟹ ¬ *P e1 e2* ⟹ ¬ *P e1* ⊥
and *a-offending*: *f* ∈ *set-offending-flows G nP*
and *a-offending-rm*: *sinvar* (*delete-edges G f*) *nP*
and *a-i-fsts*: *i* ∈ *snd' f*
and *a-not-eval-upd*: ¬ *sinvar G* (*nP*(*i* := ⊥))
shows *False*
⟨*proof*⟩

end
theory *vertex-example-simps*
imports *Lib/FiniteGraph TopoS-Vertices*
begin⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩**end**
theory *TopoS-Helper*
imports *Main TopoS-Interface*
TopoS-ENF
vertex-example-simps
begin

lemma (**in** *SecurityInvariant-preliminaries*) *sinvar-valid-remove-flattened-offending-flows*:
assumes *wf-graph* (|*nodes* = *nodesG*, *edges* = *edgesG*|)
shows *sinvar* (|*nodes* = *nodesG*, *edges* = *edgesG* − ⋃ (*set-offending-flows* (|*nodes* = *nodesG*, *edges* = *edgesG*|) *nP*) |) *nP*

⟨*proof*⟩

lemma (**in** *SecurityInvariant-preliminaries*) *sinvar-valid-remove-SOME-offending-flows*:
 assumes *set-offending-flows* (|*nodes* = *nodesG*, *edges* = *edgesG*|) *nP* ≠ {}
 shows *sinvar* (|*nodes* = *nodesG*, *edges* = *edgesG* − (*SOME F*. *F* ∈ *set-offending-flows* (|*nodes* = *nodesG*, *edges* = *edgesG*|) *nP*) |) *nP*
⟨*proof*⟩

lemma (**in** *SecurityInvariant-preliminaries*) *sinvar-valid-remove-minimalize-offending-overapprox*:
 assumes *wf-graph* (|*nodes* = *nodesG*, *edges* = *edgesG*|)
 and ¬ *sinvar* (|*nodes* = *nodesG*, *edges* = *edgesG*|) *nP*
 and *set Es* = *edgesG* **and** *distinct Es*
 shows *sinvar* (|*nodes* = *nodesG*, *edges* = *edgesG* −
 set (*minimalize-offending-overapprox Es* [] (|*nodes* = *nodesG*, *edges* = *edgesG*|) *nP*) |) *nP*
⟨*proof*⟩

end
theory *SINVAR-Subnets2*
imports../*TopoS-Helper*
begin

4.5 SecurityInvariant Subnets2

Warning, This is just a test. Please look at `SINVAR_Subnets.thy`. This security invariant has the following changes, compared to `SINVAR_Subnets.thy`: A new BorderRouter' is introduced which can send to the members of its subnet. A new InboundRouter is accessible by anyone. It can access all other routers and the outside.

datatype *subnets* = *Subnet nat* | *BorderRouter nat* | *BorderRouter' nat* | *InboundRouter* | *Unassigned*

definition *default-node-properties* :: *subnets*
 where *default-node-properties* ≡ *Unassigned*

fun *allowed-subnet-flow* :: *subnets* ⇒ *subnets* ⇒ *bool* **where**
 allowed-subnet-flow (*Subnet s1*) (*Subnet s2*) = (*s1* = *s2*) |
 allowed-subnet-flow (*Subnet s1*) (*BorderRouter s2*) = (*s1* = *s2*) |
 allowed-subnet-flow (*Subnet s1*) (*BorderRouter' s2*) = (*s1* = *s2*) |
 allowed-subnet-flow (*Subnet* -) - = *True* |
 allowed-subnet-flow (*BorderRouter* -) (*Subnet* -) = *False* |
 allowed-subnet-flow (*BorderRouter* -) - = *True* |
 allowed-subnet-flow (*BorderRouter' s1*) (*Subnet s2*) = (*s1* = *s2*) |
 allowed-subnet-flow (*BorderRouter'* -) - = *True* |
 allowed-subnet-flow InboundRouter (*Subnet* -) = *False* |
 allowed-subnet-flow InboundRouter - = *True* |
 allowed-subnet-flow Unassigned Unassigned = *True* |
 allowed-subnet-flow Unassigned InboundRouter = *True* |
 allowed-subnet-flow Unassigned - = *False*

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *subnets*) ⇒ *bool* **where**
 sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. *allowed-subnet-flow* (*nP e1*) (*nP e2*))

definition *receiver-violation* :: *bool* **where** *receiver-violation* = *False*

Only members of the same subnet or their *BorderRouter′* can access them.

lemma *allowed-subnet-flow a* (*Subnet s1*) $\Longrightarrow$ *a* = (*BorderRouter′ s1*) ∨ *a* = (*Subnet s1*)
⟨*proof*⟩

4.5.1 Preliminaries

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
⟨*proof*⟩

4.5.2 ENF

lemma *All-to-Unassigned*: ∀ *e1*. *allowed-subnet-flow e1 Unassigned*
⟨*proof*⟩
lemma *Unassigned-default-candidate*: ∀ *nP e1 e2*. ¬ *allowed-subnet-flow* (*nP e1*) (*nP e2*) ⟶ ¬ *allowed-subnet-flow Unassigned* (*nP e2*)
⟨*proof*⟩
lemma *allowed-subnet-flow-refl*: ∀ *e*. *allowed-subnet-flow e e*
⟨*proof*⟩
lemma *Subnets-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar allowed-subnet-flow*
⟨*proof*⟩
lemma *Subnets-ENF-refl*: *SecurityInvariant-withOffendingFlows.ENF-refl sinvar allowed-subnet-flow*
⟨*proof*⟩

definition *Subnets-offending-set*:: *′v graph* ⇒ (*′v* ⇒ *subnets*) ⇒ (*′v* × *′v*) *set set* **where**
Subnets-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ ¬ *allowed-subnet-flow* (*nP e1*) (*nP e2*)} })
lemma *Subnets-offending-set*:
SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = *Subnets-offending-set*
⟨*proof*⟩

interpretation *Subnets*: *SecurityInvariant-ACS*
where *default-node-properties* = *SINVAR-Subnets2.default-node-properties*
and *sinvar* = *SINVAR-Subnets2.sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *Subnets-offending-set*
⟨*proof*⟩

lemma *TopoS-Subnets2*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

hide-fact (**open**) *sinvar-mono*
hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-BLPstrict*
imports ../*TopoS-Helper*
begin

4.6 Stricter Bell LaPadula SecurityInvariant

All unclassified data sources must be labeled, default assumption: all is secret.

Warning: This is considered here an access control strategy. By default, everything is secret and one explicitly prohibits sending to non-secret hosts.

datatype *security-level* = *Unclassified* | *Confidential* | *Secret*

```
instantiation security-level :: linorder
  begin
  fun less-eq-security-level :: security-level ⇒ security-level ⇒ bool where
    (Unclassified ≤ Unclassified) = True |
    (Confidential ≤ Confidential) = True |
    (Secret ≤ Secret) = True |
    (Unclassified ≤ Confidential) = True |
    (Confidential ≤ Secret) = True  |
    (Unclassified ≤ Secret) = True  |
    (Secret ≤ Confidential) = False  |
    (Confidential ≤ Unclassified) = False  |
    (Secret ≤ Unclassified) = False

  fun less-security-level :: security-level ⇒ security-level ⇒ bool where
    (Unclassified < Unclassified) = False |
    (Confidential < Confidential) = False |
    (Secret < Secret) = False |
    (Unclassified < Confidential) = True |
    (Confidential < Secret) = True  |
    (Unclassified < Secret) = True  |
    (Secret < Confidential) = False  |
    (Confidential < Unclassified) = False  |
    (Secret < Unclassified) = False
  instance
    ⟨proof⟩
  end
```

```
definition default-node-properties :: security-level
  where  default-node-properties ≡ Secret
```

```
fun sinvar :: 'v graph ⇒ ('v ⇒ security-level) ⇒ bool where
  sinvar G nP = (∀ (e1,e2) ∈ edges G. (nP e1) ≤ (nP e2))
```

definition *receiver-violation* :: *bool* **where** *receiver-violation* ≡ *False*

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
⟨*proof*⟩

4.7 ENF

lemma *secret-default-candidate*: $\bigwedge$ (*nP*::(′*v* ⇒ *security-level*)) *e1 e2*. ¬ (*nP e1*) ≤ (*nP e2*) ⟹ ¬ *Secret* ≤ (*nP e2*)
⟨*proof*⟩
lemma *BLP-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar* (≤)
⟨*proof*⟩
lemma *BLP-ENF-refl*: *SecurityInvariant-withOffendingFlows.ENF-refl sinvar* (≤)
⟨*proof*⟩

definition *BLP-offending-set*:: ′*v graph* ⇒ (′*v* ⇒ *security-level*) ⇒ (′*v* × ′*v*) *set set* **where**
BLP-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ (*nP e1*) > (*nP e2*)} })
lemma *BLP-offending-set*: *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *BLP-offending-set*
⟨*proof*⟩

interpretation *BLPstrict*: *SecurityInvariant-ACS sinvar default-node-properties*

rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *BLP-offending-set*
⟨*proof*⟩

lemma *TopoS-BLPstrict*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

hide-fact (**open**) *sinvar-mono*

hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-Tainting*
imports ../*TopoS-Helper*
begin

4.8 SecurityInvariant Tainting for IFS

context
begin

qualified type-synonym *taints* = *string set*

Warning: an infinite set has cardinality 0

lemma *card* (*UNIV*::*taints*) = *0* ⟨*proof*⟩ **definition** *default-node-properties* :: *taints*

where *default-node-properties* ≡ {}

For all nodes n in the graph, for all nodes r which are reachable from n, node n needs the appropriate tainting fields which are set by r

definition *sinvar-tainting* :: *'v graph* ⇒ (*'v* ⇒ *taints*) ⇒ *bool* **where**
sinvar-tainting G nP ≡ ∀ *n* ∈ (*nodes G*). ∀ *r* ∈ (*succ-tran G n*). *nP n* ⊆ *nP r*

private lemma *sinvar-tainting-edges-def*: *wf-graph G* ⟹
sinvar-tainting G nP ⟷ (∀ (*v1*,*v2*) ∈ *edges G*. ∀ *r* ∈ (*succ-tran G v1*). *nP v1* ⊆ *nP r*)
⟨*proof*⟩

Alternative definition of the *sinvar-tainting*

qualified definition *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *taints*) ⇒ *bool* **where**
sinvar G nP ≡ ∀ (*v1*,*v2*) ∈ *edges G*. *nP v1* ⊆ *nP v2*

qualified lemma *sinvar-preferred-def*:
wf-graph G ⟹ *sinvar-tainting G nP* = *sinvar G nP*
⟨*proof*⟩

Information Flow Security

qualified definition *receiver-violation* :: *bool* **where** *receiver-violation* ≡ *True*

private lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
⟨*proof*⟩
interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
⟨*proof*⟩ **lemma** *Taints-def-unique*: *otherbot* ≠ {} ⟹
∃ *G p i f*. *wf-graph G* ∧ ¬ *sinvar G p* ∧ *f* ∈ (*SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G p*) ∧
sinvar (*delete-edges G f*) *p* ∧
i ∈ *snd* ‘ *f* ∧ *sinvar G* (*p*(*i* := *otherbot*))
⟨*proof*⟩

4.8.1 ENF

private lemma *Taints-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar* (⊆)
⟨*proof*⟩ **lemma** *Taints-ENF-refl*: *SecurityInvariant-withOffendingFlows.ENF-refl sinvar* (⊆)
⟨*proof*⟩ **definition** *Taints-offending-set*:: *'v graph* ⇒ (*'v* ⇒ *taints*) ⇒ (*'v* × *'v*) *set set* **where**
Taints-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ ¬ (*nP e1*) ⊆ (*nP e2*)} })
lemma *Taints-offending-set*: *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *Taints-offending-set*
⟨*proof*⟩

interpretation *Taints*: *SecurityInvariant-IFS sinvar default-node-properties*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *Taints-offending-set*

⟨*proof*⟩

lemma *TopoS-Tainting*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

end

end
theory *SINVAR-BLPbasic*
imports ../*TopoS-Helper*
begin

4.9 SecurityInvariant Basic Bell LaPadula

type-synonym *security-level* = *nat*

definition *default-node-properties* :: *security-level*
where *default-node-properties* ≡ *0*

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *security-level*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. (*nP e1*) ≤ (*nP e2*))

What we call a *security-level* is also referred to as security label (or security clearance of subjects and classification of objects) in the literature. The lowest security level is *0*, which can be understood as unclassified. Consequently, 1 = confidential, 2 = secret, 3 = topSecret, ... The total order of the security levels corresponds to the total order of the natural numbers ≤. It is important that there is smallest security level (i.e. *default-node-properties*), otherwise, a unique and secure default parameter could not exist. Hence, it is not possible to extend the security levels to *int* to model unlimited "un-confidentialness".

definition *receiver-violation* :: *bool* **where** *receiver-violation* ≡ *True*

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
⟨*proof*⟩

lemma *BLP-def-unique*: *otherbot* ≠ *0* ⟹
∃ *G p i f*. *wf-graph G* ∧ ¬ *sinvar G p* ∧ *f* ∈ (*SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G p*) ∧
sinvar (*delete-edges G f*) *p* ∧
i ∈ *snd* ' *f* ∧ *sinvar G* (*p*(*i* := *otherbot*))
⟨*proof*⟩

4.9.1 ENF

lemma *zero-default-candidate*: ⋀ *nP e1 e2*. ¬ ((≤)::*security-level* ⇒ *security-level* ⇒ *bool*) (*nP e1*) (*nP e2*) ⟹ ¬ (≤) (*nP e1*) *0*

⟨*proof*⟩
lemma *zero-default-candidate-rule*: $\bigwedge$ (*nP*::(′*v* ⇒ *security-level*)) *e1 e2*. ¬ (*nP e1*) ≤ (*nP e2*) ⟹ ¬ (*nP e1*) ≤ *0*
⟨*proof*⟩
lemma *privacylevel-refl*: ((≤)::*security-level* ⇒ *security-level* ⇒ *bool*) *e e*
⟨*proof*⟩
lemma *BLP-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar* (≤)
⟨*proof*⟩
lemma *BLP-ENF-refl*: *SecurityInvariant-withOffendingFlows.ENF-refl sinvar* (≤)
⟨*proof*⟩

definition *BLP-offending-set*:: ′*v graph* ⇒ (′*v* ⇒ *security-level*) ⇒ (′*v* × ′*v*) *set set* **where**
BLP-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ (*nP e1*) > (*nP e2*)} })
lemma *BLP-offending-set*: *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *BLP-offending-set*
⟨*proof*⟩

interpretation *BLPbasic*: *SecurityInvariant-IFS sinvar default-node-properties*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *BLP-offending-set*
⟨*proof*⟩

lemma *TopoS-BLPBasic*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

Alternate definition of the *sinvar*: For all reachable nodes, the security level is higher

lemma *sinvar-BLPbasic-tancl*:
wf-graph G ⟹ *sinvar G nP* = (∀ *v* ∈ *nodes G*. ∀ *v*′ ∈ *succ-tran G v*. (*nP v*) ≤ (*nP v*′))
⟨*proof*⟩

hide-fact (**open**) *sinvar-mono*
hide-fact *BLP-def-unique zero-default-candidate zero-default-candidate-rule privacylevel-refl BLP-ENF BLP-ENF-refl*

hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-TaintingTrusted*
imports ../*TopoS-Helper*
begin

4.10 SecurityInvariant Tainting with Untainting-Feature for IFS

context
begin
qualified datatype *taints-raw* = *TaintsUntaints-Raw* (*taints-raw*: *string set*) (*untaints-raw*: *string set*)

The *untaints-raw* set must be a subset of *taints-raw*. Otherwise, there can be entries in the untaints set, which do not affect anything. This is certainly undesirable. In addition, a unique default parameter cannot exist if we allow such dead entries.

qualified typedef *taints* = {*ts*::*taints-raw*. *untaints-raw ts* ⊆ *taints-raw ts*}
morphisms *raw-of-taints Abs-taints*
⟨*proof*⟩

setup-lifting *type-definition-taints*

lemma *taints-eq-iff*:
tsx = *tsy* ⟷ *raw-of-taints tsx* = *raw-of-taints tsy*
⟨*proof*⟩

definition *taints* :: *taints* ⇒ *string set* **where**
taints ts ≡ *taints-raw* (*raw-of-taints ts*)
definition *untaints* :: *taints* ⇒ *string set* **where**
untaints ts ≡ *untaints-raw* (*raw-of-taints ts*)

lemma *taints-wellformedness*: *untaints ts* ⊆ *taints ts*
⟨*proof*⟩

Constructor for *taints*:

definition *TaintsUntaints* :: *string set* ⇒ *string set* ⇒ *taints* **where**
TaintsUntaints ts uts = *Abs-taints* (*TaintsUntaints-Raw* (*ts* ∪ *uts*) *uts*)

lemma *raw-of-taints-TaintsUntaints*:
raw-of-taints (*TaintsUntaints ts uts*) = (*TaintsUntaints-Raw* (*ts* ∪ *uts*) *uts*)
⟨*proof*⟩

lemma *taints-TaintsUntaints*[*code*]: *taints* (*TaintsUntaints ts uts*) = *ts* ∪ *uts*
⟨*proof*⟩
lemma *untaints-TaintsUntaints*[*code*]: *untaints* (*TaintsUntaints ts uts*) = *uts*
⟨*proof*⟩

The things in the first set are tainted, those in the second set are untainted. For example, a machine produces *''foo''*: *TaintsUntaints* {*''foo''*} {}

For example, a machine consumes *''foo''* and *''bar''*, combines them in a way that they are no longer critical and outputs *''baz''*: *TaintsUntaints* {*''foo''*, *''bar''*, *''baz''*} {*''foo''*, *''bar''*} abbreviated: *TaintsUntaints* {*''baz''*} {*''foo''*, *''bar''*}

lemma *TaintsUntaints* {*''foo''*, *''bar''*, *''baz''*} {*''foo''*, *''bar''*} =
TaintsUntaints {*''baz''*} {*''foo''*, *''bar''*}
⟨*proof*⟩ **definition** *default-node-properties* :: *taints*
where *default-node-properties* ≡ *TaintsUntaints* {} {}

qualified definition *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *taints*) ⇒ *bool* **where**
sinvar G nP ≡ ∀ (*v1*,*v2*) ∈ *edges G*.
taints (*nP v1*) − *untaints* (*nP v1*) ⊆ *taints* (*nP v2*)

Information Flow Security

qualified definition *receiver-violation* :: *bool* **where** *receiver-violation* ≡ *True*

private lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
⟨*proof*⟩

Needs the well-formedness condition that *untaints otherbot* ⊆ *taints otherbot*

private lemma *Taints-def-unique*: *otherbot* ≠ *default-node-properties* ⟹
∃ *G p i f*. *wf-graph G* ∧ ¬ *sinvar G p* ∧ *f* ∈ (*SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G p*) ∧
sinvar (*delete-edges G f*) *p* ∧
i ∈ *snd* ' *f* ∧ *sinvar G* (*p*(*i* := *otherbot*))
⟨*proof*⟩

4.10.1 ENF

private lemma *Taints-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form*
sinvar (λ*c1 c2*. *taints c1* − *untaints c1* ⊆ *taints c2*)
⟨*proof*⟩ **lemma** *Taints-ENF-refl*: *SecurityInvariant-withOffendingFlows.ENF-refl*
sinvar (λ*c1 c2*. *taints c1* − *untaints c1* ⊆ *taints c2*)
⟨*proof*⟩ **definition** *Taints-offending-set*:: *'v graph* ⇒ (*'v* ⇒ *taints*) ⇒ (*'v* × *'v*) *set set* **where**
Taints-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ ¬ *taints* (*nP e1*) − *untaints* (*nP e1*) ⊆ *taints* (*nP e2*)} })
lemma *Taints-offending-set*: *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *Taints-offending-set*
⟨*proof*⟩

interpretation *Taints*: *SecurityInvariant-IFS sinvar default-node-properties*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *Taints-offending-set*
⟨*proof*⟩

lemma *TopoS-TaintingTrusted*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

end

code-datatype *TaintsUntaints*

value[*code*] *TaintsUntaints* {''*foo*''} {''*bar*''}

value[*code*] *taints* (*TaintsUntaints* {''*foo*''} {''*bar*''})

end
theory *SINVAR-BLPtrusted*
imports ../*TopoS-Helper*
begin

4.11 SecurityInvariant Basic Bell LaPadula with trusted entities

type-synonym *security-level* = *nat*

record *node-config* =
security-level::*security-level*
trusted::*bool*

definition *default-node-properties* :: *node-config*
where *default-node-properties* ≡ (| *security-level* = *0*, *trusted* = *False* |)

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *node-config*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. (*if trusted* (*nP e2*) *then True else security-level* (*nP e1*) ≤ *security-level* (*nP e2*)))

A simplified version of the Bell LaPadula model was presented in `SINVAR_BLPbasic.thy`. In this theory, we extend this template with a notion of trust by adding a Boolean flag *trusted* to the host attributes. This is a refinement to represent real-world scenarios more accurately and analogously happened to the original Bell LaPadula model (see publication "Looking Back at the Bell-La Padula Model" A trusted host can receive information of any security level and may declassify it, i.e. distribute the information with its own security level. For example, a *trusted sc* = *True* host is allowed to receive any information and with the *0* level, it is allowed to reveal it to anyone.

definition *receiver-violation* :: *bool* **where** *receiver-violation* ≡ *True*

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
⟨*proof*⟩

lemma *a* ≠ *b* ⟹ ((∃ *x*. *y x*)) ⟹ ((∀ *x*. ¬ *y x*) ⟹ *a* = *b*) ⟨*proof*⟩

lemma *BLP-def-unique*: *otherbot* ≠ *default-node-properties* ⟹
∃ *G p i f*. *wf-graph G* ∧ ¬ *sinvar G p* ∧ *f* ∈ (*SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G p*) ∧
sinvar (*delete-edges G f*) *p* ∧
i ∈ *snd* ‘ *f* ∧ *sinvar G* (*p*(*i* := *otherbot*))
⟨*proof*⟩

4.11.1 ENF

definition *BLP-P* **where** *BLP-P* ≡ (λ*n1 n2*.(*if trusted n2 then True else security-level n1* ≤ *security-level n2*))
lemma *zero-default-candidate*: ∀ *nP e1 e2*. ¬ *BLP-P* (*nP e1*) (*nP e2*) ⟶ ¬ *BLP-P* (*nP e1*) *default-node-properties*
⟨*proof*⟩
lemma *privacylevel-refl*: *BLP-P e e*
⟨*proof*⟩
lemma *BLP-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar BLP-P*
⟨*proof*⟩
lemma *BLP-ENF-refl*: *SecurityInvariant-withOffendingFlows.ENF-refl sinvar BLP-P*
⟨*proof*⟩

definition *BLP-offending-set*:: *'v graph* ⇒ (*'v* ⇒ *node-config*) ⇒ (*'v* × *'v*) *set set* **where**
BLP-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ ¬ *BLP-P* (*nP e1*) (*nP e2*)} })
lemma *BLP-offending-set*: *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *BLP-offending-set*
⟨*proof*⟩

interpretation *BLPtrusted*: *SecurityInvariant-IFS*
where *default-node-properties* = *default-node-properties*
and *sinvar* = *sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *BLP-offending-set*
⟨*proof*⟩

lemma *TopoS-BLPtrusted*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

hide-type (**open**) *node-config*
hide-const (**open**) *sinvar-mono*

hide-const (**open**) *BLP-P*
hide-fact *BLP-def-unique zero-default-candidate privacylevel-refl BLP-ENF BLP-ENF-refl*

hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *Analysis-Tainting*
imports *SINVAR-Tainting SINVAR-BLPbasic*
SINVAR-TaintingTrusted SINVAR-BLPtrusted
begin

term *SINVAR-Tainting.sinvar*
term *SINVAR-BLPbasic.sinvar*

lemma *tainting-imp-blp-cutcard*: ∀ *ts v*. *nP v* = *ts* ⟶ *finite ts* ⟹
SINVAR-Tainting.sinvar G nP ⟹ *SINVAR-BLPbasic.sinvar G* ((λ*ts*. *card* (*ts* ∩ *X*)) ∘ *nP*)
⟨*proof*⟩

lemma *tainting-imp-blp-cutcard2*: *finite X* ⟹
SINVAR-Tainting.sinvar G nP ⟹ *SINVAR-BLPbasic.sinvar G* ((λ*ts*. *card* (*ts* ∩ *X*)) ∘ *nP*)
⟨*proof*⟩

lemma ∀ *ts v*. *nP v* = *ts* ⟶ *finite ts* ⟹
SINVAR-Tainting.sinvar G nP ⟹ *SINVAR-BLPbasic.sinvar G* (*card* ∘ *nP*)
⟨*proof*⟩

lemma $\forall\, b \in snd\ `\ edges\ G.\ finite\ (nP\ b) \Longrightarrow$
$SINVAR\text{-}Tainting.sinvar\ G\ nP \Longrightarrow SINVAR\text{-}BLPbasic.sinvar\ G\ (card \circ nP)$
$\langle proof \rangle$

One tainting invariant is equal to many BLP invariants. The BLP invariants are the projection of the tainting mapping for exactly one label

lemma *tainting-iff-blp*:
defines $extract \equiv \lambda a\ ts.\ if\ a \in ts\ then\ 1{::}security\text{-}level\ else\ 0{::}security\text{-}level$
shows $SINVAR\text{-}Tainting.sinvar\ G\ nP \longleftrightarrow (\forall\, a.\ SINVAR\text{-}BLPbasic.sinvar\ G\ (extract\ a \circ nP))$
$\langle proof \rangle$

If the labels are finite, the above can be generalized to arbitrary subsets of tainting labels.

lemma *tainting-iff-blp-extended*:
defines $extract \equiv \lambda A\ ts.\ card\ (A \cap ts)$
assumes *finite*: $\forall\, ts\ v.\ nP\ v = ts \longrightarrow finite\ ts$
shows $SINVAR\text{-}Tainting.sinvar\ G\ nP \longleftrightarrow (\forall\, A.\ SINVAR\text{-}BLPbasic.sinvar\ G\ (extract\ A \circ nP))$
$\langle proof \rangle$

Translated to the Bell LaPadula model with trust: security level is the number of tainted minus the untainted things We set the Trusted flag if a machine untaints things.

lemma $\forall\, ts\ v.\ nP\ v = ts \longrightarrow finite\ (taints\ ts) \Longrightarrow$
$SINVAR\text{-}TaintingTrusted.sinvar\ G\ nP \Longrightarrow$
$SINVAR\text{-}BLPtrusted.sinvar\ G\ ((\lambda\ ts.\ (\!|security\text{-}level = card\ (taints\ ts - untaints\ ts),\ trusted = (untaints\ ts \neq \{\})|\!)\ ) \circ nP)$
$\langle proof \rangle$

lemma *tainting-iff-blp-trusted*:
defines $project \equiv \lambda a\ ts.\ (\!|$
$security\text{-}level =$
if
$a \in (taints\ ts - untaints\ ts)$
$then$
$1{::}security\text{-}level$
$else$
$0{::}security\text{-}level$
$,\ trusted = a \in untaints\ ts|\!)$
shows $SINVAR\text{-}TaintingTrusted.sinvar\ G\ nP \longleftrightarrow (\forall\, a.\ SINVAR\text{-}BLPtrusted.sinvar\ G\ (project\ a \circ nP))$
$\langle proof \rangle$

If the labels are finite, the above can be generalized to arbitrary subsets of tainting labels.

lemma *tainting-iff-blp-trusted-extended*:
defines $project \equiv \lambda A\ ts.$
$(\!|$
$security\text{-}level = card\ (A \cap (taints\ ts - untaints\ ts))$
$,\ trusted = (A \cap untaints\ ts) \neq \{\}$
$|\!)$
assumes *finite*: $\forall\, ts\ v.\ nP\ v = ts \longrightarrow finite\ (taints\ ts)$
shows $SINVAR\text{-}TaintingTrusted.sinvar\ G\ nP \longleftrightarrow (\forall\, A.\ SINVAR\text{-}BLPtrusted.sinvar\ G\ (project\ A \circ nP))$
$\langle proof \rangle$

end
theory *TopoS-Interface-impl*
imports *Lib/FiniteGraph Lib/FiniteListGraph TopoS-Interface TopoS-Helper*
begin

5 Executable Implementation with Lists

Correspondence List Implementation and set Specification

5.1 Abstraction from list implementation to set specification

Nomenclature: *-spec* is the specification, *-impl* the corresponding implementation.

-spec and *-impl* only need to comply for *wf-graph*s. We will always require the stricter *wf-list-graph*, which implies *wf-graph*.

```
lemma wf-list-graph G ⟹ wf-graph (list-graph-to-graph G)

locale TopoS-List-Impl =
  fixes default-node-properties :: 'a (‹⊥›)
  and sinvar-spec::('v::vertex) graph ⇒ ('v::vertex ⇒ 'a) ⇒ bool
  and sinvar-impl::('v::vertex) list-graph ⇒ ('v::vertex ⇒ 'a) ⇒ bool
  and receiver-violation :: bool
  and offending-flows-impl::('v::vertex) list-graph ⇒ ('v ⇒ 'a) ⇒ ('v × 'v) list list
  and node-props-impl::('v::vertex, 'a) TopoS-Params ⇒ ('v ⇒ 'a)
  and eval-impl::('v::vertex) list-graph ⇒ ('v, 'a) TopoS-Params ⇒ bool
  assumes
      spec: SecurityInvariant sinvar-spec default-node-properties receiver-violation — specification is
valid
  and
     sinvar-spec-impl: wf-list-graph G ⟹
       (sinvar-spec (list-graph-to-graph G) nP) = (sinvar-impl G nP)
  and
     offending-flows-spec-impl: wf-list-graph G ⟹
    (SecurityInvariant-withOffendingFlows.set-offending-flows sinvar-spec (list-graph-to-graph G) nP)
=
     set`set (offending-flows-impl G nP)
  and
     node-props-spec-impl:
    SecurityInvariant.node-props-formaldef default-node-properties P = node-props-impl P
  and
     eval-spec-impl:
    (distinct (nodesL G) ∧ distinct (edgesL G) ∧
    SecurityInvariant.eval sinvar-spec default-node-properties (list-graph-to-graph G) P ) =
    (eval-impl G P)
```

5.2 Security Invariants Packed

We pack all necessary functions and properties of a security invariant in a struct-like data structure.

```
record ('v::vertex, 'a) TopoS-packed =
  nm-name :: string
```

nm-receiver-violation :: *bool*
nm-default :: *'a*
nm-sinvar::(*'v*::*vertex*) *list-graph* ⇒ (*'v* ⇒ *'a*) ⇒ *bool*
nm-offending-flows::(*'v*::*vertex*) *list-graph* ⇒ (*'v* ⇒ *'a*) ⇒ (*'v* × *'v*) *list list*
nm-node-props::(*'v*::*vertex*, *'a*) *TopoS-Params* ⇒ (*'v* ⇒ *'a*)
nm-eval::(*'v*::*vertex*) *list-graph* ⇒ (*'v*, *'a*) *TopoS-Params* ⇒ *bool*

The packed list implementation must comply with the formal definition.

locale *TopoS-modelLibrary* =
fixes *m* :: (*'v*::*vertex*, *'a*) *TopoS-packed* — concrete model implementation
and *sinvar-spec*::(*'v*::*vertex*) *graph* ⇒ (*'v*::*vertex* ⇒ *'a*) ⇒ *bool* — specification
assumes
name-not-empty: *length* (*nm-name m*) > *0*
and
impl-spec: *TopoS-List-Impl*
(*nm-default m*)
sinvar-spec
(*nm-sinvar m*)
(*nm-receiver-violation m*)
(*nm-offending-flows m*)
(*nm-node-props m*)
(*nm-eval m*)

5.3 Helpful Lemmata

show that *sinvar* complies

lemma *TopoS-eval-impl-proofrule*:
assumes *inst*: *SecurityInvariant sinvar-spec default-node-properties receiver-violation*
assumes *ev*: ⋀*nP*. *wf-list-graph G* ⟹ *sinvar-spec* (*list-graph-to-graph G*) *nP* = *sinvar-impl G nP*
shows
(*distinct* (*nodesL G*) ∧ *distinct* (*edgesL G*) ∧
SecurityInvariant.eval sinvar-spec default-node-properties (*list-graph-to-graph G*) *P*) =
(*wf-list-graph G* ∧ *sinvar-impl G* (*SecurityInvariant.node-props default-node-properties P*))
⟨*proof*⟩

5.4 Helper lemmata

Provide *sinvar* function and get back a function that computes the list of offending flows

Exponential time!

definition *Generic-offending-list*:: (*'v list-graph* ⇒ (*'v* ⇒ *'a*) ⇒ *bool*)⇒ *'v list-graph* ⇒ (*'v* ⇒ *'a*) ⇒ (*'v* × *'v*) *list list* **where**
Generic-offending-list sinvar G nP = [*f* ← (*subseqs* (*edgesL G*)).
(¬ *sinvar G nP* ∧ *sinvar* (*FiniteListGraph.delete-edges G f*) *nP*) ∧
(∀(*e1*, *e2*)∈*set f*. ¬ *sinvar* (*add-edge e1 e2* (*FiniteListGraph.delete-edges G f*)) *nP*)]

proof rule: if *sinvar* complies, *Generic-offending-list* complies

lemma *Generic-offending-list-correct*:
assumes *valid*: *wf-list-graph G*
assumes *spec-impl*: ⋀*G nP*. *wf-list-graph G* ⟹ *sinvar-spec* (*list-graph-to-graph G*) *nP* = *sinvar-impl G nP*
shows *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar-spec* (*list-graph-to-graph G*) *nP* =

set'set(Generic-offending-list sinvar-impl G nP)
⟨*proof*⟩

lemma *all-edges-list-I*: *P (list-graph-to-graph G) = Pl G* ⟹
(∀ *(e1, e2)*∈ *(edges (list-graph-to-graph G)). P (list-graph-to-graph G) e1 e2)* = (∀ *(e1, e2)*∈*set (edgesL G). Pl G e1 e2)*
⟨*proof*⟩

lemma *all-nodes-list-I*: *P (list-graph-to-graph G) = Pl G* ⟹
(∀ *n* ∈ *(nodes (list-graph-to-graph G)). P (list-graph-to-graph G) n)* = (∀ *n* ∈*set (nodesL G). Pl G n)*
⟨*proof*⟩

fun *minimalize-offending-overapprox* :: *('v list-graph* ⇒ *bool)* ⇒
('v × *'v) list* ⇒ *('v* × *'v) list* ⇒ *'v list-graph* ⇒ *('v* × *'v) list* **where**
minimalize-offending-overapprox - [] keep - = keep |
minimalize-offending-overapprox m (f#fs) keep G = (if m (delete-edges G (fs@keep)) then
minimalize-offending-overapprox m fs keep G
else
minimalize-offending-overapprox m fs (f#keep) G
)

thm *minimalize-offending-overapprox-boundnP*
lemma *minimalize-offending-overapprox-spec-impl*:
assumes *valid*: *wf-list-graph (G::'v::vertex list-graph)*
and *spec-impl*: ⋀*G nP::('v* ⇒ *'a). wf-list-graph G* ⟹ *sinvar-spec (list-graph-to-graph G) nP = sinvar-impl G nP*
shows *minimalize-offending-overapprox (λG. sinvar-impl G nP) fs keeps G* =
TopoS-withOffendingFlows.minimalize-offending-overapprox (λG. sinvar-spec G nP) fs keeps (list-graph-to-graph G)
⟨*proof*⟩

With *TopoS-Interface-impl.minimalize-offending-overapprox*, we can get one offending flow

lemma *minimalize-offending-overapprox-gives-some-offending-flow*:
assumes *wf*: *wf-list-graph G*
and *NetModelLib*: *TopoS-modelLibrary m sinvar-spec*
and *violation*: ¬ *(nm-sinvar m) G nP*
shows *set (minimalize-offending-overapprox (λG. (nm-sinvar m) G nP) (edgesL G) [] G)* ∈
SecurityInvariant-withOffendingFlows.set-offending-flows sinvar-spec (list-graph-to-graph G) nP
⟨*proof*⟩

6 Security Invariant Library

end
theory *SINVAR-BLPbasic-impl*
imports *SINVAR-BLPbasic ../TopoS-Interface-impl*
begin

6.0.1 SecurityInvariant BLPbasic List Implementation

code-identifier code-module *SINVAR-BLPbasic-impl* => (*Scala*) *SINVAR-BLPbasic*

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *security-level*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *set* (*edgesL G*). (*nP e1*) ≤ (*nP e2*))

definition *BLP-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *security-level*) ⇒ (*'v* × *'v*) *list list* **where**
BLP-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G*. *case e of* (*e1*,*e2*) ⇒ (*nP e1*) > (*nP e2*)]])

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-BLPbasic.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-BLPbasic.default-node-properties P* = *NetModel-node-props P*
⟨*proof*⟩

definition *BLP-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-BLPbasic.default-node-properties P*))

interpretation *BLPbasic-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-BLPbasic.default-node-properties*
and *sinvar-spec*=*SINVAR-BLPbasic.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-BLPbasic.receiver-violation*
and *offending-flows-impl*=*BLP-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*BLP-eval*
⟨*proof*⟩

6.0.2 BLPbasic packing

definition *SINVAR-LIB-BLPbasic* :: (*'v*::*vertex*, *security-level*) *TopoS-packed* **where**
SINVAR-LIB-BLPbasic ≡
⦇ *nm-name* = *''BLPbasic''*,
nm-receiver-violation = *SINVAR-BLPbasic.receiver-violation*,
nm-default = *SINVAR-BLPbasic.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *BLP-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *BLP-eval*
⦈
interpretation *SINVAR-LIB-BLPbasic-interpretation*: *TopoS-modelLibrary SINVAR-LIB-BLPbasic*

SINVAR-BLPbasic.sinvar
⟨*proof*⟩

6.0.3 Example

definition *fabNet* :: *string list-graph* **where**

fabNet ≡ (| *nodesL* = [''*Statistics*'', ''*SensorSink*'', ''*PresenceSensor*'', ''*Webcam*'', ''*TempSensor*'', ''*FireSesnsor*'',
''*MissionControl1*'', ''*MissionControl2*'', ''*Watchdog*'', ''*Bot1*'', ''*Bot2*''],
edgesL =[(''*PresenceSensor*'', ''*SensorSink*''), (''*Webcam*'', ''*SensorSink*''),
(''*TempSensor*'', ''*SensorSink*''), (''*FireSesnsor*'', ''*SensorSink*''),
(''*SensorSink*'', ''*Statistics*''),
(''*MissionControl1*'', ''*Bot1*''), (''*MissionControl1*'', ''*Bot2*''),
(''*MissionControl2*'', ''*Bot2*''),
(''*Watchdog*'', ''*Bot1*''), (''*Watchdog*'', ''*Bot2*'')] |)
value *wf-list-graph fabNet*

definition *sensorProps-try1* :: *string* ⇒ *security-level* **where**
sensorProps-try1 ≡ (λ *n*. *SINVAR-BLPbasic.default-node-properties*)(''*PresenceSensor*'' := *2*, ''*Webcam*'' := *3*)
value *BLP-offending-list fabNet sensorProps-try1*
value *sinvar fabNet sensorProps-try1*

definition *sensorProps-try2* :: *string* ⇒ *security-level* **where**
sensorProps-try2 ≡ (λ *n*. *SINVAR-BLPbasic.default-node-properties*)(''*PresenceSensor*'' := *2*, ''*Webcam*'' := *3*,
''*SensorSink*'' := *3*)
value *BLP-offending-list fabNet sensorProps-try2*
value *sinvar fabNet sensorProps-try2*

definition *sensorProps-try3* :: *string* ⇒ *security-level* **where**
sensorProps-try3 ≡ (λ *n*. *SINVAR-BLPbasic.default-node-properties*)(''*PresenceSensor*'' := *2*, ''*Webcam*'' := *3*,
''*SensorSink*'' := *3*, ''*Statistics*'' := *3*)
value *BLP-offending-list fabNet sensorProps-try3*
value *sinvar fabNet sensorProps-try3*

Another parameter set for confidential controlling information

definition *sensorProps-conf* :: *string* ⇒ *security-level* **where**
sensorProps-conf ≡ (λ *n*. *SINVAR-BLPbasic.default-node-properties*)(''*MissionControl1*'' := *1*, ''*MissionControl2*'' := *2*,
''*Bot1*'' := *1*, ''*Bot2*'' := *2*)
value *BLP-offending-list fabNet sensorProps-conf*
value *sinvar fabNet sensorProps-conf*

Complete example:

definition *sensorProps-NMParams-try3* :: (*string*, *nat*) *TopoS-Params* **where**
sensorProps-NMParams-try3 ≡ (| *node-properties* = [''*PresenceSensor*'' ↦ *2*,
''*Webcam*'' ↦ *3*,
''*SensorSink*'' ↦ *3*,
''*Statistics*'' ↦ *3*] |)
value *BLP-eval fabNet sensorProps-NMParams-try3*

export-code *SINVAR-LIB-BLPbasic* **checking** *Scala*

hide-const (**open**) *NetModel-node-props BLP-offending-list BLP-eval*

hide-const (**open**) *sinvar*

end
theory *SINVAR-Subnets*
imports../ *TopoS-Helper*
begin

6.1 SecurityInvariant Subnets

If unsure, maybe you should look at `SINVAR_SubnetsInGW.thy`

datatype *subnets* = *Subnet nat* | *BorderRouter nat* | *Unassigned*

definition *default-node-properties* :: *subnets*
  where *default-node-properties* ≡ *Unassigned*

fun *allowed-subnet-flow* :: *subnets* ⇒ *subnets* ⇒ *bool* **where**
  allowed-subnet-flow (*Subnet s1*) (*Subnet s2*) = (*s1* = *s2*) |
  allowed-subnet-flow (*Subnet s1*) (*BorderRouter s2*) = (*s1* = *s2*) |
  allowed-subnet-flow (*Subnet s1*) *Unassigned* = *True* |
  allowed-subnet-flow (*BorderRouter s1*) (*Subnet s2*) = *False* |
  allowed-subnet-flow (*BorderRouter s1*) *Unassigned* = *True* |
  allowed-subnet-flow (*BorderRouter s1*) (*BorderRouter s2*) = *True* |
  allowed-subnet-flow Unassigned Unassigned = *True* |
  allowed-subnet-flow Unassigned - = *False*

fun *sinvar* :: ′*v graph* ⇒ (′*v* ⇒ *subnets*) ⇒ *bool* **where**
  sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. *allowed-subnet-flow* (*nP e1*) (*nP e2*))

definition *receiver-violation* :: *bool* **where** *receiver-violation* = *False*

6.1.1 Preliminaries

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
  ⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
  ⟨*proof*⟩

6.1.2 ENF

lemma *Unassigned-only-to-Unassigned*: *allowed-subnet-flow Unassigned e2* ⟷ *e2* = *Unassigned*
  ⟨*proof*⟩
lemma *All-to-Unassigned*: ∀ *e1*. *allowed-subnet-flow e1 Unassigned*
  ⟨*proof*⟩
lemma *Unassigned-default-candidate*: ∀ *nP e1 e2*. ¬ *allowed-subnet-flow* (*nP e1*) (*nP e2*) ⟶ ¬ *allowed-subnet-flow Unassigned* (*nP e2*)
  ⟨*proof*⟩
lemma *allowed-subnet-flow-refl*: ∀ *e*. *allowed-subnet-flow e e*
  ⟨*proof*⟩
lemma *Subnets-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar allowed-subnet-flow*
  ⟨*proof*⟩
lemma *Subnets-ENF-refl*: *SecurityInvariant-withOffendingFlows.ENF-refl sinvar allowed-subnet-flow*

⟨*proof*⟩

definition *Subnets-offending-set*:: *'v graph* ⇒ (*'v* ⇒ *subnets*) ⇒ (*'v* × *'v*) *set set* **where**
Subnets-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ ¬ *allowed-subnet-flow* (*nP e1*) (*nP e2*)} })
lemma *Subnets-offending-set*:
SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = *Subnets-offending-set*
⟨*proof*⟩

interpretation *Subnets*: *SecurityInvariant-ACS*
where *default-node-properties* = *SINVAR-Subnets.default-node-properties*
and *sinvar* = *SINVAR-Subnets.sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *Subnets-offending-set*
⟨*proof*⟩

lemma *TopoS-Subnets*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

6.1.3 Analysis

lemma *violating-configurations*: ¬ *sinvar G nP* ⟹
∃ (*e1*, *e2*) ∈ *edges G*. *nP e1* = *Unassigned* ∨ (∃ *s1*. *nP e1* = *Subnet s1*) ∨ (∃ *s1*. *nP e1* = *BorderRouter s1*)
⟨*proof*⟩

All cases where the model can become invalid:

theorem *violating-configurations-exhaust*: ¬ *sinvar G nP* ⟷
(∃ (*e1*, *e2*) ∈ (*edges G*).
nP e1 = *Unassigned* ∧ *nP e2* ≠ *Unassigned* ∨
(∃ *s1 s2*. *nP e1* = *Subnet s1* ∧ *s1* ≠ *s2* ∧ (*nP e2* = *Subnet s2* ∨ *nP e2* = *BorderRouter s2*)) ∨
(∃ *s1 s2*. *nP e1* = *BorderRouter s1* ∧ *nP e2* = *Subnet s2*)
) (**is** *?l* ⟷ *?r*)
⟨*proof*⟩

hide-fact (**open**) *sinvar-mono*
hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-Subnets-impl*
imports *SINVAR-Subnets* ../*TopoS-Interface-impl*
begin

6.1.4 SecurityInvariant Subnets List Implementation

code-identifier code-module *SINVAR-Subnets-impl* => (*Scala*) *SINVAR-Subnets*

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *subnets*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *set* (*edgesL G*). *allowed-subnet-flow* (*nP e1*) (*nP e2*))

definition *Subnets-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *subnets*) ⇒ (*'v* × *'v*) *list list* **where**
Subnets-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G*. *case e of* (*e1*,*e2*) ⇒ ¬ *allowed-subnet-flow* (*nP e1*) (*nP e2*)]])

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-Subnets.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-Subnets.default-node-properties P* = *NetModel-node-props P*
⟨*proof*⟩

definition *Subnets-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-Subnets.default-node-properties P*))

interpretation *Subnets-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-Subnets.default-node-properties*
and *sinvar-spec*=*SINVAR-Subnets.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-Subnets.receiver-violation*
and *offending-flows-impl*=*Subnets-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*Subnets-eval*
⟨*proof*⟩

6.1.5 Subnets packing

definition *SINVAR-LIB-Subnets* :: (*'v*::*vertex*, *SINVAR-Subnets.subnets*) *TopoS-packed* **where**
SINVAR-LIB-Subnets ≡
⦇ *nm-name* = *''Subnets''*,
nm-receiver-violation = *SINVAR-Subnets.receiver-violation*,
nm-default = *SINVAR-Subnets.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *Subnets-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *Subnets-eval*
⦈
interpretation *SINVAR-LIB-Subnets-interpretation*: *TopoS-modelLibrary SINVAR-LIB-Subnets SINVAR-Subnets.sinvar*
⟨*proof*⟩

Examples

definition *example-net-sub* :: *nat list-graph* **where**
example-net-sub ≡ ⦇ *nodesL* = [*1*::*nat*,*2*,*3*,*4*, *8*,*9*, *11*,*12*, *42*],
edgesL = [(*1*,*2*),(*1*,*3*),(*1*,*4*),(*2*,*1*),(*2*,*3*),(*2*,*4*),(*3*,*1*),(*3*,*2*),(*3*,*4*),(*4*,*1*),(*4*,*2*),(*4*,*3*),
(*4*,*11*),(*1*,*11*),
(*8*,*9*),(*9*,*8*),
(*8*,*12*),
(*11*,*12*),

(11,42), (12,42), (3,42)] ⦈
value *wf-list-graph example-net-sub*

definition *example-conf-sub* **where**
example-conf-sub ≡ ((λ*e*. *SINVAR-Subnets.default-node-properties*)
(*1* := *Subnet 1*, *2*:= *Subnet 1*, *3*:= *Subnet 1*, *4*:=*Subnet 1*,
11:=*BorderRouter 1*,
8:=*Subnet 2*, *9*:=*Subnet 2*,
12:=*BorderRouter 2*,
42 := *Unassigned*))

value *sinvar example-net-sub example-conf-sub*

definition *example-net-sub-invalid* **where**
example-net-sub-invalid ≡ *example-net-sub*⦇*edgesL* := (*42*,*4*)#(*3*,*8*)#(*11*,*8*)#(*edgesL example-net-sub*)⦈

value *sinvar example-net-sub-invalid example-conf-sub*
value *Subnets-offending-list example-net-sub-invalid example-conf-sub*

value *sinvar*
⦇ *nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] ⦈
(λ*e*. *SINVAR-Subnets.default-node-properties*)
value *sinvar*
⦇ *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*11*,*12*], *edgesL* = [(*1*,*2*),(*2*,*3*),(*3*,*4*), (*4*,*11*),(*1*,*11*), (*8*,*9*),(*9*,*8*),(*8*,*12*), (*11*,*12*)] ⦈
((λ*e*. *SINVAR-Subnets.default-node-properties*)(*1* := *Subnet 1*, *2*:= *Subnet 1*, *3*:= *Subnet 1*, *4*:=*Subnet 1*, *11*:=*BorderRouter 1*,
8:=*Subnet 2*, *9*:=*Subnet 2*, *12*:=*BorderRouter 2*))
value *sinvar*
⦇ *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*11*,*12*], *edgesL* = [(*1*,*2*),(*2*,*3*),(*3*,*4*), (*4*,*11*),(*1*,*11*), (*8*,*9*),(*9*,*8*),(*8*,*12*), (*11*,*12*)] ⦈
((λ*e*. *SINVAR-Subnets.default-node-properties*)(*1* := *Subnet 1*, *2*:= *Subnet 1*, *3*:= *Subnet 1*, *4*:=*Subnet 1*, *11*:=*BorderRouter 1*))
value *sinvar*
⦇ *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] ⦈
((λ*e*. *SINVAR-Subnets.default-node-properties*)(*8*:=*Subnet 8*, *9*:=*Subnet 8*))

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-DomainHierarchyNG*
imports ../*TopoS-Helper*
HOL−Lattice.CompleteLattice
begin

6.2 SecurityInvariant DomainHierarchyNG

6.2.1 Datatype Domain Hierarchy

A fully qualified domain name for an entity in a tree-like hierarchy

datatype *domainNameDept* = *Dept string domainNameDept* (**infixr** ‹−−› *65*) |
Leaf — leaf of the tree, end of all domainNames

Example: the CoffeeMachine of I8

value *''i8''−−''CoffeeMachine''−−Leaf*

A tree strucuture to represent the general hierarchy, i.e. possible domainNameDepts

datatype *domainTree* = *Department*
string — division
domainTree list — sub divisions

one step in tree to find matching department

fun *hierarchy-next* :: *domainTree list* ⇒ *domainNameDept* ⇒ *domainTree option* **where**
hierarchy-next [] - = *None* |
hierarchy-next (*s#ss*) *Leaf* = *None* |
hierarchy-next ((*Department d ds*)#*ss*) (*Dept n ns*) = (*if d*=*n then Some* (*Department d ds*) *else hierarchy-next ss* (*Dept n ns*))

Examples:

lemma *hierarchy-next* [*Department ''i20''* [], *Department ''i8''* [*Department ''CoffeeMachine''* [], *Department ''TeaMachine''* []]]
(*''i8''−−Leaf*)
=
Some (*Department ''i8''* [*Department ''CoffeeMachine''* [], *Department ''TeaMachine''* []]) ⟨*proof*⟩
lemma *hierarchy-next* [*Department ''i20''* [], *Department ''i8''* [*Department ''CoffeeMachine''* [], *Department ''TeaMachine''* []]]
(*''i8''−−''whatsoever''−−Leaf*)
=
Some (*Department ''i8''* [*Department ''CoffeeMachine''* [], *Department ''TeaMachine''* []]) ⟨*proof*⟩
lemma *hierarchy-next* [*Department ''i20''* [], *Department ''i8''* [*Department ''CoffeeMachine''* [], *Department ''TeaMachine''* []]]
Leaf
= *None* ⟨*proof*⟩
lemma *hierarchy-next* [*Department ''i20''* [], *Department ''i8''* [*Department ''CoffeeMachine''* [], *Department ''TeaMachine''* []]]
(*''i0''−−Leaf*)
= *None* ⟨*proof*⟩

Does a given *domainNameDept* match the specified tree structure?

fun *valid-hierarchy-pos* :: *domainTree* ⇒ *domainNameDept* ⇒ *bool* **where**
valid-hierarchy-pos (*Department d ds*) *Leaf* = *True* |
valid-hierarchy-pos (*Department d ds*) (*Dept n Leaf*) = (*d*=*n*) |
valid-hierarchy-pos (*Department d ds*) (*Dept n ns*) = (*n*=*d* ∧
(*case hierarchy-next ds ns of*
None ⇒ *False* |
Some t ⇒ *valid-hierarchy-pos t ns*))

Examples:

lemma *valid-hierarchy-pos* (*Department* ''*TUM*'' []) *Leaf* ⟨*proof*⟩
lemma *valid-hierarchy-pos* (*Department* ''*TUM*'' []) *Leaf* ⟨*proof*⟩
lemma *valid-hierarchy-pos* (*Department* ''*TUM*'' []) (''*TUM*''−−*Leaf*) ⟨*proof*⟩
lemma *valid-hierarchy-pos* (*Department* ''*TUM*'' []) (''*TUM*''−−''*facilityManagement*''−−*Leaf*) = *False* ⟨*proof*⟩
lemma *valid-hierarchy-pos* (*Department* ''*TUM*'' []) (''*LMU*''−−*Leaf*) = *False* ⟨*proof*⟩
lemma *valid-hierarchy-pos* (*Department* ''*TUM*'' [*Department* ''*i8*'' [], (*Department* ''*i20*'' [])]) (''*TUM*''−−*Leaf*) ⟨*proof*⟩
lemma *valid-hierarchy-pos* (*Department* ''*TUM*'' [*Department* ''*i8*'' [], *Department* ''*i20*'' []]) (''*TUM*''−−''*i8*''−−*Leaf*) ⟨*proof*⟩

```
lemma valid-hierarchy-pos
  (Department ''TUM'' [
    Department ''i8'' [
     Department ''CoffeeMachine'' [],
     Department ''TeaMachine'' []
    ],
    Department ''i20'' []
  ])
  (''TUM''−−''i8''−−''CoffeeMachine''−−Leaf) ⟨proof⟩
```

lemma *valid-hierarchy-pos* (*Department* ''*TUM*'' [*Department* ''*i8*'' [*Department* ''*CoffeeMachine*'' [], *Department* ''*TeaMachine*'' []], *Department* ''*i20*'' []])
(''*TUM*''−−''*i8*''−−''*CleanKitchen*''−−*Leaf*) = *False* ⟨*proof*⟩

```
instantiation domainNameDept :: order
begin
  print-context

  fun less-eq-domainNameDept :: domainNameDept ⇒ domainNameDept ⇒ bool where
    Leaf ≤ (Dept - -) = False |
    (Dept - -) ≤ Leaf = True |
    Leaf ≤ Leaf = True |
    (Dept n1 n1s) ≤ (Dept n2 n2s) = (n1=n2 ∧ n1s ≤ n2s)

  fun less-domainNameDept :: domainNameDept ⇒ domainNameDept ⇒ bool where
    Leaf < Leaf = False |
    Leaf < (Dept - -) = False |
    (Dept - -) < Leaf = True |
    (Dept n1 n1s) < (Dept n2 n2s) = (n1=n2 ∧ n1s < n2s)

  lemma Leaf-Top: a ≤ Leaf
    ⟨proof⟩

  lemma Leaf-Top-Unique: Leaf ≤ a = (a = Leaf)
    ⟨proof⟩

  lemma no-Bot: n1 ≠ n2 ⟹ z ≤ n1 −− n1s ⟹ z ≤ n2 −− n2s ⟹ False
    ⟨proof⟩

  lemma uncomparable-sup-is-Top: n1 ≠ n2 ⟹ n1 −− x ≤ z ⟹ n2 −− y ≤ z ⟹ z = Leaf
    ⟨proof⟩
```

lemma *common-inf-imp-comparable*: $(z::domainNameDept) \leq a \Longrightarrow z \leq b \Longrightarrow a \leq b \vee b \leq a$
⟨*proof*⟩

lemma *prepend-domain*: $a \leq b \Longrightarrow x{-}{-}a \leq x{-}{-}b$
⟨*proof*⟩
lemma *unfold-dmain-leq*: $y \leq zn\ {-}{-}\ zns \Longrightarrow \exists\, yns.\ y = zn\ {-}{-}\ yns \wedge yns \leq zns$
⟨*proof*⟩

lemma *less-eq-refl*:
fixes x :: *domainNameDept*
shows $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
⟨*proof*⟩

instance
⟨*proof*⟩
end

instantiation *domainNameDept* :: *Orderings.top*
begin
definition *top-domainNameDept* **where** *Orderings.top* ≡ *Leaf*
instance
⟨*proof*⟩
end

lemma (''*TUM*''−−''*BLUBB*''−−*Leaf*) ≤ (''*TUM*''−−*Leaf*) ⟨*proof*⟩

lemma (''*TUM*''−−''*i8*''−−*Leaf*) ≤ (''*TUM*''−−*Leaf*) ⟨*proof*⟩
lemma ¬ (''*TUM*''−−*Leaf*) ≤ (''*TUM*''−−''*i8*''−−*Leaf*) ⟨*proof*⟩
lemma *valid-hierarchy-pos* (*Department* ''*TUM*'' [*Department* ''*i8*'' [], *Department* ''*i20*'' []]) (''*TUM*''−−''*i8*''−−*Leaf*) ⟨*proof*⟩

lemma (''*TUM*''−−*Leaf*) ≤ *Leaf* ⟨*proof*⟩
lemma *valid-hierarchy-pos* (*Department* ''*TUM*'' [*Department* ''*i8*'' [], *Department* ''*i20*'' []]) (*Leaf*) ⟨*proof*⟩

lemma ¬ *Leaf* ≤ (''*TUM*''−−*Leaf*) ⟨*proof*⟩
lemma *valid-hierarchy-pos* (*Department* ''*TUM*'' [*Department* ''*i8*'' [], *Department* ''*i20*'' []]) (''*TUM*''−−*Leaf*) ⟨*proof*⟩

lemma ¬ (''*TUM*''−−''*BLUBB*''−−*Leaf*) ≤ (''*X*''−−''*TUM*''−−''*BLUBB*''−−*Leaf*) ⟨*proof*⟩

lemma (''*TUM*''−−''*i8*''−−''*CoffeeMachine*''−−*Leaf*) ≤ (''*TUM*''−−''*i8*''−−*Leaf*) ⟨*proof*⟩
lemma (''*TUM*''−−''*i8*''−−*Leaf*) ≤ (''*TUM*''−−''*i8*''−−*Leaf*) ⟨*proof*⟩
lemma (''*TUM*''−−''*i8*''−−''*CoffeeMachine*''−−*Leaf*) ≤ (''*TUM*''−−*Leaf*) ⟨*proof*⟩
lemma (''*TUM*''−−''*i8*''−−''*CoffeeMachine*''−−*Leaf*) ≤ (*Leaf*) ⟨*proof*⟩
lemma ¬ (''*TUM*''−−''*i8*''−−*Leaf*) ≤ (''*TUM*''−−''*i20*''−−*Leaf*) ⟨*proof*⟩
lemma ¬ (''*TUM*''−−''*i20*''−−*Leaf*) ≤ (''*TUM*''−−''*i8*''−−*Leaf*) ⟨*proof*⟩

6.2.2 Adding Chop

by putting entities higher in the hierarchy.

fun *domainNameDeptChopOne* :: *domainNameDept* ⇒ *domainNameDept* **where**
domainNameDeptChopOne Leaf = *Leaf* |
domainNameDeptChopOne (*name*−−*Leaf*) = *Leaf* |

domainNameDeptChopOne (*name*−−*dpt*) = *name*−−(*domainNameDeptChopOne dpt*)

lemma *domainNameDeptChopOne* (''*i8*''−−''*CoffeeMachine*''−−*Leaf*) = ''*i8*'' −− *Leaf* ⟨*proof*⟩
lemma *domainNameDeptChopOne* (''*i8*''−−''*CoffeeMachine*''−−''*CoffeeSlave*''−−*Leaf*) = ''*i8*'' −− ''*CoffeeMachine*'' −− *Leaf* ⟨*proof*⟩
lemma *domainNameDeptChopOne Leaf* = *Leaf* ⟨*proof*⟩

theorem *chopOne-not-decrease*: *dn* ≤ *domainNameDeptChopOne dn*
⟨*proof*⟩

lemma *chopOneContinue*: *dpt* ≠ *Leaf* ⟹ *domainNameDeptChopOne* (*name* −− *dpt*) = *name* −− *domainNameDeptChopOne* (*dpt*)
⟨*proof*⟩

fun *domainNameChop* :: *domainNameDept* ⇒ *nat* ⇒ *domainNameDept* **where**
domainNameChop Leaf - = *Leaf* |
domainNameChop namedpt 0 = *namedpt* |
domainNameChop namedpt (*Suc n*) = *domainNameChop* (*domainNameDeptChopOne namedpt*) *n*

lemma *domainNameChop* (''*i8*''−−''*CoffeeMachine*''−−*Leaf*) *2* = *Leaf* ⟨*proof*⟩
lemma *domainNameChop* (''*i8*''−−''*CoffeeMachine*''−−''*CoffeeSlave*''−−*Leaf*) *2* = ''*i8*''−−*Leaf* ⟨*proof*⟩
lemma *domainNameChop* (''*i8*''−−*Leaf*) *0* = ''*i8*''−−*Leaf* ⟨*proof*⟩
lemma *domainNameChop* (*Leaf*) *8* = *Leaf* ⟨*proof*⟩

lemma *chop0*[*simp*]: *domainNameChop dn 0* = *dn*
⟨*proof*⟩

lemma (*domainNameDeptChopOne*^^*2*) (''*d1*''−−''*d2*''−−''*d3*''−−*Leaf*) = ''*d1*''−−*Leaf* ⟨*proof*⟩

domainNameChop is equal to applying n times chop one

lemma *domainNameChopFunApply*: *domainNameChop dn n* = (*domainNameDeptChopOne*^^*n*) *dn*
⟨*proof*⟩

lemma *domainNameChopRotateSuc*: *domainNameChop dn* (*Suc n*) = *domainNameDeptChopOne* (*domainNameChop dn n*)
⟨*proof*⟩

lemma *domainNameChopRotate*: *domainNameChop* (*domainNameDeptChopOne dn*) *n* = *domainNameDeptChopOne* (*domainNameChop dn n*)
⟨*proof*⟩

theorem *chop-not-decrease-hierarchy*: *dn* ≤ *domainNameChop dn n*
⟨*proof*⟩

corollary *dn* ≤ *domainNameDeptChopOne* ((*domainNameDeptChopOne* ^^ *n*) (*dn*))
⟨*proof*⟩

compute maximum common level of both inputs

fun *chop-sup* :: *domainNameDept* ⇒ *domainNameDept* ⇒ *domainNameDept* **where**
chop-sup Leaf - = Leaf |
chop-sup - Leaf = Leaf |
chop-sup (*a*−−*as*) (*b*−−*bs*) = (*if a* ≠ *b then Leaf else a*−−(*chop-sup as bs*))

lemma *chop-sup* (′′*a*′′−−′′*b*′′−−′′*c*′′−−*Leaf*) (′′*a*′′−−′′*b*′′−−′′*d*′′−−*Leaf*) = ′′*a*′′ −− ′′*b*′′ −− *Leaf* ⟨*proof*⟩
lemma *chop-sup* (′′*a*′′−−′′*b*′′−−′′*c*′′−−*Leaf*) (′′*a*′′−−′′*x*′′−−′′*d*′′−−*Leaf*) = ′′*a*′′ −− *Leaf* ⟨*proof*⟩
lemma *chop-sup* (′′*a*′′−−′′*b*′′−−′′*c*′′−−*Leaf*) (′′*x*′′−−′′*x*′′−−′′*d*′′−−*Leaf*) = *Leaf* ⟨*proof*⟩

lemma *chop-sup-commute*: *chop-sup a b* = *chop-sup b a*
⟨*proof*⟩
lemma *chop-sup-max1*: *a* ≤ *chop-sup a b*
⟨*proof*⟩
lemma *chop-sup-max2*: *b* ≤ *chop-sup a b*
⟨*proof*⟩

lemma *chop-sup-is-sup*: ∀ *z*. *a* ≤ *z* ∧ *b* ≤ *z* ⟶ *chop-sup a b* ≤ *z*
⟨*proof*⟩

datatype *domainName* = *DN domainNameDept* | *Unassigned*

6.2.3 Makeing it a complete Lattice

instantiation *domainName* :: *partial-order*
begin

fun *leq-domainName* :: *domainName* ⇒ *domainName* ⇒ *bool* **where**
leq-domainName Unassigned - = True |
leq-domainName - Unassigned = False |
leq-domainName (*DN dnA*) (*DN dnB*) = (*dnA* ≤ *dnB*)
instance
⟨*proof*⟩
end

lemma *is-Inf* {*Unassigned*, *DN Leaf*} *Unassigned*
⟨*proof*⟩

The infinum of two elements:

fun *DN-inf* :: *domainName* ⇒ *domainName* ⇒ *domainName* **where**
DN-inf Unassigned - = Unassigned |
DN-inf - Unassigned = Unassigned |
DN-inf (*DN a*) (*DN b*) = (*if a* ≤ *b then DN a else if b* ≤ *a then DN b else Unassigned*)

lemma *DN-inf* (*DN* (′′*TUM*′′−−′′*i8*′′−−*Leaf*)) (*DN* (′′*TUM*′′−−′′*i20*′′−−*Leaf*)) = *Unassigned* ⟨*proof*⟩
lemma *DN-inf* (*DN* (′′*TUM*′′−−′′*i8*′′−−*Leaf*)) (*DN* (′′*TUM*′′−−*Leaf*)) = *DN* (′′*TUM*′′ −− ′′*i8*′′ −− *Leaf*) ⟨*proof*⟩

lemma *DN-inf-commute*: *DN-inf x y* = *DN-inf y x*
⟨*proof*⟩

lemma *DN-inf-is-inf*: *is-inf x y* (*DN-inf x y*)
⟨*proof*⟩

fun *DN-sup* :: *domainName* ⇒ *domainName* ⇒ *domainName* **where**
DN-sup Unassigned a = *a* |
DN-sup a Unassigned = *a* |
DN-sup (*DN a*) (*DN b*) = *DN* (*chop-sup a b*)

lemma *DN-sup-commute*: *DN-sup x y* = *DN-sup y x*
⟨*proof*⟩

lemma *DN-sup-is-sup*: *is-sup x y* (*DN-sup x y*)
⟨*proof*⟩

domainName is a Lattice:

instantiation *domainName* :: *lattice*
begin
instance
⟨*proof*⟩
end

datatype *domainNameTrust* = *DN* (*domainNameDept* × *nat*) | *Unassigned*

fun *leq-domainNameTrust* :: *domainNameTrust* ⇒ *domainNameTrust* ⇒ *bool* (**infixr** ‹$\sqsubseteq_{trust}$› *65*)
where
leq-domainNameTrust Unassigned - = *True* |
leq-domainNameTrust - *Unassigned* = *False* |
leq-domainNameTrust (*DN* (*dnA*, *trustA*)) (*DN* (*dnB*, *trustB*)) = (*dnA* ≤ (*domainNameChop dnB trustB*))

lemma *leq-domainNameTrust-refl*: $x \sqsubseteq_{trust} x$
⟨*proof*⟩

lemma *leq-domainNameTrust-NOT-trans*: $\exists x\ y\ z.\ x \sqsubseteq_{trust} y \wedge y \sqsubseteq_{trust} z \wedge \neg\ x \sqsubseteq_{trust} z$
⟨*proof*⟩

lemma *leq-domainNameTrust-NOT-antisym*: $\exists x\ y.\ x \sqsubseteq_{trust} y \wedge y \sqsubseteq_{trust} x \wedge x \neq y$
⟨*proof*⟩

6.2.4 The network security invariant

definition *default-node-properties* :: *domainNameTrust*
where *default-node-properties* = *Unassigned*

The sender is, noticing its trust level, on the same or higher hierarchy level as the receiver.

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *domainNameTrust*) ⇒ *bool* **where**
sinvar G nP = (∀ (*s*, *r*) ∈ *edges G*. (*nP r*) $\sqsubseteq_{trust}$ (*nP s*))

a domain name must be in the supplied tree

fun *verify-globals* :: *'v graph* ⇒ (*'v* ⇒ *domainNameTrust*) ⇒ *domainTree* ⇒ *bool* **where**
verify-globals G nP tree = (∀ *v* ∈ *nodes G*.
case (*nP v*) *of Unassigned* ⇒ *True* | *DN* (*level*, *trust*) ⇒ *valid-hierarchy-pos tree level*
)

lemma *verify-globals* (| *nodes=set* [*1*,*2*,*3*], *edges=set* [] |) (λ*n*. *default-node-properties*) (*Department* *''TUM''* [])
⟨*proof*⟩

definition *receiver-violation* :: *bool* **where** *receiver-violation* = *False*

thm *SecurityInvariant-withOffendingFlows.sinvar-mono-def*
lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
⟨*proof*⟩

6.2.5 ENF

lemma *DomainHierarchyNG-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar* (λ *s r*. *r* $\sqsubseteq_{trust}$ *s*)
⟨*proof*⟩
lemma *DomainHierarchyNG-ENF-refl*: *SecurityInvariant-withOffendingFlows.ENF-refl sinvar* (λ *s r*. *r* $\sqsubseteq_{trust}$ *s*)
⟨*proof*⟩
lemma *unassigned-default-candidate*: ∀ *nP s r*. ¬ (*nP r*) $\sqsubseteq_{trust}$ (*nP s*) ⟶ ¬ (*nP r*) $\sqsubseteq_{trust}$ *default-node-properties*
⟨*proof*⟩

definition *DomainHierarchyNG-offending-set*:: *'v graph* ⇒ (*'v* ⇒ *domainNameTrust*) ⇒ (*'v* × *'v*) *set set* **where**
DomainHierarchyNG-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ ¬ (*nP e2*) $\sqsubseteq_{trust}$ (*nP e1*)} })
lemma *DomainHierarchyNG-offending-set*: *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *DomainHierarchyNG-offending-set*
⟨*proof*⟩

lemma *Unassigned-unique-default*: *otherbot* ≠ *default-node-properties* ⟹

$\exists$ *G nP gP i f.*
wf-graph G $\wedge$
$\neg$ *sinvar G nP* $\wedge$
f $\in$ *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G nP* $\wedge$
sinvar (*delete-edges G f*) *nP* $\wedge$
(*i* $\in$ *fst* ' *f* $\wedge$ *sinvar G* (*nP*(*i* := *otherbot*)))
⟨*proof*⟩

interpretation *DomainHierarchyNG*: *SecurityInvariant-ACS*
where *default-node-properties* = *default-node-properties*
and *sinvar* = *sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *DomainHierarchyNG-offending-set*
⟨*proof*⟩

lemma *TopoS-DomainHierarchyNG*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

hide-const (**open**) *sinvar receiver-violation*

end
theory *SINVAR-DomainHierarchyNG-impl*
imports *SINVAR-DomainHierarchyNG ../TopoS-Interface-impl*
begin

6.2.6 SecurityInvariant DomainHierarchy List Implementation

code-identifier code-module *SINVAR-DomainHierarchyNG-impl* => (*Scala*) *SINVAR-DomainHierarchyNG*

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *domainNameTrust*) ⇒ *bool* **where**
sinvar G nP = ($\forall$ (*s*, *r*) $\in$ *set* (*edgesL G*). (*nP r*) $\sqsubseteq_{trust}$ (*nP s*))

definition *DomainHierarchyNG-sanity-check-config* :: *domainNameTrust list* ⇒ *domainTree* ⇒ *bool*
where
DomainHierarchyNG-sanity-check-config host-attributes tree = ($\forall$ *c* $\in$ *set host-attributes.*
case c of Unassigned ⇒ *True*
| *DN* (*level*, *trust*) ⇒ *valid-hierarchy-pos tree level*
)

fun *verify-globals* :: *'v list-graph* ⇒ (*'v* ⇒ *domainNameTrust*) ⇒ *domainTree* ⇒ *bool* **where**
verify-globals G nP tree = ($\forall$ *v* $\in$ *set* (*nodesL G*).
case (*nP v*) *of Unassigned* ⇒ *True* | *DN* (*level*, *trust*) ⇒ *valid-hierarchy-pos tree level*
)

lemma *DomainHierarchyNG-sanity-check-config c tree* ⟹
{*x*. $\exists$ *v*. *nP v* = *x*} = *set c* ⟹
verify-globals G nP tree

⟨*proof*⟩

definition *DomainHierarchyNG-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *domainNameTrust*) ⇒ (*'v* × *'v*) *list list* **where**
DomainHierarchyNG-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G*. *case e of* (*s*,*r*) ⇒ ¬ (*nP r*) $\sqsubseteq_{trust}$ (*nP s*)]])

lemma *DomainHierarchyNG.node-props P* =
(λ*i*. *case node-properties P i of None* ⇒ *SINVAR-DomainHierarchyNG.default-node-properties* | *Some property* ⇒ *property*)
⟨*proof*⟩

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-DomainHierarchyNG.default-node-properties*))

lemma[*code-unfold*]: *DomainHierarchyNG.node-props P* = *NetModel-node-props P*
⟨*proof*⟩

definition *DomainHierarchyNG-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-DomainHierarchyNG.default-node-properties P*))

interpretation *DomainHierarchyNG-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-DomainHierarchyNG.default-node-properties*
and *sinvar-spec*=*SINVAR-DomainHierarchyNG.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-DomainHierarchyNG.receiver-violation*
and *offending-flows-impl*=*DomainHierarchyNG-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*DomainHierarchyNG-eval*
⟨*proof*⟩

6.2.7 DomainHierarchyNG packing

definition *SINVAR-LIB-DomainHierarchyNG* :: (*'v*::*vertex*, *domainNameTrust*) *TopoS-packed* **where**
SINVAR-LIB-DomainHierarchyNG ≡
⦇ *nm-name* = *''DomainHierarchyNG''*,
nm-receiver-violation = *SINVAR-DomainHierarchyNG.receiver-violation*,
nm-default = *SINVAR-DomainHierarchyNG.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *DomainHierarchyNG-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *DomainHierarchyNG-eval*
⦈
interpretation *SINVAR-LIB-DomainHierarchyNG-interpretation*: *TopoS-modelLibrary SINVAR-LIB-DomainHierarc*

SINVAR-DomainHierarchyNG.sinvar

⟨*proof*⟩

Examples:

definition *example-TUM-net* :: *string list-graph* **where**
example-TUM-net ≡ (| *nodesL*=[''*Gateway*'', ''*LowerSVR*'', ''*UpperSRV*''],
edgesL=[
(''*Gateway*'',''*LowerSVR*''), (''*Gateway*'',''*UpperSRV*''),
(''*LowerSVR*'', ''*Gateway*''),
(''*UpperSRV*'', ''*Gateway*'')
] |)
value *wf-list-graph example-TUM-net*

definition *example-TUM-config* :: *string* ⇒ *domainNameTrust* **where**
example-TUM-config ≡ ((λ *e*. *default-node-properties*)
(''*Gateway*'':= *DN* (''*ACD*''−−''*AISD*''−−*Leaf*, *1*),
''*LowerSVR*'':= *DN* (''*ACD*''−−''*AISD*''−−*Leaf*, *0*),
''*UpperSRV*'':= *DN* (''*ACD*''−−*Leaf*, *0*)
))

definition *example-TUM-hierarchy* :: *domainTree* **where**
example-TUM-hierarchy ≡ (*Department* ''*ACD*'' [
Department ''*AISD*'' []
])

value *verify-globals example-TUM-net example-TUM-config example-TUM-hierarchy*
value *sinvar example-TUM-net example-TUM-config*

definition *example-TUM-net-invalid* **where**
example-TUM-net-invalid ≡ *example-TUM-net*(|*edgesL* :=
(''*LowerSRV*'', ''*UpperSRV*'')#(*edgesL example-TUM-net*)|)

value *verify-globals example-TUM-net-invalid example-TUM-config example-TUM-hierarchy*
value *sinvar example-TUM-net-invalid example-TUM-config*
value *DomainHierarchyNG-offending-list example-TUM-net-invalid example-TUM-config*

hide-const (**open**) *NetModel-node-props*

hide-const (**open**) *sinvar*

end
theory *SINVAR-BLPtrusted-impl*
imports *SINVAR-BLPtrusted ../TopoS-Interface-impl*
begin

6.2.8 SecurityInvariant List Implementation

code-identifier code-module *SINVAR-BLPtrusted-impl* => (*Scala*) *SINVAR-BLPtrusted*

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *SINVAR-BLPtrusted.node-config*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *set* (*edgesL G*). (*if trusted* (*nP e2*) *then True else security-level* (*nP e1*) ≤ *security-level* (*nP e2*)))

definition *BLP-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *SINVAR-BLPtrusted.node-config*) ⇒ (*'v* × *'v*) *list list* **where**
BLP-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G. case e of* (*e1*,*e2*) ⇒ ¬ *SINVAR-BLPtrusted.BLP-P* (*nP e1*) (*nP e2*)]])

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-BLPtrusted.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-BLPtrusted.default-node-properties P* = *NetModel-node-props P*
⟨*proof*⟩

definition *BLP-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-BLPtrusted.default-node-properties P*))

interpretation *BLPtrusted-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-BLPtrusted.default-node-properties*
and *sinvar-spec*=*SINVAR-BLPtrusted.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-BLPtrusted.receiver-violation*
and *offending-flows-impl*=*BLP-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*BLP-eval*
⟨*proof*⟩

6.2.9 BLPtrusted packing

definition *SINVAR-LIB-BLPtrusted* :: (*'v*::*vertex*, *SINVAR-BLPtrusted.node-config*) *TopoS-packed* **where**
SINVAR-LIB-BLPtrusted ≡
⦇ *nm-name* = *''BLPtrusted''*,
nm-receiver-violation = *SINVAR-BLPtrusted.receiver-violation*,
nm-default = *SINVAR-BLPtrusted.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *BLP-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *BLP-eval*
⦈
interpretation *SINVAR-LIB-BLPtrusted-interpretation*: *TopoS-modelLibrary SINVAR-LIB-BLPtrusted*

SINVAR-BLPtrusted.sinvar
⟨*proof*⟩

6.2.10 Example

export-code *SINVAR-LIB-BLPtrusted* **checking** *Scala*

hide-const (**open**) *NetModel-node-props BLP-offending-list BLP-eval*

hide-const (**open**) *sinvar*

end
theory *SINVAR-SecGwExt*
imports ../*TopoS-Helper*
begin

6.3 SecurityInvariant PolEnforcePointExtended

A PolEnforcePoint is an application-level central policy enforcement point. Legacy note: The old verions called it a SecurityGateway.

Hosts may belong to a certain domain. Sometimes, a pattern where intra-domain communication between domain members must be approved by a central instance is required.

We call such a central instance PolEnforcePoint and present a template for this architecture. Five host roles are distinguished:. A PolEnforcePoint, aPolEnforcePointIN which accessible from the outside, a DomainMember, a less-restricted AccessibleMember which is accessible from the outside world, and a default value Unassigned that reflects none of these roles.

datatype *secgw-member* = *PolEnforcePoint* | *PolEnforcePointIN* | *DomainMember* | *AccessibleMember* | *Unassigned*

definition *default-node-properties* :: *secgw-member*
  where *default-node-properties* ≡ *Unassigned*

fun *allowed-secgw-flow* :: *secgw-member* ⇒ *secgw-member* ⇒ *bool* **where**
  allowed-secgw-flow PolEnforcePoint - = True |
  allowed-secgw-flow PolEnforcePointIN - = True |
  allowed-secgw-flow DomainMember DomainMember = False |
  allowed-secgw-flow DomainMember - = True |
  allowed-secgw-flow AccessibleMember DomainMember = False |
  allowed-secgw-flow AccessibleMember - = True |
  allowed-secgw-flow Unassigned Unassigned = True |
  allowed-secgw-flow Unassigned PolEnforcePointIN = True |
  allowed-secgw-flow Unassigned AccessibleMember = True |
  allowed-secgw-flow Unassigned - = False

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *secgw-member*) ⇒ *bool* **where**
  sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. *e1* ≠ *e2* ⟶ *allowed-secgw-flow* (*nP e1*) (*nP e2*))

definition *receiver-violation* :: *bool* **where** *receiver-violation* = *False*

6.3.1 Preliminaries

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
  ⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
  ⟨*proof*⟩

6.3.2 ENF

lemma *PolEnforcePoint-ENFnr*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-not-refl sinvar allowed-secgw-flow*
⟨*proof*⟩
lemma *Unassigned-botdefault*: ∀ *e1 e2*. *e2* ≠ *Unassigned* ⟶ ¬ *allowed-secgw-flow e1 e2* ⟶ ¬ *allowed-secgw-flow Unassigned e2*
⟨*proof*⟩
lemma *Unassigned-not-to-Member*: ¬ *allowed-secgw-flow Unassigned DomainMember*
⟨*proof*⟩
lemma *All-to-Unassigned*: ∀ *e1*. *allowed-secgw-flow e1 Unassigned*
⟨*proof*⟩

definition *PolEnforcePointExtended-offending-set*:: *'v graph* ⇒ (*'v* ⇒ *secgw-member*) ⇒ (*'v* × *'v*) *set set* **where**
PolEnforcePointExtended-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ *e1* ≠ *e2* ∧ ¬ *allowed-secgw-flow* (*nP e1*) (*nP e2*)} })
lemma *PolEnforcePointExtended-offending-set*: *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *PolEnforcePointExtended-offending-set*
⟨*proof*⟩

interpretation *PolEnforcePointExtended*: *SecurityInvariant-ACS*
where *default-node-properties* = *default-node-properties*
and *sinvar* = *sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *PolEnforcePointExtended-offending-set*
⟨*proof*⟩

lemma *TopoS-PolEnforcePointExtended*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

hide-const (**open**) *sinvar receiver-violation*

end
theory *SINVAR-SecGwExt-impl*
imports *SINVAR-SecGwExt* ../*TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-SecGwExt-impl* => (*Scala*) *SINVAR-SecGwExt*

6.3.3 SecurityInvariant PolEnforcePointExtended List Implementation

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *SINVAR-SecGwExt.secgw-member*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *set* (*edgesL G*). *e1* ≠ *e2* ⟶ *SINVAR-SecGwExt.allowed-secgw-flow* (*nP e1*) (*nP e2*))

definition *PolEnforcePointExtended-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *secgw-member*) ⇒ (*'v* × *'v*) *list list* **where**
PolEnforcePointExtended-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G*. *case e of* (*e1*,*e2*) ⇒ *e1* ≠ *e2* ∧ ¬ *allowed-secgw-flow* (*nP e1*) (*nP e2*)]])

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-SecGwExt.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-SecGwExt.default-node-properties P* = *NetModel-node-props P*
⟨*proof*⟩

definition *PolEnforcePoint-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-SecGwExt.default-node-properties P*))

interpretation *PolEnforcePoint-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-SecGwExt.default-node-properties*
and *sinvar-spec*=*SINVAR-SecGwExt.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-SecGwExt.receiver-violation*
and *offending-flows-impl*=*PolEnforcePointExtended-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*PolEnforcePoint-eval*
⟨*proof*⟩

6.3.4 PolEnforcePoint packing

definition *SINVAR-LIB-PolEnforcePointExtended* :: (*'v*::*vertex*, *secgw-member*) *TopoS-packed* **where**
SINVAR-LIB-PolEnforcePointExtended ≡
(| *nm-name* = *''PolEnforcePointExtended''*,
nm-receiver-violation = *SINVAR-SecGwExt.receiver-violation*,
nm-default = *SINVAR-SecGwExt.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *PolEnforcePointExtended-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *PolEnforcePoint-eval*
|)
interpretation *SINVAR-LIB-PolEnforcePointExtended-interpretation*: *TopoS-modelLibrary SINVAR-LIB-PolEnforce*

SINVAR-SecGwExt.sinvar
⟨*proof*⟩

Examples

definition *example-net-secgw* :: *nat list-graph* **where**
example-net-secgw ≡ (| *nodesL* = [*1*::*nat*,*2*, *3*, *8*,*9*, *11*,*12*],
edgesL = [(*3*,*8*),(*8*,*3*),(*2*,*8*),(*8*,*1*),(*1*,*9*),(*9*,*2*),(*2*,*9*),(*9*,*1*), (*1*,*3*), (*8*,*11*),(*8*,*12*), (*11*,*9*), (*11*,*3*), (*11*,*12*)] |)
value *wf-list-graph example-net-secgw*

definition *example-conf-secgw* **where**
example-conf-secgw ≡ ((λ*e*. *SINVAR-SecGwExt.default-node-properties*)
(*1* := *DomainMember*, *2*:= *DomainMember*, *3*:= *AccessibleMember*,
8:= *PolEnforcePoint*, *9*:= *PolEnforcePointIN*))

export-code *sinvar* **checking** *SML*
definition *test* = *sinvar* (| *nodesL*=[*1*::*nat*], *edgesL*=[] |) (λ-. *SINVAR-SecGwExt.default-node-properties*)

export-code *test* **checking** *SML*
value *sinvar* (| *nodesL*=[*1*::*nat*], *edgesL*=[] |) (λ-. *SINVAR-SecGwExt.default-node-properties*)

value *sinvar example-net-secgw example-conf-secgw*
value *PolEnforcePoint-offending-list example-net-secgw example-conf-secgw*

definition *example-net-secgw-invalid* **where**
example-net-secgw-invalid ≡ *example-net-secgw*(|*edgesL* := (*3*,*1*)#(*11*,*1*)#(*11*,*8*)#(*1*,*2*)#(*edgesL example-net-secgw*)|)

value *sinvar example-net-secgw-invalid example-conf-secgw*
value *PolEnforcePoint-offending-list example-net-secgw-invalid example-conf-secgw*

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-Sink*
imports ../*TopoS-Helper*
begin

6.4 SecurityInvariant Sink (IFS)

datatype *node-config* = *Sink* | *SinkPool* | *Unassigned*

definition *default-node-properties* :: *node-config*
where *default-node-properties* = *Unassigned*

fun *allowed-sink-flow* :: *node-config* ⇒ *node-config* ⇒ *bool* **where**
allowed-sink-flow Sink - = *False* |
allowed-sink-flow SinkPool SinkPool = *True* |
allowed-sink-flow SinkPool Sink = *True* |
allowed-sink-flow SinkPool - = *False* |
allowed-sink-flow Unassigned - = *True*

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *node-config*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. *e1* ≠ *e2* ⟶ *allowed-sink-flow* (*nP e1*) (*nP e2*))

definition *receiver-violation* :: *bool* **where** *receiver-violation* = *True*

6.4.1 Preliminaries

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
⟨*proof*⟩

6.4.2 ENF

lemma *Sink-ENFnr*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-not-refl sinvar allowed-sink-flow*
⟨*proof*⟩
lemma *Unassigned-to-All*: ∀ *e2*. *allowed-sink-flow Unassigned e2*
⟨*proof*⟩
lemma *Unassigned-default-candidate*: ∀ *e1 e2*. ¬ *allowed-sink-flow e1 e2* ⟶ ¬ *allowed-sink-flow e1 Unassigned*
⟨*proof*⟩

definition *Sink-offending-set*:: *'v graph* ⇒ (*'v* ⇒ *node-config*) ⇒ (*'v* × *'v*) *set set* **where**
Sink-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ *e1* ≠ *e2* ∧ ¬ *allowed-sink-flow* (*nP e1*) (*nP e2*)} })
lemma *Sink-offending-set*:
SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = *Sink-offending-set*
⟨*proof*⟩

interpretation *Sink*: *SecurityInvariant-IFS*
where *default-node-properties* = *default-node-properties*
and *sinvar* = *sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *Sink-offending-set*
⟨*proof*⟩

lemma *TopoS-Sink*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

hide-fact (**open**) *sinvar-mono*
hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-Sink-impl*
imports *SINVAR-Sink* ../*TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-Sink-impl* => (*Scala*) *SINVAR-Sink*

6.4.3 SecurityInvariant Sink (IFS) List Implementation

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *node-config*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *set* (*edgesL G*). *e1* ≠ *e2* ⟶ *SINVAR-Sink.allowed-sink-flow* (*nP e1*) (*nP e2*))

definition *Sink-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *SINVAR-Sink.node-config*) ⇒ (*'v* × *'v*) *list list* **where**
Sink-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G*. *case e of* (*e1*,*e2*) ⇒ *e1* ≠ *e2* ∧ ¬ *allowed-sink-flow* (*nP e1*) (*nP e2*)]])

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-Sink.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-Sink.default-node-properties P* = *NetModel-node-props P*
⟨*proof*⟩

definition *Sink-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-Sink.default-node-properties P*))

interpretation *Sink-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-Sink.default-node-properties*
and *sinvar-spec*=*SINVAR-Sink.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-Sink.receiver-violation*
and *offending-flows-impl*=*Sink-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*Sink-eval*
⟨*proof*⟩

6.4.4 Sink packing

definition *SINVAR-LIB-Sink* :: (*'v*::*vertex*, *node-config*) *TopoS-packed* **where**
SINVAR-LIB-Sink ≡
⦇ *nm-name* = *''Sink''*,
nm-receiver-violation = *SINVAR-Sink.receiver-violation*,
nm-default = *SINVAR-Sink.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *Sink-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *Sink-eval*
⦈
interpretation *SINVAR-LIB-Sink-interpretation*: *TopoS-modelLibrary SINVAR-LIB-Sink SINVAR-Sink.sinvar*
⟨*proof*⟩

Examples

definition *example-net-sink* :: *nat list-graph* **where**
example-net-sink ≡ ⦇ *nodesL* = [*1*::*nat*,*2*,*3*, *8*, *11*,*12*],
edgesL = [(*1*,*8*),(*1*,*2*), (*2*,*8*),(*3*,*8*),(*4*,*8*), (*2*,*3*),(*3*,*2*), (*11*,*8*),(*12*,*8*), (*11*,*12*), (*1*,*12*)] ⦈
value *wf-list-graph example-net-sink*

definition *example-conf-sink* **where**
example-conf-sink ≡ (λ*e*. *SINVAR-Sink.default-node-properties*)(*8*:= *Sink*, *2*:= *SinkPool*, *3*:= *SinkPool*, *4*:= *SinkPool*)

value *sinvar example-net-sink example-conf-sink*
value *Sink-offending-list example-net-sink example-conf-sink*

definition *example-net-sink-invalid* **where**
example-net-sink-invalid ≡ *example-net-sink*⦇*edgesL* := (*2*,*1*)#(*8*,*11*)#(*8*,*2*)#(*edgesL example-net-sink*)⦈

value *sinvar example-net-sink-invalid example-conf-sink*
value *Sink-offending-list example-net-sink-invalid example-conf-sink*

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-SubnetsInGW*
imports../*TopoS-Helper*
begin

6.5 SecurityInvariant SubnetsInGW

datatype *subnets* = *Member* | *InboundGateway* | *Unassigned*

definition *default-node-properties* :: *subnets*
where *default-node-properties* ≡ *Unassigned*

fun *allowed-subnet-flow* :: *subnets* ⇒ *subnets* ⇒ *bool* **where**
allowed-subnet-flow Member - = *True* |
allowed-subnet-flow InboundGateway - = *True* |
allowed-subnet-flow Unassigned Unassigned = *True* |
allowed-subnet-flow Unassigned InboundGateway = *True*|
allowed-subnet-flow Unassigned Member = *False*

fun *sinvar* :: ′*v graph* ⇒ (′*v* ⇒ *subnets*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. *allowed-subnet-flow* (*nP e1*) (*nP e2*))

definition *receiver-violation* :: *bool* **where** *receiver-violation* = *False*

6.5.1 Preliminaries

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
⟨*proof*⟩

6.5.2 ENF

lemma *Unassigned-not-to-Member*: ¬ *allowed-subnet-flow Unassigned Member*
⟨*proof*⟩
lemma *All-to-Unassigned*: *allowed-subnet-flow e1 Unassigned*
⟨*proof*⟩
lemma *Member-to-All*: *allowed-subnet-flow Member e2*
⟨*proof*⟩
lemma *Unassigned-default-candidate*: ∀ *nP e1 e2*. ¬ *allowed-subnet-flow* (*nP e1*) (*nP e2*) ⟶ ¬ *allowed-subnet-flow Unassigned* (*nP e2*)
⟨*proof*⟩
lemma *allowed-subnet-flow-refl*: *allowed-subnet-flow e e*
⟨*proof*⟩

lemma *SubnetsInGW-ENF*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar allowed-subnet-flow*
⟨*proof*⟩
lemma *SubnetsInGW-ENF-refl*: *SecurityInvariant-withOffendingFlows.ENF-refl sinvar allowed-subnet-flow*
⟨*proof*⟩

definition *SubnetsInGW-offending-set*:: *'v graph* ⇒ (*'v* ⇒ *subnets*) ⇒ (*'v* × *'v*) *set set* **where**
SubnetsInGW-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* ∈ *edges G*. *case e of* (*e1*,*e2*) ⇒ ¬ *allowed-subnet-flow* (*nP e1*) (*nP e2*)} })
lemma *SubnetsInGW-offending-set*:
SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = *SubnetsInGW-offending-set*
⟨*proof*⟩

interpretation *SubnetsInGW*: *SecurityInvariant-ACS*
where *default-node-properties* = *SINVAR-SubnetsInGW.default-node-properties*
and *sinvar* = *SINVAR-SubnetsInGW.sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *SubnetsInGW-offending-set*
⟨*proof*⟩

lemma *TopoS-SubnetsInGW*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

hide-fact (**open**) *sinvar-mono*
hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-SubnetsInGW-impl*
imports *SINVAR-SubnetsInGW* ../*TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-SubnetsInGW-impl* => (*Scala*) *SINVAR-SubnetsInGW*

6.5.3 SecurityInvariant SubnetsInGw List Implementation

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *subnets*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *set* (*edgesL G*). *SINVAR-SubnetsInGW.allowed-subnet-flow* (*nP e1*) (*nP e2*))

definition *SubnetsInGW-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *subnets*) ⇒ (*'v* × *'v*) *list list* **where**
SubnetsInGW-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G*. *case e of* (*e1*,*e2*) ⇒ ¬ *allowed-subnet-flow* (*nP e1*) (*nP e2*)]])

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-SubnetsInGW.default-node-properties*))

lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-SubnetsInGW.default-node-properties P = NetModel-node-props P*
⟨*proof*⟩

definition *SubnetsInGW-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-SubnetsInGW.default-node-properties P*))

interpretation *SubnetsInGW-impl*:*TopoS-List-Impl*
where *default-node-properties=SINVAR-SubnetsInGW.default-node-properties*
and *sinvar-spec=SINVAR-SubnetsInGW.sinvar*
and *sinvar-impl=sinvar*
and *receiver-violation=SINVAR-SubnetsInGW.receiver-violation*
and *offending-flows-impl=SubnetsInGW-offending-list*
and *node-props-impl=NetModel-node-props*
and *eval-impl=SubnetsInGW-eval*
⟨*proof*⟩

6.5.4 SubnetsInGW packing

definition *SINVAR-LIB-SubnetsInGW* :: (*'v*::*vertex*, *subnets*) *TopoS-packed* **where**
SINVAR-LIB-SubnetsInGW ≡
⦇ *nm-name* = *''SubnetsInGW''*,
nm-receiver-violation = *SINVAR-SubnetsInGW.receiver-violation*,
nm-default = *SINVAR-SubnetsInGW.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *SubnetsInGW-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *SubnetsInGW-eval*
⦈
interpretation *SINVAR-LIB-SubnetsInGW-interpretation*: *TopoS-modelLibrary SINVAR-LIB-SubnetsInGW SINVAR-SubnetsInGW.sinvar*
⟨*proof*⟩

Examples

definition *example-net-sub* :: *nat list-graph* **where**
example-net-sub ≡ ⦇ *nodesL* = [*1*::*nat*,*2*,*3*,*4*, *8*, *11*,*12*,*42*],
edgesL = [(*1*,*2*),(*1*,*3*),(*1*,*4*),(*2*,*1*),(*2*,*3*),(*2*,*4*),(*3*,*1*),(*3*,*2*),(*3*,*4*),(*4*,*1*),(*4*,*2*),(*4*,*3*),
(*8*,*1*),(*8*,*2*),
(*8*,*11*),
(*11*,*8*), (*12*,*8*),
(*11*,*42*), (*12*,*42*), (*8*,*42*)] ⦈
value *wf-list-graph example-net-sub*

definition *example-conf-sub* **where**
example-conf-sub ≡ ((λ*e*. *SINVAR-SubnetsInGW.default-node-properties*)
(*1* := *Member*, *2*:= *Member*, *3*:= *Member*, *4*:=*Member*,
8:=*InboundGateway*))

value *sinvar example-net-sub example-conf-sub*

definition *example-net-sub-invalid* **where**
example-net-sub-invalid ≡ *example-net-sub*⦇*edgesL* := (*42*,*4*)#(*edgesL example-net-sub*)⦈

value *sinvar example-net-sub-invalid example-conf-sub*
value *SubnetsInGW-offending-list example-net-sub-invalid example-conf-sub*

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-CommunicationPartners*
imports ../*TopoS-Helper*
begin

6.6 SecurityInvariant CommunicationPartners

Idea of this securityinvariant: Only some nodes can communicate with Master nodes. It constrains who may access master nodes, Master nodes can access the world (except other prohibited master nodes). A node configured as Master has a list of nodes that can access it. Also, in order to be able to access a Master node, the sender must be denoted as a node we Care about. By default, all nodes are set to DontCare, thus they cannot access Master nodes. But they can access all other DontCare nodes and Care nodes.

TL;DR: An access control list determines who can access a master node.

datatype *'v node-config = DontCare | Care | Master 'v list*

definition *default-node-properties* :: *'v node-config*
  where *default-node-properties = DontCare*

Unrestricted accesses among DontCare nodes!

fun *allowed-flow* :: $'v$ *node-config* $\Rightarrow$ $'v$ $\Rightarrow$ $'v$ *node-config* $\Rightarrow$ $'v$ $\Rightarrow$ *bool* **where**
  allowed-flow DontCare - DontCare - = True |
  allowed-flow DontCare - Care - = True |
  allowed-flow DontCare - (Master -) - = False |
  allowed-flow Care - Care - = True |
  allowed-flow Care - DontCare - = True |
  allowed-flow Care s (Master M) r = $(s \in set\ M)$ |
  allowed-flow (Master -) s (Master M) r = $(s \in set\ M)$ |
  allowed-flow (Master -) - Care - = True |
  allowed-flow (Master -) - DontCare - = True

fun *sinvar* :: $'v$ *graph* $\Rightarrow$ $('v \Rightarrow 'v$ *node-config*$)$ $\Rightarrow$ *bool* **where**
  sinvar G nP = $(\forall\ (s,r) \in edges\ G.\ s \neq r \longrightarrow allowed\text{-}flow\ (nP\ s)\ s\ (nP\ r)\ r)$

definition *receiver-violation* :: *bool* **where** *receiver-violation = False*

6.6.1 Preliminaries

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
  ⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*

where *sinvar* = *sinvar*
⟨*proof*⟩

6.6.2 ENRnr

lemma *CommunicationPartners-ENRnrSR*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-not-r sinvar allowed-flow*
⟨*proof*⟩
lemma *Unassigned-weakrefl*: $\forall$ *s r. allowed-flow DontCare s DontCare r*
⟨*proof*⟩
lemma *Unassigned-botdefault*: $\forall$ *s r.* (*nP r*) $\neq$ *DontCare* $\longrightarrow \neg$ *allowed-flow* (*nP s*) *s* (*nP r*) *r* $\longrightarrow$ $\neg$ *allowed-flow DontCare s* (*nP r*) *r*
⟨*proof*⟩
lemma $\neg$ *allowed-flow DontCare s* (*Master M*) *r* ⟨*proof*⟩
lemma $\neg$ *allowed-flow any s* (*Master* []) *r* ⟨*proof*⟩

lemma *All-to-Unassigned*: $\forall$ *s r. allowed-flow* (*nP s*) *s DontCare r*
⟨*proof*⟩
lemma *Unassigned-default-candidate*: $\forall$ *s r.* $\neg$ *allowed-flow* (*nP s*) *s* (*nP r*) *r* $\longrightarrow \neg$ *allowed-flow DontCare s* (*nP r*) *r*
⟨*proof*⟩

definition *CommunicationPartners-offending-set*:: *'v graph* $\Rightarrow$ (*'v* $\Rightarrow$ *'v node-config*) $\Rightarrow$ (*'v* $\times$ *'v*) *set set* **where**
CommunicationPartners-offending-set G nP = (*if sinvar G nP then*
{}
else
{ {*e* $\in$ *edges G. case e of* (*e1*,*e2*) $\Rightarrow$ *e1* $\neq$ *e2* $\wedge$ $\neg$ *allowed-flow* (*nP e1*) *e1* (*nP e2*) *e2*} })
lemma *CommunicationPartners-offending-set*:
SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = *CommunicationPartners-offending-set*
⟨*proof*⟩

interpretation *CommunicationPartners*: *SecurityInvariant-ACS*
where *default-node-properties* = *default-node-properties*
and *sinvar* = *sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar* = *CommunicationPartners-offending-set*
⟨*proof*⟩

lemma *TopoS-SubnetsInGW*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

Example:

lemma *sinvar* (|*nodes* = {''*db1*'', ''*db2*'', ''*h1*'', ''*h2*'', ''*foo*'', ''*bar*''},
edges = {(''*h1*'', ''*db1*''), (''*h2*'', ''*db1*''), (''*h1*'', ''*h2*''),
(''*db1*'', ''*h1*''), (''*db1*'', ''*foo*''), (''*db1*'', ''*db2*''), (''*db1*'', ''*db1*''),
(''*h1*'', ''*foo*''), (''*foo*'', ''*h1*''), (''*foo*'', ''*bar*'')}|)
(((((λ*h. default-node-properties*)(''*h1*'' := *Care*))(''*h2*'' := *Care*))
(''*db1*'' := *Master* [''*h1*'', ''*h2*'']))(''*db2*'' := *Master* [''*db1*''])) ⟨*proof*⟩

hide-fact (**open**) *sinvar-mono*
hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-CommunicationPartners-impl*
imports *SINVAR-CommunicationPartners ../TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-CommunicationPartners-impl* => (*Scala*) *SINVAR-CommunicationPartners*

6.6.3 SecurityInvariant CommunicationPartners List Implementation

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *'v node-config*) ⇒ *bool* **where**
sinvar G nP = (∀ (*s*,*r*) ∈ *set* (*edgesL G*). *s* ≠ *r* ⟶ *SINVAR-CommunicationPartners.allowed-flow* (*nP s*) *s* (*nP r*) *r*)

definition *CommunicationPartners-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *'v node-config*) ⇒ (*'v* × *'v*) *list list* **where**
CommunicationPartners-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G*. *case e of* (*e1*,*e2*) ⇒ *e1* ≠ *e2* ∧ ¬ *allowed-flow* (*nP e1*) *e1* (*nP e2*) *e2*]])

thm *SINVAR-CommunicationPartners.CommunicationPartners.node-props.simps*
definition *NetModel-node-props* (*P*::(*'v*::*vertex*, *'v node-config*) *TopoS-Params*) =
(λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-CommunicationPartners.default-node-*
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-CommunicationPartners.default-node-properties P* = *NetModel-node-props P*
⟨*proof*⟩

definition *CommunicationPartners-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-CommunicationPartners.default-node-properties P*))

interpretation *CommunicationPartners-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-CommunicationPartners.default-node-properties*
and *sinvar-spec*=*SINVAR-CommunicationPartners.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-CommunicationPartners.receiver-violation*
and *offending-flows-impl*=*CommunicationPartners-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*CommunicationPartners-eval*
⟨*proof*⟩

6.6.4 CommunicationPartners packing

definition *SINVAR-LIB-CommunicationPartners* :: (*'v*::*vertex*, *'v SINVAR-CommunicationPartners.node-config*) *TopoS-packed* **where**
SINVAR-LIB-CommunicationPartners ≡
(| *nm-name* = *''CommunicationPartners''*,
nm-receiver-violation = *SINVAR-CommunicationPartners.receiver-violation*,
nm-default = *SINVAR-CommunicationPartners.default-node-properties*,
nm-sinvar = *sinvar*,

nm-offending-flows = CommunicationPartners-offending-list,
nm-node-props = NetModel-node-props,
nm-eval = CommunicationPartners-eval
⦈
interpretation *SINVAR-LIB-CommunicationPartners-interpretation*: *TopoS-modelLibrary SINVAR-LIB-Communica*
SINVAR-CommunicationPartners.sinvar
⟨*proof*⟩

Examples

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-NoRefl*
imports ../*TopoS-Helper*
begin

6.7 SecurityInvariant NoRefl

Hosts are not allowed to communicate with themselves.

This can be used to effectively lift hosts to roles. Just list all roles that are allowed to communicate with themselves. Otherwise, communication between hosts of the same role (group) is prohibited. Useful in conjunction with the security gateway.

datatype *node-config = NoRefl | Refl*

definition *default-node-properties* :: *node-config*
where *default-node-properties = NoRefl*

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *node-config*) ⇒ *bool* **where**
sinvar G nP = (∀ (*s*, *r*) ∈ *edges G*. *s* = *r* ⟶ *nP s* = *Refl*)

definition *receiver-violation* :: *bool* **where** *receiver-violation* = *False*

6.7.1 Preliminaries

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*
where *sinvar = sinvar*
⟨*proof*⟩

lemma *NoRfl-ENRsr*: *SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-sr sinvar*
(λ nP_s s nP_r r. $s = r \longrightarrow nP_s = Refl$)
⟨*proof*⟩

definition *NoRefl-offending-set*:: *'v graph* ⇒ (*'v* ⇒ *node-config*) ⇒ (*'v* × *'v*) *set set* **where**
NoRefl-offending-set G nP = (*if sinvar G nP then*
{}
else

{ {e ∈ edges G. case e of (e1,e2) ⇒ e1 = e2 ∧ nP e1 = NoRefl} })

thm *SecurityInvariant-withOffendingFlows.ENFsr-offending-set*[*OF NoRefl-ENRsr*]

lemma *NoRefl-offending-set*: *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = NoRefl-offending-set*
⟨*proof*⟩

lemma *NoRefl-unique-default*:
∀ *G f nP i. wf-graph G ∧ f ∈ set-offending-flows G nP ∧ i ∈ fst ‘ f ⟶ ¬ sinvar G* (*nP*(*i* := *otherbot*)) ⟹
otherbot = NoRefl
⟨*proof*⟩

interpretation *NoRefl*: *SecurityInvariant-ACS*
where *default-node-properties = default-node-properties*
and *sinvar = sinvar*
rewrites *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar = NoRefl-offending-set*
⟨*proof*⟩

It can also be interpreted as IFS

lemma *NoRefl-SecurityInvariant-IFS*: *SecurityInvariant-IFS sinvar default-node-properties*
⟨*proof*⟩

lemma *TopoS-NoRefl*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

hide-fact (**open**) *sinvar-mono*
hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-NoRefl-impl*
imports *SINVAR-NoRefl ../TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-NoRefl-impl* => (*Scala*) *SINVAR-NoRefl*

6.7.2 SecurityInvariant NoRefl List Implementation

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *node-config*) ⇒ *bool* **where**
sinvar G nP = (∀ (*s*,*r*) ∈ *set* (*edgesL G*). *s* = *r* ⟶ *nP s* = *Refl*)

definition *NoRefl-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *node-config*) ⇒ (*'v* × *'v*) *list list* **where**
NoRefl-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G. case e of* (*e1*,*e2*) ⇒ *e1* = *e2* ∧ *nP e1* = *NoRefl*]])

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-NoRefl.default-node-properties*))

lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-NoRefl.default-node-properties P* = *NetModel-node-props P*
⟨*proof*⟩

definition *NoRefl-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-NoRefl.default-node-properties P*))

interpretation *NoRefl-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-NoRefl.default-node-properties*
and *sinvar-spec*=*SINVAR-NoRefl.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-NoRefl.receiver-violation*
and *offending-flows-impl*=*NoRefl-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*NoRefl-eval*
⟨*proof*⟩

6.7.3 PolEnforcePoint packing

definition *SINVAR-LIB-NoRefl* :: (*'v*::*vertex*, *node-config*) *TopoS-packed* **where**
SINVAR-LIB-NoRefl ≡
⦇ *nm-name* = *''NoRefl''*,
nm-receiver-violation = *SINVAR-NoRefl.receiver-violation*,
nm-default = *SINVAR-NoRefl.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *NoRefl-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *NoRefl-eval*
⦈
interpretation *SINVAR-LIB-NoRefl-interpretation*: *TopoS-modelLibrary SINVAR-LIB-NoRefl SINVAR-NoRefl.sinvar*
⟨*proof*⟩

Examples

definition *example-net* :: *nat list-graph* **where**
example-net ≡ ⦇ *nodesL* = [*1*::*nat*,*2*,*3*],
edgesL = [(*1*,*2*),(*2*,*2*),(*2*,*1*),(*1*,*3*)] ⦈
lemma *wf-list-graph example-net* ⟨*proof*⟩

definition *example-conf* **where**
example-conf ≡ ((λ*e*. *SINVAR-NoRefl.default-node-properties*)(*2*:= *Refl*))

lemma *sinvar example-net example-conf* ⟨*proof*⟩
lemma *NoRefl-offending-list example-net* (λ*e*. *SINVAR-NoRefl.default-node-properties*) = [[(*2*, *2*)]]
⟨*proof*⟩

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-Tainting-impl*
imports *SINVAR-Tainting* ../*TopoS-Interface-impl*

begin

6.7.4 SecurityInvariant Tainting List Implementation

code-identifier code-module *SINVAR-Tainting-impl* => (*Scala*) *SINVAR-Tainting*

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *SINVAR-Tainting.taints*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *set* (*edgesL G*). (*nP e1*) ⊆ (*nP e2*))

definition *Tainting-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *SINVAR-Tainting.taints*) ⇒ (*'v* × *'v*) *list list* **where**
Tainting-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G*. *case e of* (*e1*,*e2*) ⇒ ¬(*nP e1*) ⊆ (*nP e2*)]])

definition *NetModel-node-props P* =
(λ *i*. (*case* (*node-properties P*) *i of*
Some property ⇒ *property*
| *None* ⇒ *SINVAR-Tainting.default-node-properties*))

lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-Tainting.default-node-properties P* = *NetModel-node-props P*
⟨*proof*⟩

definition *Tainting-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-Tainting.default-node-properties P*))

interpretation *Tainting-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-Tainting.default-node-properties*
and *sinvar-spec*=*SINVAR-Tainting.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-Tainting.receiver-violation*
and *offending-flows-impl*=*Tainting-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*Tainting-eval*
⟨*proof*⟩

6.7.5 Tainting packing

definition *SINVAR-LIB-Tainting* :: (*'v*::*vertex*, *SINVAR-Tainting.taints*) *TopoS-packed* **where**
SINVAR-LIB-Tainting ≡
⦇ *nm-name* = *''Tainting''*,
nm-receiver-violation = *SINVAR-Tainting.receiver-violation*,
nm-default = *SINVAR-Tainting.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *Tainting-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *Tainting-eval*
⦈
interpretation *SINVAR-LIB-BLPbasic-interpretation*: *TopoS-modelLibrary SINVAR-LIB-Tainting*

SINVAR-Tainting.sinvar
⟨*proof*⟩

6.7.6 Example

context
begin
private definition *tainting-example* :: *string list-graph* **where**
tainting-example ≡ (| *nodesL* = [''*produce 1*'',
''*produce 2*'',
''*produce 3*'',
''*read 1 2*'',
''*read 3*'',
''*consume 1 2 3*'',
''*consume 3*''],
edgesL =[(''*produce 1*'', ''*read 1 2*''),
(''*produce 2*'', ''*read 1 2*''),
(''*produce 3*'', ''*read 3*''),
(''*read 3*'', ''*read 1 2*''),
(''*read 1 2*'', ''*consume 1 2 3*''),
(''*read 3*'', ''*consume 3*'')] |)
lemma *wf-list-graph tainting-example* ⟨*proof*⟩ **definition** *tainting-example-props* :: *string* ⇒ *SINVAR-Tainting.taints* **where**
tainting-example-props ≡ (λ *n. SINVAR-Tainting.default-node-properties*)
(''*produce 1*'' := {''*1*''},
''*produce 2*'' := {''*2*''},
''*produce 3*'' := {''*3*''},
''*read 1 2*'' := {''*1*'',''*2*'', ''*3*''},
''*read 3*'' := {''*3*''},
''*consume 1 2 3*'' := {''*1*'',''*2*'',''*3*''},
''*consume 3*'' := {''*3*''})
private lemma *sinvar tainting-example tainting-example-props* ⟨*proof*⟩
end

export-code *SINVAR-LIB-Tainting* **checking** *Scala*

hide-const (**open**) *NetModel-node-props Tainting-offending-list Tainting-eval*

hide-const (**open**) *sinvar*

end
theory *SINVAR-TaintingTrusted-impl*
imports *SINVAR-TaintingTrusted ../TopoS-Interface-impl*
begin

6.7.7 SecurityInvariant Tainting with Trust List Implementation

code-identifier code-module *SINVAR-Tainting-impl* => (*Scala*) *SINVAR-Tainting*

lemma $A - B \subseteq C \longleftrightarrow (\forall a{\in}A.\ a \in C \vee a \in B)$ ⟨*proof*⟩
lemma $\neg(A - B \subseteq C) \longleftrightarrow (\exists\, a \in A.\ a \notin C \wedge a \notin B)$ ⟨*proof*⟩

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *SINVAR-TaintingTrusted.taints*) ⇒ *bool* **where**

sinvar G nP = (∀ (*v1*,*v2*) ∈ *set* (*edgesL G*). *taints* (*nP v1*) − *untaints* (*nP v1*) ⊆ *taints* (*nP v2*))

export-code *sinvar* **checking** *SML*
value[*code*] *sinvar* (| *nodesL* = [], *edgesL* =[] |) (λ-. *SINVAR-TaintingTrusted.default-node-properties*)
lemma *sinvar* (| *nodesL* = [], *edgesL* =[] |) (λ-. *SINVAR-TaintingTrusted.default-node-properties*)
⟨*proof*⟩

definition *TaintingTrusted-offending-list*
:: *'v list-graph* ⇒ (*'v* ⇒ *SINVAR-TaintingTrusted.taints*) ⇒ (*'v* × *'v*) *list list* **where**
TaintingTrusted-offending-list G nP = (*if sinvar G nP then*
[]
else
[[*e* ← *edgesL G*. *case e of* (*v1*,*v2*) ⇒ ¬(*taints* (*nP v1*) − *untaints* (*nP v1*) ⊆ *taints* (*nP v2*))]])

export-code *TaintingTrusted-offending-list* **checking** *SML*

definition *NetModel-node-props P* =
(λ *i*. (*case* (*node-properties P*) *i of*
Some property ⇒ *property*
| *None* ⇒ *SINVAR-TaintingTrusted.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-TaintingTrusted.default-node-properties P* = *NetModel-node-props P*
⟨*proof*⟩

definition *TaintingTrusted-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-TaintingTrusted.default-node-properties P*))

interpretation *TaintingTrusted-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-TaintingTrusted.default-node-properties*
and *sinvar-spec*=*SINVAR-TaintingTrusted.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-TaintingTrusted.receiver-violation*
and *offending-flows-impl*=*TaintingTrusted-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*TaintingTrusted-eval*
⟨*proof*⟩

6.7.8 TaintingTrusted packing

definition *SINVAR-LIB-TaintingTrusted* :: (*'v*::*vertex*, *SINVAR-TaintingTrusted.taints*) *TopoS-packed*
where
SINVAR-LIB-TaintingTrusted ≡
(| *nm-name* = *''TaintingTrusted''*,
nm-receiver-violation = *SINVAR-TaintingTrusted.receiver-violation*,
nm-default = *SINVAR-TaintingTrusted.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *TaintingTrusted-offending-list*,

nm-node-props = *NetModel-node-props*,
nm-eval = *TaintingTrusted-eval*
⦈
interpretation *SINVAR-LIB-BLPbasic-interpretation*: *TopoS-modelLibrary SINVAR-LIB-TaintingTrusted SINVAR-TaintingTrusted.sinvar*
⟨*proof*⟩

6.7.9 Example

context
begin
private definition *tainting-example* :: *string list-graph* **where**
tainting-example ≡ ⦇ *nodesL* = [*''produce 1''*,
''produce 2'',
''produce 3'',
''read 1 2'',
''read 3'',
''consume 1 2 3'',
''consume 3''],
edgesL =[(*''produce 1''*, *''read 1 2''*),
(*''produce 2''*, *''read 1 2''*),
(*''produce 3''*, *''read 3''*),
(*''read 3''*, *''read 1 2''*),
(*''read 1 2''*, *''consume 1 2 3''*),
(*''read 3''*, *''consume 3''*)] ⦈
lemma *wf-list-graph tainting-example* ⟨*proof*⟩ **definition** *tainting-example-props* :: *string* ⇒ *SINVAR-TaintingTrusted.taints* **where**
tainting-example-props ≡ (λ *n. SINVAR-TaintingTrusted.default-node-properties*)
(*''produce 1''* := *TaintsUntaints* {*''1''*} {},
''produce 2'' := *TaintsUntaints* {*''2''*} {},
''produce 3'' := *TaintsUntaints* {*''3''*} {},
''read 1 2'' := *TaintsUntaints* {*''3''*,*''foo''*} {*''1''*,*''2''*},
''read 3'' := *TaintsUntaints* {*''3''*} {},
''consume 1 2 3'' := *TaintsUntaints* {*''foo''*,*''3''*} {},
''consume 3'' := *TaintsUntaints* {*''3''*} {})

value *tainting-example-props* (*''consume 1 2 3''*)
value[*code*] *TaintingTrusted-offending-list tainting-example tainting-example-props*
private lemma *sinvar tainting-example tainting-example-props* ⟨*proof*⟩
end

export-code *SINVAR-LIB-TaintingTrusted* **checking** *Scala*
export-code *SINVAR-LIB-TaintingTrusted* **checking** *SML*

hide-const (**open**) *NetModel-node-props TaintingTrusted-offending-list TaintingTrusted-eval*

hide-const (**open**) *sinvar*

end
theory *SINVAR-Dependability*
imports ../*TopoS-Helper*
begin

6.8 SecurityInvariant Dependability

type-synonym *dependability-level* = *nat*

definition *default-node-properties* :: *dependability-level*
  where *default-node-properties* ≡ *0*

Less-equal other nodes depend on the output of a node than its dependability level.

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *dependability-level*) ⇒ *bool* **where**
  sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. (*num-reachable G e1*) ≤ (*nP e1*))

definition *receiver-violation* :: *bool* **where**
  receiver-violation ≡ *False*

It does not matter whether we iterate over all edges or all nodes. We chose all edges because it is in line with the other models.

fun *sinvar-nodes* :: *'v graph* ⇒ (*'v* ⇒ *dependability-level*) ⇒ *bool* **where**
  sinvar-nodes G nP = (∀ *v* ∈ *nodes G*. (*num-reachable G v*) ≤ (*nP v*))

theorem *sinvar-edges-nodes-iff*: *wf-graph G* ⟹
  sinvar-nodes G nP = *sinvar G nP*
⟨*proof*⟩

lemma *num-reachable-le-nodes*: ⟦ *wf-graph G* ⟧ ⟹ *num-reachable G v* ≤ *card* (*nodes G*)
  ⟨*proof*⟩

nP is valid if all dependability level are greater equal the total number of nodes in the graph

lemma ⟦ *wf-graph G*; ∀ *v* ∈ *nodes G*. *nP v* ≥ *card* (*nodes G*) ⟧ ⟹ *sinvar G nP*
  ⟨*proof*⟩

Generate a valid configuration to start from:

Takes arbitrary configuration, returns a valid one

fun *dependability-fix-nP* :: *'v graph* ⇒ (*'v* ⇒ *dependability-level*) ⇒ (*'v* ⇒ *dependability-level*) **where**
  dependability-fix-nP G nP = (λ*v*. *if num-reachable G v* ≤ (*nP v*) *then* (*nP v*) *else num-reachable G v*)

dependability-fix-nP always gives you a valid solution

lemma *dependability-fix-nP-valid*: ⟦ *wf-graph G* ⟧ ⟹ *sinvar G* (*dependability-fix-nP G nP*)
  ⟨*proof*⟩

furthermore, it gives you a minimal solution, i.e. if someone supplies a configuration with a value lower than calculated by *dependability-fix-nP*, this is invalid!

lemma *dependability-fix-nP-minimal-solution*: ⟦ *wf-graph G*; ∃ *v* ∈ *nodes G*. (*nP v*) < (*dependability-fix-nP G* (λ-. *0*)) *v* ⟧ ⟹ ¬ *sinvar G nP*
  ⟨*proof*⟩

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
⟨*proof*⟩

interpretation *Dependability*: *SecurityInvariant-ACS*
where *default-node-properties* = *SINVAR-Dependability.default-node-properties*
and *sinvar* = *SINVAR-Dependability.sinvar*
⟨*proof*⟩

lemma *TopoS-Dependability*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-Dependability-impl*
imports *SINVAR-Dependability ../TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-Dependability-impl* => (*Scala*) *SINVAR-Dependability*

6.8.1 SecurityInvariant Dependability List Implementation

Less-equal other nodes depend on the output of a node than its dependability level.

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *dependability-level*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *set* (*edgesL G*). (*num-reachable G e1*) ≤ (*nP e1*))

value *sinvar*
(| *nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] |)
(λ*e*. *3*)
value *sinvar*
(| *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] |)
(λ*e*. *2*)

Generate a valid configuration to start from:

fun *dependability-fix-nP* :: *'v list-graph* ⇒ (*'v* ⇒ *dependability-level*) ⇒ (*'v* ⇒ *dependability-level*)
where
dependability-fix-nP G nP = (λ*v*. *let nr* = *num-reachable G v in* (*if nr* ≤ (*nP v*) *then* (*nP v*) *else nr*))

theorem *dependability-fix-nP-impl-correct*: *wf-list-graph G* ⟹ *dependability-fix-nP G nP* = *SINVAR-Dependability.dependability-fix-nP* (*list-graph-to-graph G*) *nP*
⟨*proof*⟩

value *let G* = (| *nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*1*), (*2*,*1*), (*3*,*1*), (*4*,*1*), (*1*,*2*), (*1*,*3*)] |) *in* (*let nP* = *dependability-fix-nP G* (λ*e*. *0*) *in map* (λ*v*. *nP v*) (*nodesL G*))

value *let G* = ⦇ *nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*1*)] ⦈ *in* (*let nP* = *dependability-fix-nP G* (*λe. 0*) *in map* (*λv. nP v*) (*nodesL G*))

definition *Dependability-offending-list*:: *'v list-graph* ⇒ (*'v* ⇒ *dependability-level*) ⇒ (*'v* × *'v*) *list list*
where
Dependability-offending-list = *Generic-offending-list sinvar*

definition *NetModel-node-props P* = (λ *i.* (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-Dependability.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-Dependability.default-node-properties P* = *NetModel-node-props P*
⟨*proof*⟩

definition *Dependability-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-Dependability.default-node-properties P*))

lemma *sinvar-correct*: *wf-list-graph G* ⟹ *SINVAR-Dependability.sinvar* (*list-graph-to-graph G*) *nP* = *sinvar G nP*
⟨*proof*⟩

interpretation *Dependability-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-Dependability.default-node-properties*
and *sinvar-spec*=*SINVAR-Dependability.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-Dependability.receiver-violation*
and *offending-flows-impl*=*Dependability-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*Dependability-eval*
⟨*proof*⟩

6.8.2 Dependability packing

definition *SINVAR-LIB-Dependability* :: (*'v*::*vertex*, *SINVAR-Dependability.dependability-level*) *TopoS-packed*
where
SINVAR-LIB-Dependability ≡
⦇ *nm-name* = *''Dependability''*,
nm-receiver-violation = *SINVAR-Dependability.receiver-violation*,
nm-default = *SINVAR-Dependability.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *Dependability-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *Dependability-eval*
⦈
interpretation *SINVAR-LIB-Dependability-interpretation*: *TopoS-modelLibrary SINVAR-LIB-Dependability*

SINVAR-Dependability.sinvar
⟨*proof*⟩

Example:

value *let G* = ⦇ *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] ⦈
in sinvar G ((λ *n*. *SINVAR-Dependability.default-node-properties*)(*1*:=*3*, *2*:=*2*, *3*:=*1*, *4*:=*0*, *8*:=*2*, *9*:=*2*, *10*:=*0*))

value *let G* = ⦇ *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] ⦈
in sinvar G ((λ *n*. *SINVAR-Dependability.default-node-properties*)(*1*:=*10*, *2*:=*10*, *3*:=*10*, *4*:=*10*, *8*:=*10*, *9*:=*10*, *10*:=*10*))

value *let G* = ⦇ *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] ⦈
in sinvar G ((λ *n*. *2*))

value *let G* = ⦇ *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] ⦈
in Dependability-eval G ⦇*node-properties*=[*1*↦*3*, *2*↦*2*, *3*↦*1*, *4*↦*0*, *8*↦*2*, *9*↦*2*, *10*↦*0*] ⦈

value *Dependability-offending-list* ⦇ *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] ⦈ (λ *n*. *2*)

hide-fact (**open**) *sinvar-correct*
hide-const (**open**) *sinvar NetModel-node-props*

end
theory *SINVAR-NonInterference*
imports ../*TopoS-Helper*
begin

6.9 SecurityInvariant NonInterference

datatype *node-config* = *Interfering* | *Unrelated*

definition *default-node-properties* :: *node-config*
where *default-node-properties* = *Interfering*

definition *undirected-reachable* :: ′*v graph* ⇒ ′*v* => ′*v set* **where**
undirected-reachable G v = (*succ-tran* (*undirected G*) *v*) − {*v*}

lemma *undirected-reachable-mono*:
E′ ⊆ *E* ⟹ *undirected-reachable* ⦇*nodes* = *N*, *edges* = *E*′⦈ *n* ⊆ *undirected-reachable* ⦇*nodes* = *N*, *edges* = *E*⦈ *n*
⟨*proof*⟩

fun *sinvar* :: ′*v graph* ⇒ (′*v* ⇒ *node-config*) ⇒ *bool* **where**
sinvar G nP = (∀ *n* ∈ (*nodes G*). (*nP n*) = *Interfering* ⟶ (*nP* ‘ (*undirected-reachable G n*)) ⊆ {*Unrelated*})

lemma *sinvar G nP* ⟷
(∀ *n* ∈ {*v*′ ∈ (*nodes G*). (*nP v*′) = *Interfering*}. {*nP v*′ | *v*′. *v*′ ∈ *undirected-reachable G n*} ⊆ {*Unrelated*})
⟨*proof*⟩

definition *receiver-violation* :: *bool* **where**
receiver-violation = *True*

simplifications for sets we need in the uniqueness proof

lemma *tmp1*: $\{(b, a).\ a = \textit{vertex-1} \wedge b = \textit{vertex-2}\} = \{(\textit{vertex-2}, \textit{vertex-1})\}$ ⟨*proof*⟩
lemma *tmp6*: $\{(\textit{vertex-1}, \textit{vertex-2}), (\textit{vertex-2}, \textit{vertex-1})\}^+ =$
$\{(\textit{vertex-1}, \textit{vertex-1}), (\textit{vertex-2}, \textit{vertex-2}), (\textit{vertex-1}, \textit{vertex-2}), (\textit{vertex-2}, \textit{vertex-1})\}$
⟨*proof*⟩
lemma *tmp2*: $(\textit{insert}\ (\textit{vertex-1}, \textit{vertex-2})\ \{(b, a).\ a = \textit{vertex-1} \wedge b = \textit{vertex-2}\})^+ =$
$\{(\textit{vertex-1}, \textit{vertex-1}), (\textit{vertex-2}, \textit{vertex-2}), (\textit{vertex-1}, \textit{vertex-2}), (\textit{vertex-2}, \textit{vertex-1})\}$
⟨*proof*⟩
lemma *tmp4*: $\{(e1, e2).\ e1 = \textit{vertex-1} \wedge e2 = \textit{vertex-2} \wedge (e1 = \textit{vertex-1} \longrightarrow e2 \neq \textit{vertex-2})\} = \{\}$ ⟨*proof*⟩
lemma *tmp5*: $\{(b, a).\ a = \textit{vertex-1} \wedge b = \textit{vertex-2} \vee a = \textit{vertex-1} \wedge b = \textit{vertex-2} \wedge (a = \textit{vertex-1} \longrightarrow b \neq \textit{vertex-2})\} =$
$\{(\textit{vertex-2}, \textit{vertex-1})\}$ ⟨*proof*⟩
lemma *unique-default-example*: *undirected-reachable* (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) *vertex-1* = {*vertex-2*}
⟨*proof*⟩
lemma *unique-default-example-hlp1*: *delete-edges* (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) {(*vertex-1*, *vertex-2*)} =
(|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {}|)
⟨*proof*⟩
lemma *unique-default-example-2*:
undirected-reachable (*delete-edges* (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) {(*vertex-1*,*vertex-2*)}) *vertex-1* = {}
⟨*proof*⟩
lemma *unique-default-example-3*:
undirected-reachable (*delete-edges* (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) {(*vertex-1*,*vertex-2*)}) *vertex-2* = {}
⟨*proof*⟩
lemma *unique-default-example-4*:
(*undirected-reachable* (*add-edge vertex-1 vertex-2* (*delete-edges* (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) {(*vertex-1*, *vertex-2*)})) *vertex-1*) = {*vertex-2*}
⟨*proof*⟩
lemma *unique-default-example-5*:
(*undirected-reachable* (*add-edge vertex-1 vertex-2* (*delete-edges* (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) {(*vertex-1*, *vertex-2*)})) *vertex-2*) = {*vertex-1*}
⟨*proof*⟩

lemma *empty-undirected-reachable-false*: $xb \in \textit{undirected-reachable}\ (\textit{delete-edges}\ G\ (\textit{edges}\ G))\ na \longleftrightarrow \textit{False}$
⟨*proof*⟩

6.9.1 monotonic and preliminaries

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*

where *sinvar* = *sinvar*
⟨*proof*⟩

interpretation *NonInterference*: *SecurityInvariant-IFS*
where *default-node-properties* = *SINVAR-NonInterference.default-node-properties*
and *sinvar* = *SINVAR-NonInterference.sinvar*
⟨*proof*⟩

lemma *TopoS-NonInterference*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

hide-const (**open**) *sinvar receiver-violation default-node-properties*

— Hide all the helper lemmas.
hide-fact *tmp1 tmp2 tmp4 tmp5 tmp6 unique-default-example unique-default-example-2 unique-default-example-3 unique-default-example-4 unique-default-example-5 empty-undirected-reachable-false*

end
theory *SINVAR-NonInterference-impl*
imports *SINVAR-NonInterference ../TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-NonInterference-impl* => (*Scala*) *SINVAR-NonInterference*

6.9.2 SecurityInvariant NonInterference List Implementation

definition *undirected-reachable* :: *'v list-graph* ⇒ *'v* => *'v list* **where**
undirected-reachable G v = *removeAll v* (*succ-tran* (*undirected G*) *v*)

lemma *undirected-reachable-set*: *set* (*undirected-reachable G v*) = {*e2*. (*v*,*e2*) ∈ (*set* (*edgesL* (*undirected G*)))$^+$} − {*v*}
⟨*proof*⟩

fun *sinvar-set* :: *'v list-graph* ⇒ (*'v* ⇒ *node-config*) ⇒ *bool* **where**
sinvar-set G nP = (∀ *n* ∈ *set* (*nodesL G*). (*nP n*) = *Interfering* ⟶ *set* (*map nP* (*undirected-reachable G n*)) ⊆ {*Unrelated*})

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *node-config*) ⇒ *bool* **where**
sinvar G nP = (∀ *n* ∈ *set* (*nodesL G*). (*nP n*) = *Interfering* ⟶ (*let result* = *remdups* (*map nP* (*undirected-reachable G n*)) *in result* = [] ∨ *result* = [*Unrelated*]))

lemma *P* = *Q* ⟹ (∀ *x*. *P x*) = (∀ *x*. *Q x*)
⟨*proof*⟩

lemma *sinvar-eq-help1*: *nP* ‘ *set* (*undirected-reachable G n*) = *set* (*map nP* (*undirected-reachable G n*))
⟨*proof*⟩

lemma *sinvar-eq-help2*: *set l* = {*Unrelated*} ⟹ *remdups l* = [*Unrelated*]
⟨*proof*⟩
lemma *sinvar-eq-help3*: (*let result* = *remdups* (*map nP* (*undirected-reachable G n*)) *in result* = [] ∨ *result* = [*Unrelated*]) = (*set* (*map nP* (*undirected-reachable G n*)) ⊆ {*Unrelated*})
⟨*proof*⟩

lemma *sinvar-list-eq-set*: *sinvar* = *sinvar-set*
⟨*proof*⟩

value *sinvar*
(| *nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] |)
(λ*e*. *SINVAR-NonInterference.default-node-properties*)
value *sinvar*
(| *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*)] |)
((λ*e*. *SINVAR-NonInterference.default-node-properties*)(*1*:= *Interfering*, *2*:= *Unrelated*, *3*:= *Unrelated*, *4*:= *Unrelated*))
value *sinvar*
(| *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*5*, *8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*5*,*4*), (*8*,*9*),(*9*,*8*)] |)
((λ*e*. *SINVAR-NonInterference.default-node-properties*)(*1*:= *Interfering*, *2*:= *Unrelated*, *3*:= *Unrelated*, *4*:= *Unrelated*))
value *sinvar*
(| *nodesL* = [*1*::*nat*], *edgesL* = [(*1*,*1*)] |)
((λ*e*. *SINVAR-NonInterference.default-node-properties*)(*1*:= *Interfering*))

value (*undirected-reachable* (| *nodesL* = [*1*::*nat*], *edgesL* = [(*1*,*1*)] |) *1*) = []

definition *NonInterference-offending-list*:: ′*v list-graph* ⇒ (′*v* ⇒ *node-config*) ⇒ (′*v* × ′*v*) *list list*
where
NonInterference-offending-list = *Generic-offending-list sinvar*

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-NonInterference.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-NonInterference.default-node-properties P* = *NetModel-node-props P*
⟨*proof*⟩

definition *NonInterference-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-NonInterference.default-node-properties P*))

lemma *sinvar-correct*: *wf-list-graph G* ⟹ *SINVAR-NonInterference.sinvar* (*list-graph-to-graph G*) *nP* = *sinvar G nP*
⟨*proof*⟩

interpretation *NonInterference-impl*: *TopoS-List-Impl*
where *default-node-properties=SINVAR-NonInterference.default-node-properties*
and *sinvar-spec=SINVAR-NonInterference.sinvar*
and *sinvar-impl=sinvar*
and *receiver-violation=SINVAR-NonInterference.receiver-violation*
and *offending-flows-impl=NonInterference-offending-list*
and *node-props-impl=NetModel-node-props*
and *eval-impl=NonInterference-eval*
⟨*proof*⟩

6.9.3 NonInterference packing

definition *SINVAR-LIB-NonInterference* :: (*'v*::*vertex*, *node-config*) *TopoS-packed* **where**
SINVAR-LIB-NonInterference ≡
⦇ *nm-name* = *''NonInterference''*,
nm-receiver-violation = *SINVAR-NonInterference.receiver-violation*,
nm-default = *SINVAR-NonInterference.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *NonInterference-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *NonInterference-eval*
⦈
interpretation *SINVAR-LIB-NonInterference-interpretation*: *TopoS-modelLibrary SINVAR-LIB-NonInterference SINVAR-NonInterference.sinvar*
⟨*proof*⟩

Example:

context begin
private definition *example-graph* = ⦇ *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*5*, *8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*5*,*4*), (*8*,*9*), (*9*,*8*)] ⦈
private definition*example-conf* = ((λ*e*. *SINVAR-NonInterference.default-node-properties*)
(*1*:= *Interfering*, *2*:= *Unrelated*, *3*:= *Unrelated*, *4*:= *Unrelated*, *8*:= *Unrelated*, *9*:= *Unrelated*))

private lemma ¬ *sinvar example-graph example-conf* ⟨*proof*⟩ **lemma** *NonInterference-offending-list example-graph example-conf* =
[[(*1*, *2*)], [(*2*, *3*)], [(*3*, *4*)], [(*5*, *4*)]] ⟨*proof*⟩
end

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-ACLcommunicateWith*
imports ../*TopoS-Helper*
begin

6.10 SecurityInvariant ACLcommunicateWith

An access control list strategy that says that hosts must only transitively access each other if allowed

Warning: this transitive model has exponential computational complexity

definition *default-node-properties* :: *'v list*
where *default-node-properties* ≡ []

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *'v list*) ⇒ *bool* **where**
sinvar G nP = (∀ *v* ∈ *nodes G*. (∀ *a* ∈ (*succ-tran G v*). *a* ∈ *set* (*nP v*)))

definition *receiver-violation* :: *bool* **where**
receiver-violation ≡ *False*

lemma *ACLcommunicateWith-sinvar-alternative*:
wf-graph G ⟹ *sinvar G nP* = (∀ (*e1*,*e2*) ∈ (*edges G*)$^+$. *e2* ∈ *set* (*nP e1*))
⟨*proof*⟩

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
⟨*proof*⟩

lemma *unique-default-example*: *succ-tran* (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) *vertex-2* = {}
⟨*proof*⟩

interpretation *ACLcommunicateWith*: *SecurityInvariant-ACS*
where *default-node-properties* = *SINVAR-ACLcommunicateWith.default-node-properties*
and *sinvar* = *SINVAR-ACLcommunicateWith.sinvar*
⟨*proof*⟩

lemma *TopoS-ACLcommunicateWith*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-ACLnotCommunicateWith*
imports ../*TopoS-Helper SINVAR-ACLcommunicateWith*
begin

6.11 SecurityInvariant ACLnotCommunicateWith

An access control list strategy that says that hosts must not transitively access each other.

node properties: a set of hosts this host must not access

definition *default-node-properties* :: *'v set*
where *default-node-properties* ≡ *UNIV*

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *'v set*) ⇒ *bool* **where**
sinvar G nP = (∀ *v* ∈ *nodes G*. ∀ *a* ∈ (*succ-tran G v*). *a* ∉ (*nP v*))

definition *receiver-violation* :: *bool* **where**
receiver-violation ≡ *False*

It is the inverse of *SINVAR-ACLcommunicateWith.sinvar*

lemma *ACLcommunicateNotWith-inverse-ACLcommunicateWith*:
∀ *v*. *UNIV* − *nP′ v* = *set* (*nP v*) ⟹ *SINVAR-ACLcommunicateWith.sinvar G nP* ⟷ *sinvar G nP′*
⟨*proof*⟩

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
⟨*proof*⟩

lemma *succ-tran-empty*: (*succ-tran* (|*nodes* = *nodes G*, *edges* = {}|) *v*) = {}
⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
⟨*proof*⟩

lemma *unique-default-example*: *succ-tran* (|*nodes* = {*vertex-1*, *vertex-2*}, *edges* = {(*vertex-1*, *vertex-2*)}|) *vertex-2* = {}
⟨*proof*⟩

interpretation *ACLnotCommunicateWith*: *SecurityInvariant-ACS*
where *default-node-properties* = *SINVAR-ACLnotCommunicateWith.default-node-properties*
and *sinvar* = *SINVAR-ACLnotCommunicateWith.sinvar*
⟨*proof*⟩

lemma *TopoS-ACLnotCommunicateWith*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-ACLnotCommunicateWith-impl*
imports *SINVAR-ACLnotCommunicateWith ../TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-ACLnotCommunicateWith-impl* => (*Scala*) *SINVAR-ACLnotCommunicateW*

6.11.1 SecurityInvariant ACLnotCommunicateWith List Implementation

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *'v set*) ⇒ *bool* **where**
sinvar G nP = (∀ *v* ∈ *set* (*nodesL G*). ∀ *a* ∈ *set* (*succ-tran G v*). *a* ∉ (*nP v*))

definition *NetModel-node-props* (*P*::(*'v*::*vertex*, *'v set*) *TopoS-Params*) =
(λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-ACLnotCommunicateWith.default-nod*
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-ACLnotCommunicateWith.default-node-properties*
P = *NetModel-node-props P*
⟨*proof*⟩

definition *ACLnotCommunicateWith-offending-list* = *Generic-offending-list sinvar*

definition *ACLnotCommunicateWith-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-ACLnotCommunicateWith.default-node-properties*
P))

lemma *sinvar-correct*: *wf-list-graph G* ⟹ *SINVAR-ACLnotCommunicateWith.sinvar* (*list-graph-to-graph*
G) *nP* = *sinvar G nP*
⟨*proof*⟩

interpretation *ACLnotCommunicateWith-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-ACLnotCommunicateWith.default-node-properties*
and *sinvar-spec*=*SINVAR-ACLnotCommunicateWith.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-ACLnotCommunicateWith.receiver-violation*
and *offending-flows-impl*=*ACLnotCommunicateWith-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*ACLnotCommunicateWith-eval*
⟨*proof*⟩

6.11.2 packing

definition *SINVAR-LIB-ACLnotCommunicateWith*:: (*'v*::*vertex*, *'v set*) *TopoS-packed* **where**
SINVAR-LIB-ACLnotCommunicateWith ≡
(| *nm-name* = *''ACLnotCommunicateWith''*,
nm-receiver-violation = *SINVAR-ACLnotCommunicateWith.receiver-violation*,
nm-default = *SINVAR-ACLnotCommunicateWith.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *ACLnotCommunicateWith-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *ACLnotCommunicateWith-eval*
|)
interpretation *SINVAR-LIB-ACLnotCommunicateWith-interpretation*: *TopoS-modelLibrary SINVAR-LIB-ACLnotCommunicateWith*
SINVAR-ACLnotCommunicateWith.sinvar
⟨*proof*⟩

Examples

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-ACLcommunicateWith-impl*
imports *SINVAR-ACLcommunicateWith* ../*TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-ACLcommunicateWith-impl* => (*Scala*) *SINVAR-ACLcommunicateWith*

6.11.3 List Implementation

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *'v list*) ⇒ *bool* **where**
sinvar G nP = (∀ *v* ∈ *set* (*nodesL G*). ∀ *a* ∈ (*set* (*succ-tran G v*)). *a* ∈ *set* (*nP v*))

definition *NetModel-node-props* (*P*::(*'v*::*vertex*, *'v list*) *TopoS-Params*) =
(λ *i*. (*case* (*node-properties P*) *i of Some property* ⇒ *property* | *None* ⇒ *SINVAR-ACLcommunicateWith.default-node-pr*
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-ACLcommunicateWith.default-node-properties*
P = *NetModel-node-props P*
⟨*proof*⟩

definition *ACLcommunicateWith-offending-list* = *Generic-offending-list sinvar*

definition *ACLcommunicateWith-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-ACLcommunicateWith.default-node-properties P*))

lemma *sinvar-correct*: *wf-list-graph G* ⟹ *SINVAR-ACLcommunicateWith.sinvar* (*list-graph-to-graph*
G) *nP* = *sinvar G nP*
⟨*proof*⟩

interpretation *SINVAR-ACLcommunicateWith-impl*:*TopoS-List-Impl*
where *default-node-properties*=*SINVAR-ACLcommunicateWith.default-node-properties*
and *sinvar-spec*=*SINVAR-ACLcommunicateWith.sinvar*
and *sinvar-impl*=*sinvar*
and *receiver-violation*=*SINVAR-ACLcommunicateWith.receiver-violation*
and *offending-flows-impl*=*ACLcommunicateWith-offending-list*
and *node-props-impl*=*NetModel-node-props*
and *eval-impl*=*ACLcommunicateWith-eval*
⟨*proof*⟩

6.11.4 packing

definition *SINVAR-LIB-ACLcommunicateWith*:: (*'v*::*vertex*, *'v list*) *TopoS-packed* **where**
SINVAR-LIB-ACLcommunicateWith ≡
⦇ *nm-name* = *''ACLcommunicateWith''*,
nm-receiver-violation = *SINVAR-ACLcommunicateWith.receiver-violation*,
nm-default = *SINVAR-ACLcommunicateWith.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *ACLcommunicateWith-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *ACLcommunicateWith-eval*
⦈
interpretation *SINVAR-LIB-ACLcommunicateWith-interpretation*: *TopoS-modelLibrary SINVAR-LIB-ACLcommunic*
SINVAR-ACLcommunicateWith.sinvar
⟨*proof*⟩

Examples

context begin

1 can access 2 and 3 2 can access 3

private lemma *sinvar*
⦇ *nodesL* = [*1*::*nat*, *2*, *3*],
edgesL = [(*1*,*2*), (*2*,*3*)]⦈
(((λ*v*. *SINVAR-ACLcommunicateWith.default-node-properties*)
(*1* := [*2*,*3*]))
(*2* := [*3*])) ⟨*proof*⟩

Everyone can access everyone, except for 1: 1 must not access 4. The offending flows may be any edge on the path from 1 to 4

lemma *ACLcommunicateWith-offending-list*
⦇ *nodesL* = [*1*::*nat*, *2*, *3*, *4*],
edgesL = [(*1*,*2*), (*2*,*3*), (*3*, *4*)]⦈
(((((λ*v*. *SINVAR-ACLcommunicateWith.default-node-properties*)
(*1* := [*1*,*2*,*3*]))
(*2* := [*1*,*2*,*3*,*4*]))
(*3* := [*1*,*2*,*3*,*4*]))
(*4* := [*1*,*2*,*3*,*4*])) =
[[(*1*, *2*)], [(*2*, *3*)], [(*3*, *4*)]] ⟨*proof*⟩

If we add the additional edge from 1 to 3, then the offending flows are either

(3.4) , because this disconnects 4 from the graph completely

- any pair of edges which disconnects 1 from 3

lemma *ACLcommunicateWith-offending-list*
⦇ *nodesL* = [*1*::*nat*, *2*, *3*, *4*],
edgesL = [(*1*,*2*), (*1*,*3*), (*2*,*3*), (*3*, *4*)]⦈
(((((λ*v*. *SINVAR-ACLcommunicateWith.default-node-properties*)
(*1* := [*1*,*2*,*3*]))
(*2* := [*1*,*2*,*3*,*4*]))
(*3* := [*1*,*2*,*3*,*4*]))
(*4* := [*1*,*2*,*3*,*4*])) =
[[(*1*, *2*), (*1*, *3*)], [(*1*, *3*), (*2*, *3*)], [(*3*, *4*)]] ⟨*proof*⟩
end

hide-const (**open**) *NetModel-node-props*
hide-const (**open**) *sinvar*

end
theory *SINVAR-Dependability-norefl*
imports ../*TopoS-Helper*
begin

6.12 SecurityInvariant *Dependability-norefl*

A version of the Dependability model but if a node reaches itself, it is ignored

type-synonym *dependability-level* = *nat*

definition *default-node-properties* :: *dependability-level*
where *default-node-properties* ≡ *0*

Less-equal other nodes depend on the output of a node than its dependability level.

fun *sinvar* :: *'v graph* ⇒ (*'v* ⇒ *dependability-level*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *edges G*. (*num-reachable-norefl G e1*) ≤ (*nP e1*))

definition *receiver-violation* :: *bool* **where**
receiver-violation ≡ *False*

lemma *sinvar-mono*: *SecurityInvariant-withOffendingFlows.sinvar-mono sinvar*
⟨*proof*⟩

interpretation *SecurityInvariant-preliminaries*
where *sinvar* = *sinvar*
⟨*proof*⟩

interpretation *Dependability*: *SecurityInvariant-ACS*
where *default-node-properties* = *SINVAR-Dependability-norefl.default-node-properties*
and *sinvar* = *SINVAR-Dependability-norefl.sinvar*
⟨*proof*⟩

lemma *TopoS-Dependability-norefl*: *SecurityInvariant sinvar default-node-properties receiver-violation*
⟨*proof*⟩

hide-const (**open**) *sinvar receiver-violation default-node-properties*

end
theory *SINVAR-Dependability-norefl-impl*
imports *SINVAR-Dependability-norefl ../TopoS-Interface-impl*
begin

code-identifier code-module *SINVAR-Dependability-norefl-impl* => (*Scala*) *SINVAR-Dependability-norefl*

6.12.1 SecurityInvariant Dependability norefl List Implementation

fun *sinvar* :: *'v list-graph* ⇒ (*'v* ⇒ *dependability-level*) ⇒ *bool* **where**
sinvar G nP = (∀ (*e1*,*e2*) ∈ *set* (*edgesL G*). (*num-reachable-norefl G e1*) ≤ (*nP e1*))

value *sinvar*
(| *nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] |)
(λ*e*. *3*)
value *sinvar*
(| *nodesL* = [*1*::*nat*,*2*,*3*,*4*,*8*,*9*,*10*], *edgesL* = [(*1*,*2*), (*2*,*3*), (*3*,*4*), (*8*,*9*),(*9*,*8*)] |)
(λ*e*. *2*)

definition *Dependability-norefl-offending-list*:: $'v$ *list-graph* $\Rightarrow$ ($'v \Rightarrow$ *dependability-level*) $\Rightarrow$ ($'v \times 'v$) *list list* **where**
Dependability-norefl-offending-list = *Generic-offending-list sinvar*

definition *NetModel-node-props P* = (λ *i*. (*case* (*node-properties P*) *i of Some property* $\Rightarrow$ *property* | *None* $\Rightarrow$ *SINVAR-Dependability-norefl.default-node-properties*))
lemma[*code-unfold*]: *SecurityInvariant.node-props SINVAR-Dependability-norefl.default-node-properties P* = *NetModel-node-props P*
⟨*proof*⟩

definition *Dependability-norefl-eval G P* = (*wf-list-graph G* ∧
sinvar G (*SecurityInvariant.node-props SINVAR-Dependability-norefl.default-node-properties P*))

lemma *sinvar-correct*: *wf-list-graph G* $\Longrightarrow$ *SINVAR-Dependability-norefl.sinvar* (*list-graph-to-graph G*) *nP* = *sinvar G nP*
⟨*proof*⟩

interpretation *Dependability-norefl-impl*:*TopoS-List-Impl*
where *default-node-properties=SINVAR-Dependability-norefl.default-node-properties*
and *sinvar-spec=SINVAR-Dependability-norefl.sinvar*
and *sinvar-impl=sinvar*
and *receiver-violation=SINVAR-Dependability-norefl.receiver-violation*
and *offending-flows-impl=Dependability-norefl-offending-list*
and *node-props-impl=NetModel-node-props*
and *eval-impl=Dependability-norefl-eval*
⟨*proof*⟩

6.12.2 packing

definition *SINVAR-LIB-Dependability-norefl* :: ($'v$::*vertex*, *SINVAR-Dependability-norefl.dependability-level*) *TopoS-packed* **where**
SINVAR-LIB-Dependability-norefl ≡
⦇ *nm-name* = ''*Dependability-norefl*'',
nm-receiver-violation = *SINVAR-Dependability-norefl.receiver-violation*,
nm-default = *SINVAR-Dependability-norefl.default-node-properties*,
nm-sinvar = *sinvar*,
nm-offending-flows = *Dependability-norefl-offending-list*,
nm-node-props = *NetModel-node-props*,
nm-eval = *Dependability-norefl-eval*
⦈
interpretation *SINVAR-LIB-Dependability-norefl-interpretation*: *TopoS-modelLibrary SINVAR-LIB-Dependability-no*
SINVAR-Dependability-norefl.sinvar
⟨*proof*⟩

hide-fact (**open**) *sinvar-correct*

hide-const (**open**) *sinvar NetModel-node-props*

end
theory *TopoS-Library*
imports
Lib/FiniteListGraph-Impl
Security-Invariants/SINVAR-BLPbasic-impl
Security-Invariants/SINVAR-Subnets-impl
Security-Invariants/SINVAR-DomainHierarchyNG-impl
Security-Invariants/SINVAR-BLPtrusted-impl
Security-Invariants/SINVAR-SecGwExt-impl
Security-Invariants/SINVAR-Sink-impl
Security-Invariants/SINVAR-SubnetsInGW-impl
Security-Invariants/SINVAR-CommunicationPartners-impl
Security-Invariants/SINVAR-NoRefl-impl
Security-Invariants/SINVAR-Tainting-impl
Security-Invariants/SINVAR-TaintingTrusted-impl

Security-Invariants/SINVAR-Dependability-impl
Security-Invariants/SINVAR-NonInterference-impl
Security-Invariants/SINVAR-ACLnotCommunicateWith-impl
Security-Invariants/SINVAR-ACLcommunicateWith-impl
Security-Invariants/SINVAR-Dependability-norefl-impl
Lib/Efficient-Distinct
HOL−Library.Code-Target-Nat
begin

end
theory *TopoS-Composition-Theory*
imports *TopoS-Interface TopoS-Helper*
begin

7 Composition Theory

Several invariants may apply to one policy.

The security invariants are all collected in a list. The list corresponds to the security requirements. The list should have the type (*'v graph* ⇒ *bool*) *list*, i.e. a list of predicates over the policy. We need in instantiated security invariant, i.e. get rid of *'a* and *'b*

record (*'v*) *SecurityInvariant-configured* =
c-sinvar::(*'v*) *graph* ⇒ *bool*
c-offending-flows::(*'v*) *graph* ⇒ (*'v* × *'v*) *set set*
c-isIFS::*bool*

— parameters 1-3 are the *SecurityInvariant*: *sinvar* ⊥ *receiver-violation*
Fourth parameter is the host attribute mapping *nP*
TODO: probably check *wf-graph* here and optionally some host-attribute sanity checker as in DomainHierachy.
fun *new-configured-SecurityInvariant* ::
(((*'v*::*vertex*) *graph* ⇒ (*'v* ⇒ *'a*) ⇒ *bool*) × *'a* × *bool* × (*'v* ⇒ *'a*)) ⇒ (*'v SecurityInvariant-configured*) *option* **where**
new-configured-SecurityInvariant (*sinvar*, *defbot*, *receiver-violation*, *nP*) =

(
if *SecurityInvariant sinvar defbot receiver-violation then*
Some ⦇
c-sinvar = (λ*G. sinvar G nP*),
c-offending-flows = (λ*G. SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G nP*),
c-isIFS = *receiver-violation*
⦈
else None
)

declare *new-configured-SecurityInvariant.simps*[*simp del*]

lemma *new-configured-TopoS-sinvar-correct*:
SecurityInvariant sinvar defbot receiver-violation $\Longrightarrow$
c-sinvar (*the* (*new-configured-SecurityInvariant* (*sinvar*, *defbot*, *receiver-violation*, *nP*))) = (λ*G. sinvar G nP*)
⟨*proof*⟩

lemma *new-configured-TopoS-offending-flows-correct*:
SecurityInvariant sinvar defbot receiver-violation $\Longrightarrow$
c-offending-flows (*the* (*new-configured-SecurityInvariant* (*sinvar*, *defbot*, *receiver-violation*, *nP*))) =

(λ*G. SecurityInvariant-withOffendingFlows.set-offending-flows sinvar G nP*)
⟨*proof*⟩

We now collect all the core properties of a security invariant, but without the *'a 'b* types, so it is instantiated with a concrete configuration.

locale *configured-SecurityInvariant* =
fixes *m* :: (*'v*::*vertex*) *SecurityInvariant-configured*
assumes
— As in SecurityInvariant definition
valid-c-offending-flows:
c-offending-flows m G = {*F. F* ⊆ (*edges G*) ∧ ¬ *c-sinvar m G* ∧ *c-sinvar m* (*delete-edges G F*) ∧ (∀ (*e1*, *e2*) ∈ *F.* ¬ *c-sinvar m* (*add-edge e1 e2* (*delete-edges G F*)))}
and
— A empty network can have no security violations
defined-offending:
⟦ *wf-graph* ⦇ *nodes* = *N*, *edges* = {} ⦈ ⟧ $\Longrightarrow$ *c-sinvar m* ⦇ *nodes* = *N*, *edges* = {}⦈
and
— prohibiting more does not decrease security
mono-sinvar:
⟦ *wf-graph* ⦇ *nodes* = *N*, *edges* = *E* ⦈; *E'* ⊆ *E*; *c-sinvar m* ⦇ *nodes* = *N*, *edges* = *E* ⦈ ⟧ $\Longrightarrow$
c-sinvar m ⦇ *nodes* = *N*, *edges* = *E'* ⦈
begin

lemma *sinvar-monoI*:
SecurityInvariant-withOffendingFlows.sinvar-mono (λ (*G*::(*'v*::*vertex*) *graph*) (*nP*::*'v* ⇒ *'a*). *c-sinvar m G*)
⟨*proof*⟩

if the network where nobody communicates with anyone fulfilles its security requirement, the offending flows are always defined.

lemma *defined-offending'*:

$\llbracket$ *wf-graph G*; ¬ *c-sinvar m G* $\rrbracket \Longrightarrow$ *c-offending-flows m G* ≠ {}
⟨*proof*⟩

lemma *subst-offending-flows*: $\bigwedge$ *nP*. *SecurityInvariant-withOffendingFlows.set-offending-flows* (λ*G nP*. *c-sinvar m G*) *G nP* = *c-offending-flows m G*
⟨*proof*⟩

all the *SecurityInvariant-preliminaries* stuff must hold, for an arbitrary *nP*

lemma *SecurityInvariant-preliminariesD*:
SecurityInvariant-preliminaries (λ (*G*::(′*v*::*vertex*) *graph*) (*nP*::′*v* ⇒ ′*a*). *c-sinvar m G*)
⟨*proof*⟩

lemma *negative-mono*:
$\bigwedge$ *N E′ E*. *wf-graph* (| *nodes* = *N*, *edges* = *E* |) $\Longrightarrow$
E′ ⊆ *E* $\Longrightarrow$ ¬ *c-sinvar m* (| *nodes* = *N*, *edges* = *E′* |) $\Longrightarrow$ ¬ *c-sinvar m* (| *nodes* = *N*, *edges* = *E* |)
⟨*proof*⟩

7.1 Reusing Lemmata

lemmas *mono-extend-set-offending-flows* =
SecurityInvariant-preliminaries.mono-extend-set-offending-flows[*OF SecurityInvariant-preliminariesD*, *simplified subst-offending-flows*]

$\llbracket$*wf-graph* (|*nodes* = *V*, *edges* = *E*|); *E′* ⊆ *E*; *F′* ∈ *c-offending-flows m* (|*nodes* = *V*, *edges* = *E′*|)$\rrbracket \Longrightarrow$ ∃ *F*∈*c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|). *F′* ⊆ *F*

lemmas *offending-flows-union-mono* =
SecurityInvariant-preliminaries.offending-flows-union-mono[*OF SecurityInvariant-preliminariesD*, *simplified subst-offending-flows*]

$\llbracket$*wf-graph* (|*nodes* = *V*, *edges* = *E*|); *E′* ⊆ *E*$\rrbracket \Longrightarrow \bigcup$ (*c-offending-flows m* (|*nodes* = *V*, *edges* = *E′*|)) ⊆ $\bigcup$ (*c-offending-flows m* (|*nodes* = *V*, *edges* = *E*|))

lemmas *sinvar-valid-remove-flattened-offending-flows* =
SecurityInvariant-preliminaries.sinvar-valid-remove-flattened-offending-flows[*OF SecurityInvariant-preliminariesD*, *simplified subst-offending-flows*]

wf-graph (|*nodes* = *nodesG*, *edges* = *edgesG*|) $\Longrightarrow$ *c-sinvar m* (|*nodes* = *nodesG*, *edges* = *edgesG* − $\bigcup$ (*c-offending-flows m* (|*nodes* = *nodesG*, *edges* = *edgesG*|))|)

lemmas *sinvar-valid-remove-SOME-offending-flows* =
SecurityInvariant-preliminaries.sinvar-valid-remove-SOME-offending-flows[*OF SecurityInvariant-preliminariesD*, *simplified subst-offending-flows*]

c-offending-flows m (|*nodes* = *nodesG*, *edges* = *edgesG*|) ≠ {} $\Longrightarrow$ *c-sinvar m* (|*nodes* = *nodesG*, *edges* = *edgesG* − (*SOME F*. *F* ∈ *c-offending-flows m* (|*nodes* = *nodesG*, *edges* = *edgesG*|))|)

lemmas *sinvar-valid-remove-minimalize-offending-overapprox* =
SecurityInvariant-preliminaries.sinvar-valid-remove-minimalize-offending-overapprox[*OF SecurityInvariant-preliminariesD*, *simplified subst-offending-flows*]

$\llbracket$*wf-graph* (|*nodes* = *nodesG*, *edges* = *edgesG*|); ¬ *c-sinvar m* (|*nodes* = *nodesG*, *edges* = *edgesG*|); *set Es* = *edgesG*; *distinct Es*$\rrbracket \Longrightarrow$ *c-sinvar m* (|*nodes* = *nodesG*, *edges* = *edgesG* −

set (*SecurityInvariant-withOffendingFlows.minimalize-offending-overapprox* (λ*G nP*. *c-sinvar m G*) *Es* [] ⦇*nodes* = *nodesG*, *edges* = *edgesG*⦈ *nP*))⦈

lemmas *empty-offending-contra* =
SecurityInvariant-withOffendingFlows.empty-offending-contra[**where** *sinvar*=(λ*G nP*. *c-sinvar m G*), *simplified subst-offending-flows*]

⟦*F* ∈ *c-offending-flows m G*; *F* = {}⟧ ⟹ *False*

lemmas *Un-set-offending-flows-bound-minus-subseteq* =
SecurityInvariant-preliminaries.Un-set-offending-flows-bound-minus-subseteq[*OF SecurityInvariant-preliminariesD*, *simplified subst-offending-flows*]

⟦*wf-graph* ⦇*nodes* = *V*, *edges* = *E*⦈; ⋃ (*c-offending-flows m* ⦇*nodes* = *V*, *edges* = *E*⦈) ⊆ *X*⟧ ⟹ ⋃ (*c-offending-flows m* ⦇*nodes* = *V*, *edges* = *E* − *E'*⦈) ⊆ *X* − *E'*

lemmas *Un-set-offending-flows-bound-minus-subseteq'* =
SecurityInvariant-preliminaries.Un-set-offending-flows-bound-minus-subseteq'[*OF SecurityInvariant-preliminariesD*, *simplified subst-offending-flows*]

⟦*wf-graph* ⦇*nodes* = *V*, *edges* = *E*⦈; ⋃ (*c-offending-flows m* ⦇*nodes* = *V*, *edges* = *E*⦈) ⊆ *X*⟧ ⟹ ⋃ (*c-offending-flows m* ⦇*nodes* = *V*, *edges* = *E* − *E'*⦈) ⊆ *X* − *E'*

end

thm *configured-SecurityInvariant-def*

configured-SecurityInvariant m ≡ (∀ *G*. *c-offending-flows m G* = {*F*. *F* ⊆ *edges G* ∧ ¬ *c-sinvar m G* ∧ *c-sinvar m* (*delete-edges G F*) ∧ (∀ (*e1*, *e2*)∈*F*. ¬ *c-sinvar m* (*add-edge e1 e2* (*delete-edges G F*)))}) ∧ (∀ *N*. *wf-graph* ⦇*nodes* = *N*, *edges* = {}⦈ ⟶ *c-sinvar m* ⦇*nodes* = *N*, *edges* = {}⦈) ∧ (∀ *N E E'*. *wf-graph* ⦇*nodes* = *N*, *edges* = *E*⦈ ⟶ *E'* ⊆ *E* ⟶ *c-sinvar m* ⦇*nodes* = *N*, *edges* = *E*⦈ ⟶ *c-sinvar m* ⦇*nodes* = *N*, *edges* = *E'*⦈)

thm *configured-SecurityInvariant.mono-sinvar*

⟦*configured-SecurityInvariant m*; *wf-graph* ⦇*nodes* = *N*, *edges* = *E*⦈; *E'* ⊆ *E*; *c-sinvar m* ⦇*nodes* = *N*, *edges* = *E*⦈⟧ ⟹ *c-sinvar m* ⦇*nodes* = *N*, *edges* = *E'*⦈

Naming convention: m :: network security requirement M :: network security requirement list

The function *new-configured-SecurityInvariant* takes some tuple and if it returns a result, the locale assumptions are automatically fulfilled.

theorem *new-configured-SecurityInvariant-sound*:
⟦ *new-configured-SecurityInvariant* (*sinvar*, *defbot*, *receiver-violation*, *nP*) = *Some m* ⟧ ⟹
configured-SecurityInvariant m
⟨*proof*⟩

All security invariants are valid according to the definition

definition *valid-reqs* :: (*'v*::*vertex*) *SecurityInvariant-configured list* ⇒ *bool* **where**
valid-reqs M ≡ ∀ *m* ∈ *set M*. *configured-SecurityInvariant m*

7.2 Algorithms

A (generic) security invariant corresponds to a type of security requirements (type: *'v graph* ⇒ (*'v* ⇒ *'a*) ⇒ *bool*). A configured security invariant is a security requirement in a scenario specific setting (type: *'v graph* ⇒ *bool*). I.e., it is a security requirement as listed in the requirements document. All security requirements are fulfilled for a fixed policy G if all security requirements are fulfilled for G.

Get all possible offending flows from all security requirements

definition *get-offending-flows* :: *'v SecurityInvariant-configured list* ⇒ *'v graph* ⇒ ((*'v* × *'v*) *set set*) **where**
get-offending-flows M G = (⋃*m*∈*set M*. *c-offending-flows m G*)

definition *all-security-requirements-fulfilled* :: (*'v*::*vertex*) *SecurityInvariant-configured list* ⇒ *'v graph* ⇒ *bool* **where**
all-security-requirements-fulfilled M G ≡ ∀ *m* ∈ *set M*. (*c-sinvar m*) *G*

Generate a valid topology from the security requirements

fun *generate-valid-topology* :: *'v SecurityInvariant-configured list* ⇒ *'v graph* ⇒ *'v graph* **where**
generate-valid-topology [] *G* = *G* |
generate-valid-topology (*m*#*Ms*) *G* = *delete-edges* (*generate-valid-topology Ms G*) (⋃ (*c-offending-flows m G*))

— return all Access Control Strategy models from a list of models
definition *get-ACS* :: (*'v*::*vertex*) *SecurityInvariant-configured list* ⇒ *'v SecurityInvariant-configured list* **where**
get-ACS M ≡ [*m* ← *M*. ¬ *c-isIFS m*]
— return all Information Flows Strategy models from a list of models
definition *get-IFS* :: (*'v*::*vertex*) *SecurityInvariant-configured list* ⇒ *'v SecurityInvariant-configured list* **where**
get-IFS M ≡ [*m* ← *M*. *c-isIFS m*]
lemma *get-ACS-union-get-IFS*: *set* (*get-ACS M*) ∪ *set* (*get-IFS M*) = *set M*
⟨*proof*⟩

7.3 Lemmata

lemma *valid-reqs1*: *valid-reqs* (*m* # *M*) ⟹ *configured-SecurityInvariant m*
⟨*proof*⟩
lemma *valid-reqs2*: *valid-reqs* (*m* # *M*) ⟹ *valid-reqs M*
⟨*proof*⟩
lemma *get-offending-flows-alt1*: *get-offending-flows M G* = ⋃ {*c-offending-flows m G* | *m*. *m* ∈ *set M*}
⟨*proof*⟩
lemma *get-offending-flows-un*: ⋃(*get-offending-flows M G*) = (⋃*m*∈*set M*. ⋃(*c-offending-flows m G*))
⟨*proof*⟩

lemma *all-security-requirements-fulfilled-mono*:
⟦ *valid-reqs M*; *E'* ⊆ *E*; *wf-graph* ⦇ *nodes* = *V*, *edges* = *E* ⦈ ⟧ ⟹
all-security-requirements-fulfilled M ⦇ *nodes* = *V*, *edges* = *E* ⦈ ⟹
all-security-requirements-fulfilled M ⦇ *nodes* = *V*, *edges* = *E'* ⦈
⟨*proof*⟩

7.4 generate valid topology

lemma *generate-valid-topology-nodes*:
nodes (*generate-valid-topology M G*) = (*nodes G*)

⟨*proof*⟩

lemma *generate-valid-topology-def-alt*:
generate-valid-topology M G = *delete-edges G* ($\bigcup$ (*get-offending-flows M G*))
⟨*proof*⟩

lemma *wf-graph-generate-valid-topology*: *wf-graph G* $\Longrightarrow$ *wf-graph* (*generate-valid-topology M G*)
⟨*proof*⟩

lemma *generate-valid-topology-mono-models*:
edges (*generate-valid-topology* (*m#M*) (| *nodes* = *V*, *edges* = *E* |)) ⊆ *edges* (*generate-valid-topology M* (| *nodes* = *V*, *edges* = *E* |))
⟨*proof*⟩

lemma *generate-valid-topology-subseteq-edges*:
edges (*generate-valid-topology M G*) ⊆ (*edges G*)
⟨*proof*⟩

generate-valid-topology generates a valid topology (Policy)!

theorem *generate-valid-topology-sound*:
⟦ *valid-reqs M*; *wf-graph* (|*nodes* = *V*, *edges* = *E*|) ⟧ $\Longrightarrow$
all-security-requirements-fulfilled M (*generate-valid-topology M* (|*nodes* = *V*, *edges* = *E*|))
⟨*proof*⟩

lemma *generate-valid-topology-as-set*:
generate-valid-topology M G = *delete-edges G* ($\bigcup$ *m* ∈ *set M*. ($\bigcup$ (*c-offending-flows m G*)))
⟨*proof*⟩

lemma *c-offending-flows-subseteq-edges*: *configured-SecurityInvariant m* $\Longrightarrow$ $\bigcup$(*c-offending-flows m G*) ⊆ *edges G*
⟨*proof*⟩

Does it also generate a maximum topology? It does, if the security invariants are in ENF-form. That means, if all security invariants can be expressed as a predicate over the edges, ∃ *P*. ∀ *G*. *c-sinvar m G* = (∀ (*v1*, *v2*)∈*edges G*. *P* (*v1*, *v2*))

definition *max-topo* :: (*'v*::*vertex*) *SecurityInvariant-configured list* ⇒ *'v graph* ⇒ *bool* **where**
max-topo M G ≡ *all-security-requirements-fulfilled M G* ∧ (
∀ (*v1*, *v2*) ∈ (*nodes G* × *nodes G*) − (*edges G*). ¬ *all-security-requirements-fulfilled M* (*add-edge v1 v2 G*))

lemma *unique-offending-obtain*:
assumes *m*: *configured-SecurityInvariant m* **and** *unique*: *c-offending-flows m G* = {*F*}
obtains *P* **where** *F* = {(*v1*, *v2*) ∈ *edges G*. ¬ *P* (*v1*, *v2*)} **and** *c-sinvar m G* = (∀ (*v1*,*v2*) ∈ *edges G*. *P* (*v1*, *v2*)) **and**
(∀ (*v1*,*v2*) ∈ *edges G* − *F*. *P* (*v1*, *v2*))
⟨*proof*⟩

lemma *enf-offending-flows*:
assumes *vm*: *configured-SecurityInvariant m* **and** *enf*: ∀ *G*. *c-sinvar m G* = (∀ *e* ∈ *edges G*. *P e*)
shows ∀ *G*. *c-offending-flows m G* = (*if c-sinvar m G then* {} *else* {{*e* ∈ *edges G*. ¬ *P e*}})
⟨*proof*⟩

lemma *enf-not-fulfilled-if-in-offending*:
assumes *validRs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *enf*: $\forall\, m \in set\ M.\ \exists\, P.\ \forall\, G.\ c\text{-}sinvar\ m\ G = (\forall\, e \in edges\ G.\ P\ e)$
shows $\forall\, x \in (\bigcup m{\in}set\ M.\ \bigcup(c\text{-}offending\text{-}flows\ m\ (fully\text{-}connected\ G))).$
$\neg\ all\text{-}security\text{-}requirements\text{-}fulfilled\ M\ (\!|\ nodes = V,\ edges = insert\ x\ E|\!)$
⟨*proof*⟩

theorem *generate-valid-topology-max-topo*: $[\![\ valid\text{-}reqs\ M;\ wf\text{-}graph\ G;$
$\forall\, m \in set\ M.\ \exists\, P.\ \forall\, G.\ c\text{-}sinvar\ m\ G = (\forall\, e \in edges\ G.\ P\ e)]\!] \Longrightarrow$
$max\text{-}topo\ M\ (generate\text{-}valid\text{-}topology\ M\ (fully\text{-}connected\ G))$
⟨*proof*⟩

lemma *enf-all-valid-policy-subset-of-max*:
assumes *validRs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *enf*: $\forall\, m \in set\ M.\ \exists\, P.\ \forall\, G.\ c\text{-}sinvar\ m\ G = (\forall\, e \in edges\ G.\ P\ e)$
and *nodesG'*: $nodes\ G = nodes\ G'$
shows $[\![\ wf\text{-}graph\ G';$
$all\text{-}security\text{-}requirements\text{-}fulfilled\ M\ G']\!] \Longrightarrow$
$edges\ G' \subseteq edges\ (generate\text{-}valid\text{-}topology\ M\ (fully\text{-}connected\ G))$
⟨*proof*⟩

7.5 More Lemmata

lemma (**in** *configured-SecurityInvariant*) *c-sinvar-valid-imp-no-offending-flows*:
$c\text{-}sinvar\ m\ G \Longrightarrow c\text{-}offending\text{-}flows\ m\ G = \{\}$
⟨*proof*⟩

lemma *all-security-requirements-fulfilled-imp-no-offending-flows*:
$valid\text{-}reqs\ M \Longrightarrow all\text{-}security\text{-}requirements\text{-}fulfilled\ M\ G \Longrightarrow (\bigcup m{\in}set\ M.\ \bigcup(c\text{-}offending\text{-}flows\ m\ G)) = \{\}$
⟨*proof*⟩

corollary *all-security-requirements-fulfilled-imp-get-offending-empty*:
$valid\text{-}reqs\ M \Longrightarrow all\text{-}security\text{-}requirements\text{-}fulfilled\ M\ G \Longrightarrow get\text{-}offending\text{-}flows\ M\ G = \{\}$
⟨*proof*⟩

corollary *generate-valid-topology-does-nothing-if-valid*:
$[\![\ valid\text{-}reqs\ M;\ all\text{-}security\text{-}requirements\text{-}fulfilled\ M\ G]\!] \Longrightarrow$
$generate\text{-}valid\text{-}topology\ M\ G = G$
⟨*proof*⟩

lemma *mono-extend-get-offending-flows*: $[\![\ valid\text{-}reqs\ M;$
$wf\text{-}graph\ (\!|nodes = V,\ edges = E|\!);$
$E' \subseteq E;$
$F' \in get\text{-}offending\text{-}flows\ M\ (\!|nodes = V,\ edges = E'|\!)\ ]\!] \Longrightarrow$
$\exists\, F{\in}get\text{-}offending\text{-}flows\ M\ (\!|nodes = V,\ edges = E|\!).\ F' \subseteq F$
⟨*proof*⟩

lemma *get-offending-flows-subseteq-edges*: $valid\text{-}reqs\ M \Longrightarrow F \in get\text{-}offending\text{-}flows\ M\ (\!|nodes = V,\ edges = E|\!) \Longrightarrow F \subseteq E$

⟨proof⟩

thm *configured-SecurityInvariant.offending-flows-union-mono*
lemma *get-offending-flows-union-mono*: $\llbracket$*valid-reqs M*;
wf-graph (|*nodes* = *V*, *edges* = *E*|); $E' \subseteq E$ $\rrbracket \Longrightarrow$
$\bigcup$(*get-offending-flows M* (|*nodes* = *V*, *edges* = *E'*|)) $\subseteq$ $\bigcup$(*get-offending-flows M* (|*nodes* = *V*, *edges* = *E*|))
⟨proof⟩

thm *configured-SecurityInvariant.Un-set-offending-flows-bound-minus-subseteq'*
lemma *Un-set-offending-flows-bound-minus-subseteq'*:$\llbracket$*valid-reqs M*;
wf-graph (|*nodes* = *V*, *edges* = *E*|); $E' \subseteq E$;
$\bigcup$(*get-offending-flows M* (|*nodes* = *V*, *edges* = *E*|)) $\subseteq X$ $\rrbracket \Longrightarrow$ $\bigcup$(*get-offending-flows M* (|*nodes* = *V*, *edges* = *E* − *E'*|)) $\subseteq X - E'$
⟨proof⟩

lemma *ENF-uniquely-defined-offedning*: *valid-reqs M* $\Longrightarrow$ *wf-graph G* $\Longrightarrow$
$\forall$ *m* $\in$ *set M*. $\exists$ *P*. $\forall$ *G*. *c-sinvar m G* = ($\forall$ *e* $\in$ *edges G*. *P e*) $\Longrightarrow$
$\forall$ *m* $\in$ *set M*. $\forall$ *G*. $\neg$ *c-sinvar m G* $\longrightarrow$ ($\exists$ *OFF*. *c-offending-flows m G* = {*OFF*})
⟨proof⟩

lemma assumes *configured-SecurityInvariant m*
and $\forall$ *G*. $\neg$ *c-sinvar m G* $\longrightarrow$ ($\exists$ *OFF*. *c-offending-flows m G* = {*OFF*})
shows $\exists$ *OFF-P*. $\forall$ *G*. *c-offending-flows m G* = (*if c-sinvar m G then* {} *else* {*OFF-P G*})
⟨proof⟩

Hilber's eps operator example

lemma (*SOME x. x* : {*1*::*nat*, *2*, *3*}) = *x* $\Longrightarrow$ *x* = *1* $\vee$ *x* = *2* $\vee$ *x* = *3*
⟨proof⟩

Only removing one offending flow should be enough

fun *generate-valid-topology-SOME* :: *'v SecurityInvariant-configured list* $\Rightarrow$ *'v graph* $\Rightarrow$ *'v graph*
where
generate-valid-topology-SOME [] *G* = *G* |
generate-valid-topology-SOME (*m#Ms*) *G* = (*if c-sinvar m G*
then generate-valid-topology-SOME Ms G
else delete-edges (*generate-valid-topology-SOME Ms G*) (*SOME F*. *F* $\in$ *c-offending-flows m G*)
)

lemma *generate-valid-topology-SOME-nodes*: *nodes* (*generate-valid-topology-SOME M* (|*nodes* = *V*, *edges* = *E*|)) = *V*
⟨proof⟩

theorem *generate-valid-topology-SOME-sound*:
$\llbracket$ *valid-reqs M*; *wf-graph* (|*nodes* = *V*, *edges* = *E*|) $\rrbracket \Longrightarrow$
all-security-requirements-fulfilled M (*generate-valid-topology-SOME M* (|*nodes* = *V*, *edges* = *E*|))
⟨proof⟩

lemma *generate-valid-topology-SOME-def-alt*:

generate-valid-topology-SOME M G = delete-edges G ($\bigcup$ *m* ∈ *set M. if c-sinvar m G then* {} *else* (*SOME F. F* ∈ *c-offending-flows m G*))
⟨*proof*⟩

lemma *generate-valid-topology-SOME-superset*:
⟦ *valid-reqs M*; *wf-graph G* ⟧ ⟹
edges (*generate-valid-topology M G*) ⊆ *edges* (*generate-valid-topology-SOME M G*)
⟨*proof*⟩

Notation: *generate-valid-topology-SOME*: non-deterministic choice *generate-valid-topology-some*: executable which selects always the same

fun *generate-valid-topology-some* :: *'v SecurityInvariant-configured list* ⇒ (*'v*×*'v*) *list* ⇒ *'v graph* ⇒ *'v graph* **where**
generate-valid-topology-some [] - *G* = *G* |
generate-valid-topology-some (*m#Ms*) *Es G* = (*if c-sinvar m G*
then generate-valid-topology-some Ms Es G
else delete-edges (*generate-valid-topology-some Ms Es G*) (*set* (*minimalize-offending-overapprox* (*c-sinvar m*) *Es* [] *G*))
)

theorem *generate-valid-topology-some-sound*:
⟦ *valid-reqs M*; *wf-graph* (|*nodes* = *V*, *edges* = *E*|); *set Es* = *E*; *distinct Es* ⟧ ⟹
all-security-requirements-fulfilled M (*generate-valid-topology-some M Es* (|*nodes* = *V*, *edges* = *E*|))
⟨*proof*⟩

end
theory *TopoS-Stateful-Policy*
imports *TopoS-Composition-Theory*
begin

8 Stateful Policy

Details described in [1].

Algorithm

term *TopoS-Composition-Theory.generate-valid-topology*

generates a valid high-level topology. Now we discuss how to turn this into a stateful policy.

Example: SensorNode produces data and has no security level. SensorSink has high security level SensorNode -> SensorSink, but not the other way round. Implementation: UDP in one direction

Alice is in internal protected subnet. Google can not arbitrarily access Alice. Alice sends requests to google. It is desirable that Alice gets the response back Implementation: TCP and stateful packet filter that allows, once Alice establishes a connection, to get a response back via this connection.

Result: IFS violations undesirable. ACS violations may be okay under certain conditions.

term *all-security-requirements-fulfilled*

$G = (V, E_{fix}, E_{state})$

record *'v stateful-policy* =
  hosts :: *'v set* — nodes, vertices
  flows-fix :: (*'v* ×*'v*) *set* — edges in high-level policy
  flows-state :: (*'v* ×*'v*) *set* — edges that can have stateful flows, i.e. backflows

All the possible ways packets can travel in a *'v stateful-policy*. They can either choose the fixed links; Or use a stateful link, i.e. establish state. Once state is established, packets can flow back via the established link.

definition *all-flows* :: *'v stateful-policy* ⇒ (*'v* × *'v*) *set* **where**
  all-flows $\mathcal{T}$ ≡ *flows-fix* $\mathcal{T}$ ∪ *flows-state* $\mathcal{T}$ ∪ *backflows* (*flows-state* $\mathcal{T}$)

definition *stateful-policy-to-network-graph* :: *'v stateful-policy* ⇒ *'v graph* **where**
  stateful-policy-to-network-graph $\mathcal{T}$ = (| *nodes* = *hosts* $\mathcal{T}$, *edges* = *all-flows* $\mathcal{T}$ |)

'v stateful-policy syntactically well-formed

locale *wf-stateful-policy* =
  fixes $\mathcal{T}$ :: *'v stateful-policy*
  assumes *E-wf*: *fst* ' (*flows-fix* $\mathcal{T}$) ⊆ (*hosts* $\mathcal{T}$)
        snd ' (*flows-fix* $\mathcal{T}$) ⊆ (*hosts* $\mathcal{T}$)
  and *E-state-fix*: *flows-state* $\mathcal{T}$ ⊆ *flows-fix* $\mathcal{T}$
  and *finite-Hosts*: *finite* (*hosts* $\mathcal{T}$)
begin

  lemma *E-wfD*: **assumes** (*v*,*v'*) ∈ *flows-fix* $\mathcal{T}$
    shows *v* ∈ *hosts* $\mathcal{T}$ *v'* ∈ *hosts* $\mathcal{T}$
    ⟨*proof*⟩

  lemma *E-state-valid*: *fst* ' (*flows-state* $\mathcal{T}$) ⊆ (*hosts* $\mathcal{T}$)
        snd ' (*flows-state* $\mathcal{T}$) ⊆ (*hosts* $\mathcal{T}$)
  ⟨*proof*⟩

  lemma *E-state-validD*: **assumes** (*v*,*v'*) ∈ *flows-state* $\mathcal{T}$
    shows *v* ∈ *hosts* $\mathcal{T}$ *v'* ∈ *hosts* $\mathcal{T}$
    ⟨*proof*⟩

  lemma *finite-fix*: *finite* (*flows-fix* $\mathcal{T}$)
  ⟨*proof*⟩

  lemma *finite-state*: *finite* (*flows-state* $\mathcal{T}$)
    ⟨*proof*⟩

  lemma *finite-backflows-state*: *finite* (*backflows* (*flows-state* $\mathcal{T}$))
    ⟨*proof*⟩

  lemma *E-state-backflows-wf*: *fst* ' *backflows* (*flows-state* $\mathcal{T}$) ⊆ (*hosts* $\mathcal{T}$)
        snd ' *backflows* (*flows-state* $\mathcal{T}$) ⊆ (*hosts* $\mathcal{T}$)
    ⟨*proof*⟩

end

Minimizing stateful flows such that only newly added backflows remain

  definition *filternew-flows-state* :: *'v stateful-policy* ⇒ (*'v* × *'v*) *set* **where**

filternew-flows-state $\mathcal{T} \equiv \{(s, r) \in$ *flows-state* $\mathcal{T}.\ (r, s) \notin$ *flows-fix* $\mathcal{T}\}$

lemma *filternew-subseteq-flows-state*: *filternew-flows-state* $\mathcal{T} \subseteq$ *flows-state* $\mathcal{T}$
⟨*proof*⟩
lemma *filternew-flows-state-alt*: *filternew-flows-state* $\mathcal{T}$ = *flows-state* $\mathcal{T}$ − (*backflows* (*flows-fix* $\mathcal{T}$))
⟨*proof*⟩
lemma *filternew-flows-state-alt2*: *filternew-flows-state* $\mathcal{T}$ = $\{e \in$ *flows-state* $\mathcal{T}.\ e \notin$ *backflows* (*flows-fix* $\mathcal{T}$)$\}$
⟨*proof*⟩
lemma *backflows-filternew-flows-state*: *backflows* (*filternew-flows-state* $\mathcal{T}$) = (*backflows* (*flows-state* $\mathcal{T}$)) − (*flows-fix* $\mathcal{T}$)
⟨*proof*⟩

lemma *stateful-policy-to-network-graph-filternew*: ⟦ *wf-stateful-policy* $\mathcal{T}$ ⟧ $\Longrightarrow$
stateful-policy-to-network-graph $\mathcal{T}$ =
stateful-policy-to-network-graph (|*hosts* = *hosts* $\mathcal{T}$, *flows-fix* = *flows-fix* $\mathcal{T}$, *flows-state* = *filternew-flows-state* $\mathcal{T}$ |)
⟨*proof*⟩

lemma *backflows-filternew-disjunct-flows-fix*:
$\forall\ b \in$ (*backflows* (*filternew-flows-state* $\mathcal{T}$)). $b \notin$ *flows-fix* $\mathcal{T}$
⟨*proof*⟩

Given a high-level policy, we can construct a pretty large syntactically valid low level policy. However, the stateful policy will almost certainly violate security requirements!

lemma *wf-graph G* $\Longrightarrow$ *wf-stateful-policy* (| *hosts* = *nodes G*, *flows-fix* = *nodes G* × *nodes G*, *flows-state* = *nodes G* × *nodes G* |)
⟨*proof*⟩

wf-stateful-policy implies *wf-graph*

lemma *wf-stateful-policy-is-wf-graph*: *wf-stateful-policy* $\mathcal{T}$ $\Longrightarrow$ *wf-graph* (|*nodes* = *hosts* $\mathcal{T}$, *edges* = *all-flows* $\mathcal{T}$|)
⟨*proof*⟩

lemma ($\forall F \in$ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$). $F \subseteq$ *backflows* (*filternew-flows-state* $\mathcal{T}$)) $\longleftrightarrow$
$\bigcup$(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$)) $\subseteq$ (*backflows* (*flows-state* $\mathcal{T}$)) − (*flows-fix* $\mathcal{T}$)
⟨*proof*⟩

When is a stateful policy $\mathcal{T}$ compliant with a high-level policy G and the security requirements M?

locale *stateful-policy-compliance* =
fixes $\mathcal{T}$ :: (*'v::vertex*) *stateful-policy*
fixes G :: *'v graph*
fixes M :: (*'v*) *SecurityInvariant-configured list*
assumes
— the graph must be syntactically valid
wfG: *wf-graph G*
and
— security requirements must be valid

validReqs: *valid-reqs M*
and
— the high-level policy must be valid
high-level-policy-valid: *all-security-requirements-fulfilled M G*
and
— the stateful policy must be syntactically valid
stateful-policy-wf:
wf-stateful-policy $\mathcal{T}$
and
— the stateful policy must talk about the same nodes as the high-level policy
hosts-nodes:
hosts $\mathcal{T}$ = *nodes G*
and
— only flows that are allowed in the high-level policy are allowed in the stateful policy
flows-edges:
flows-fix $\mathcal{T} \subseteq$ *edges G*
and
— the low level policy must comply with the high-level policy
— all information flow strategy requirements must be fulfilled, i.e. no leaks!
compliant-stateful-IFS:
all-security-requirements-fulfilled (*get-IFS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$)
and
— No Access Control side effects must occur
compliant-stateful-ACS:
$\forall F \in$ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$). $F \subseteq$ *backflows* (*filternew-flows-state* $\mathcal{T}$)

begin
lemma *compliant-stateful-ACS-no-side-effects-filternew-helper*:
$\forall$ $E \subseteq$ *backflows* (*filternew-flows-state* $\mathcal{T}$). $\forall$ $F \in$ *get-offending-flows* (*get-ACS M*) (| *nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T} \cup E$ |). $F \subseteq E$
⟨*proof*⟩

theorem *compliant-stateful-ACS-no-side-effects*:
$\forall$ $E \subseteq$ *backflows* (*flows-state* $\mathcal{T}$). $\forall$ $F \in$ *get-offending-flows*(*get-ACS M*) (| *nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T} \cup E$ |). $F \subseteq E$
⟨*proof*⟩

corollary *compliant-stateful-ACS-no-side-effects'*: $\forall$ $E \subseteq$ *backflows* (*flows-state* $\mathcal{T}$). $\forall$ $F \in$ *get-offending-flows*(*get-ACS M*) (| *nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T} \cup$ *flows-state* $\mathcal{T} \cup E$ |). $F \subseteq E$
⟨*proof*⟩

The high level graph generated from the low level policy is a valid graph

lemma *valid-stateful-policy*: *wf-graph* (|*nodes* = *hosts* $\mathcal{T}$, *edges* = *all-flows* $\mathcal{T}$|)
⟨*proof*⟩

The security requirements are definitely fulfilled if we consider only the fixed flows and the normal direction of the stateful flows (i.e. no backflows). I.e. considering no states, everything must be fulfilled

lemma *compliant-stateful-ACS-static-valid*: *all-security-requirements-fulfilled* (*get-ACS M*) (| *nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ |)
⟨*proof*⟩

theorem *compliant-stateful-ACS-static-valid'*:
all-security-requirements-fulfilled M ⦇ *nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ ∪ *flows-state* $\mathcal{T}$ ⦈
⟨*proof*⟩

The flows with state are a subset of the flows allowed by the policy

theorem *flows-state-edges*: *flows-state* $\mathcal{T}$ ⊆ *edges G*
⟨*proof*⟩

All offending flows are subsets of the reveres stateful flows

lemma *compliant-stateful-ACS-only-state-violations*:
∀ *F* ∈ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$). *F* ⊆ *backflows* (*flows-state* $\mathcal{T}$)
⟨*proof*⟩

theorem *compliant-stateful-ACS-only-state-violations'*: ∀ *F* ∈ *get-offending-flows M* (*stateful-policy-to-network-graph* $\mathcal{T}$). *F* ⊆ *backflows* (*flows-state* $\mathcal{T}$)
⟨*proof*⟩

All violations are backflows of valid flows

corollary *compliant-stateful-ACS-only-state-violations-union*: ⋃(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$)) ⊆ *backflows* (*flows-state* $\mathcal{T}$)
⟨*proof*⟩

corollary *compliant-stateful-ACS-only-state-violations-union'*: ⋃(*get-offending-flows M* (*stateful-policy-to-network-gr* $\mathcal{T}$)) ⊆ *backflows* (*flows-state* $\mathcal{T}$)
⟨*proof*⟩

All individual flows cause no side effects, i.e. each backflow causes at most itself as violation, no other side-effect violations are induced.

lemma *compliant-stateful-ACS-no-state-singleflow-side-effect*:
∀ (v_1, v_2) ∈ *backflows* (*flows-state* $\mathcal{T}$).
⋃(*get-offending-flows*(*get-ACS M*) ⦇ *nodes* = *hosts* $\mathcal{T}$, *edges* = *flows-fix* $\mathcal{T}$ ∪ *flows-state* $\mathcal{T}$ ∪ {(v_1, v_2)} ⦈) ⊆ {(v_1, v_2)}
⟨*proof*⟩
end

8.1 Summarizing the important theorems

No information flow security requirements are violated (including all added stateful flows)

thm *stateful-policy-compliance.compliant-stateful-IFS*

There are not access control side effects when allowing stateful backflows. I.e. for all possible subsets of the to-allow backflows, the violations they cause are only these backflows themselves

thm *stateful-policy-compliance.compliant-stateful-ACS-no-side-effects'*

Also, considering all backflows individually, they cause no side effect, i.e. the only violation added is the backflow itself

thm *stateful-policy-compliance.compliant-stateful-ACS-no-state-singleflow-side-effect*

In particular, all introduced offending flows for access control strategies are at most the stateful backflows

thm *stateful-policy-compliance.compliant-stateful-ACS-only-state-violations-union*

Which implies: all introduced offending flows are at most the stateful backflows

thm *stateful-policy-compliance.compliant-stateful-ACS-only-state-violations-union′*

Disregarding the backflows of stateful flows, all security requirements are fulfilled.

thm *stateful-policy-compliance.compliant-stateful-ACS-static-valid′*

end
theory *TopoS-Composition-Theory-impl*
imports *TopoS-Interface-impl TopoS-Composition-Theory*
begin

9 Composition Theory – List Implementation

Several invariants may apply to one policy.

term *X*::(′*v*::*vertex*, ′*a*) *TopoS-packed*

9.1 Generating instantiated (configured) network security invariants

record (′*v*) *SecurityInvariant* =
implc-type :: *string*
implc-description :: *string*
implc-sinvar ::(′*v*) *list-graph* ⇒ *bool*
implc-offending-flows ::(′*v*) *list-graph* ⇒ (′*v* × ′*v*) *list list*
implc-isIFS :: *bool*

Test if this definition is compliant with the formal definition on sets.

definition *SecurityInvariant-complies-formal-def* ::
(′*v*) *SecurityInvariant* ⇒ ′*v TopoS-Composition-Theory.SecurityInvariant-configured* ⇒ *bool* **where**
SecurityInvariant-complies-formal-def impl spec ≡
(∀ *G*. *wf-list-graph G* ⟶ *implc-sinvar impl G* = *c-sinvar spec* (*list-graph-to-graph G*)) ∧
(∀ *G*. *wf-list-graph G* ⟶ *set*‘*set* (*implc-offending-flows impl G*) = *c-offending-flows spec* (*list-graph-to-graph G*)) ∧
(*implc-isIFS impl* = *c-isIFS spec*)

fun *new-configured-list-SecurityInvariant* ::
(′*v*::*vertex*, ′*a*) *TopoS-packed* ⇒ (′*v*::*vertex*, ′*a*) *TopoS-Params* ⇒ *string* ⇒
(′*v SecurityInvariant*) **where**
new-configured-list-SecurityInvariant m C description =
(*let nP* = *nm-node-props m C in*
⦇
implc-type = *nm-name m*,
implc-description = *description*,
implc-sinvar = (λ*G*. (*nm-sinvar m*) *G nP*),
implc-offending-flows = (λ*G*. (*nm-offending-flows m*) *G nP*),
implc-isIFS = *nm-receiver-violation m*
⦈)

the *new-configured-SecurityInvariant* must give a result if we have the SecurityInvariant modelLibrary

lemma *TopoS-modelLibrary-yields-new-configured-SecurityInvariant*:
assumes *NetModelLib*: *TopoS-modelLibrary m sinvar-spec*
and *nPdef*: *nP = nm-node-props m C*
and *formalSpec*: *Spec* = ⦇
c-sinvar = (λ*G*. *sinvar-spec G nP*),
c-offending-flows = (λ*G*. *SecurityInvariant-withOffendingFlows.set-offending-flows sinvar-spec G nP*),
c-isIFS = *nm-receiver-violation m*
⦈
shows *new-configured-SecurityInvariant* (*sinvar-spec*, *nm-default m*, *nm-receiver-violation m*, *nP*) = *Some Spec*
⟨*proof*⟩
thm *TopoS-modelLibrary-yields-new-configured-SecurityInvariant*[*simplified*]

lemma *new-configured-list-SecurityInvariant-complies*:
assumes *NetModelLib*: *TopoS-modelLibrary m sinvar-spec*
and *nPdef*: *nP = nm-node-props m C*
and *formalSpec*: *Spec* = *new-configured-SecurityInvariant* (*sinvar-spec*, *nm-default m*, *nm-receiver-violation m*, *nP*)
and *implSpec*: *Impl* = *new-configured-list-SecurityInvariant m C description*
shows *SecurityInvariant-complies-formal-def Impl* (*the Spec*)
⟨*proof*⟩

corollary *new-configured-list-SecurityInvariant-complies'*:
⟦ *TopoS-modelLibrary m sinvar-spec* ⟧ ⟹
SecurityInvariant-complies-formal-def (*new-configured-list-SecurityInvariant m C description*)
(*the* (*new-configured-SecurityInvariant* (*sinvar-spec*, *nm-default m*, *nm-receiver-violation m*, *nm-node-props m C*)))
⟨*proof*⟩
thm *new-configured-SecurityInvariant-sound*
— we get that *new-configured-list-SecurityInvariant* has all the necessary properties (modulo *SecurityInvariant-complies-formal-def*)

9.2 About security invariants

specification and implementation comply.

type-synonym *'v security-models-spec-impl*=(*'v SecurityInvariant* × *'v TopoS-Composition-Theory.SecurityInvariant-*
list

definition *get-spec* :: *'v security-models-spec-impl* ⇒ (*'v TopoS-Composition-Theory.SecurityInvariant-configured*) *list* **where**
get-spec M ≡ [*snd m*. *m* ← *M*]
definition *get-impl* :: *'v security-models-spec-impl* ⇒ (*'v SecurityInvariant*) *list* **where**
get-impl M ≡ [*fst m*. *m* ← *M*]

9.3 Calculating offending flows

fun *implc-get-offending-flows* :: (*'v*) *SecurityInvariant list* ⇒ *'v list-graph* ⇒ ((*'v* × *'v*) *list list*) **where**
implc-get-offending-flows [] *G* = [] |

implc-get-offending-flows (*m*#*Ms*) *G* = (*implc-offending-flows m G*)@(*implc-get-offending-flows Ms G*)

lemma *implc-get-offending-flows-fold*:
implc-get-offending-flows M G = *fold* (λ*m accu*. *accu*@(*implc-offending-flows m G*)) *M* []
⟨*proof*⟩

lemma *implc-get-offending-flows-Un*: *set*'*set* (*implc-get-offending-flows M G*) = (⋃*m*∈*set M*. *set*'*set* (*implc-offending-flows m G*))
⟨*proof*⟩

lemma *implc-get-offending-flows-map-concat*: (*implc-get-offending-flows M G*) = *concat* [*implc-offending-flows m G*. *m* ← *M*]
⟨*proof*⟩

theorem *implc-get-offending-flows-complies*:
assumes *a1*: ∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*
and *a2*: *wf-list-graph G*
shows *set*'*set* (*implc-get-offending-flows* (*get-impl M*) *G*) = (*get-offending-flows* (*get-spec M*) (*list-graph-to-graph G*))
⟨*proof*⟩

9.4 Accessors

definition *get-IFS* :: '*v SecurityInvariant list* ⇒ '*v SecurityInvariant list* **where**
get-IFS M ≡ [*m* ← *M*. *implc-isIFS m*]
definition *get-ACS* :: '*v SecurityInvariant list* ⇒ '*v SecurityInvariant list* **where**
get-ACS M ≡ [*m* ← *M*. ¬ *implc-isIFS m*]

lemma *get-IFS-get-ACS-complies*:
assumes *a*: ∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*
shows ∀ (*m-impl*, *m-spec*) ∈ *set* (*zip* (*get-IFS* (*get-impl M*)) (*TopoS-Composition-Theory.get-IFS* (*get-spec M*))).
SecurityInvariant-complies-formal-def m-impl m-spec
and ∀ (*m-impl*, *m-spec*) ∈ *set* (*zip* (*get-ACS* (*get-impl M*)) (*TopoS-Composition-Theory.get-ACS* (*get-spec M*))).
SecurityInvariant-complies-formal-def m-impl m-spec
⟨*proof*⟩

lemma *get-IFS-get-ACS-select-simps*:
assumes *a1*: ∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*
shows ∀ (*m-impl*, *m-spec*) ∈ *set* (*zip* (*get-IFS* (*get-impl M*)) (*TopoS-Composition-Theory.get-IFS* (*get-spec M*))). *SecurityInvariant-complies-formal-def m-impl m-spec* (**is** ∀ (*m-impl*, *m-spec*) ∈ *set ?zippedIFS*. *SecurityInvariant-complies-formal-def m-impl m-spec*)
and (*get-impl* (*zip* (*TopoS-Composition-Theory-impl.get-IFS* (*get-impl M*)) (*TopoS-Composition-Theory.get-IFS* (*get-spec M*)))) = *TopoS-Composition-Theory-impl.get-IFS* (*get-impl M*)
and (*get-spec* (*zip* (*TopoS-Composition-Theory-impl.get-IFS* (*get-impl M*)) (*TopoS-Composition-Theory.get-IFS* (*get-spec M*)))) = *TopoS-Composition-Theory.get-IFS* (*get-spec M*)
and ∀ (*m-impl*, *m-spec*) ∈ *set* (*zip* (*get-ACS* (*get-impl M*)) (*TopoS-Composition-Theory.get-ACS*

(*get-spec M*))). *SecurityInvariant-complies-formal-def m-impl m-spec* (**is** ∀ (*m-impl*, *m-spec*) ∈ *set ?zippedACS*. *SecurityInvariant-complies-formal-def m-impl m-spec*)
and (*get-impl* (*zip* (*TopoS-Composition-Theory-impl.get-ACS* (*get-impl M*)) (*TopoS-Composition-Theory.get-ACS* (*get-spec M*)))) = *TopoS-Composition-Theory-impl.get-ACS* (*get-impl M*)
and (*get-spec* (*zip* (*TopoS-Composition-Theory-impl.get-ACS* (*get-impl M*)) (*TopoS-Composition-Theory.get-ACS* (*get-spec M*)))) = *TopoS-Composition-Theory.get-ACS* (*get-spec M*)
⟨*proof*⟩

thm *get-IFS-get-ACS-select-simps*

9.5 All security requirements fulfilled

definition *all-security-requirements-fulfilled* :: *'v SecurityInvariant list* ⇒ *'v list-graph* ⇒ *bool* **where**
all-security-requirements-fulfilled M G ≡ ∀ *m* ∈ *set M*. (*implc-sinvar m*) *G*

lemma *all-security-requirements-fulfilled-complies*:
⟦ ∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*;
wf-list-graph (*G*::(*'v*::*vertex*) *list-graph*) ⟧ ⟹
all-security-requirements-fulfilled (*get-impl M*) *G* ⟷ *TopoS-Composition-Theory.all-security-requirements-fulfilled* (*get-spec M*) (*list-graph-to-graph G*)
⟨*proof*⟩

9.6 generate valid topology

value *concat* [[*1*::*int*,*2*,*3*], [*4*,*6*,*5*]]

fun *generate-valid-topology* :: *'v SecurityInvariant list* ⇒ *'v list-graph* ⇒ (*'v list-graph*) **where**
generate-valid-topology M G = *delete-edges G* (*concat* (*implc-get-offending-flows M G*))

lemma *generate-valid-topology-complies*:
⟦ ∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*;
wf-list-graph (*G*::(*'v list-graph*)) ⟧ ⟹
list-graph-to-graph (*generate-valid-topology* (*get-impl M*) *G*) =
TopoS-Composition-Theory.generate-valid-topology (*get-spec M*) (*list-graph-to-graph G*)
⟨*proof*⟩

9.7 generate valid topology

tuned for invariants where we don't want to calculate all offending flows

Theoretic foundations: The algorithm *generate-valid-topology-SOME* picks ONE offending flow non-deterministically. This is sound: ⟦*valid-reqs ?M*; *wf-graph* ⦇*nodes* = *?V*, *edges* = *?E*⦈⟧ ⟹ *TopoS-Composition-Theory.all-security-requirements-fulfilled ?M* (*generate-valid-topology-SOME ?M* ⦇*nodes* = *?V*, *edges* = *?E*⦈). However, this non-deterministic choice is hard to implement. To pick one offending flow deterministically, we have implemented *TopoS-Interface-impl.minimalize-offending-o* It gives back one offending flow: ⟦*SecurityInvariant-preliminaries ?sinvar*; *wf-graph ?G*; *SecurityInvariant-withOffendingFlows.is-offending-flows ?sinvar* (*set ?ff*) *?G ?nP*; *set ?ff* ⊆ *edges ?G*; *distinct ?ff*⟧ ⟹ *set* (*SecurityInvariant-withOffendingFlows.minimalize-offending-overapprox ?sinvar ?ff* [] *?G ?nP*) ∈ *SecurityInvariant-withOffendingFlows.set-offending-flows ?sinvar ?G ?nP* The good thing about this function is, that it does not need to construct the complete *SecurityInvariant-withOffendingFlows.set-offending-flows*. Therefore, it can be used for security invariants which may have an exponential number of offending flows. The corresponding algorithm that uses this function is *generate-valid-topology-some*. It is also sound: ⟦*valid-reqs ?M*; *wf-graph* (⦇*nodes* = *?V*, *edges* = *?E*⦈); *set ?Es* = *?E*; *distinct ?Es*⟧ ⟹ *TopoS-Composition-Theory.all-security-requirements-fulfilled ?M* (*generate-valid-topology-some ?M ?Es* (⦇*nodes* = *?V*, *edges* = *?E*⦈)).

fun *generate-valid-topology-some* :: *′v SecurityInvariant list* ⇒ *′v list-graph* ⇒ (*′v list-graph*) **where**
generate-valid-topology-some [] *G* = *G* |
generate-valid-topology-some (*m*#*Ms*) *G* = (*if implc-sinvar m G*
then generate-valid-topology-some Ms G
else delete-edges (*generate-valid-topology-some Ms G*) (*minimalize-offending-overapprox* (*implc-sinvar m*) (*edgesL G*) [] *G*)
)

thm *TopoS-Composition-Theory.generate-valid-topology-some-sound*

lemma *generate-valid-topology-some-complies*:
⟦ ∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*;
wf-list-graph (*G*::(*′v*::*vertex list-graph*)) ⟧ ⟹
list-graph-to-graph (*generate-valid-topology-some* (*get-impl M*) *G*) =
TopoS-Composition-Theory.generate-valid-topology-some (*get-spec M*) (*edgesL G*) (*list-graph-to-graph G*)
⟨*proof*⟩

end
theory *TopoS-Stateful-Policy-Algorithm*
imports *TopoS-Stateful-Policy TopoS-Composition-Theory*
begin

10 Stateful Policy – Algorithm

10.1 Some unimportant lemmata

lemma *False-set*: {(*r*, *s*). *False*} = {} ⟨*proof*⟩
lemma *valid-reqs-ACS-D*: *valid-reqs M* ⟹ *valid-reqs* (*get-ACS M*)
⟨*proof*⟩
lemma *valid-reqs-IFS-D*: *valid-reqs M* ⟹ *valid-reqs* (*get-IFS M*)
⟨*proof*⟩
lemma *all-security-requirements-fulfilled-ACS-D*: *all-security-requirements-fulfilled M G* ⟹
all-security-requirements-fulfilled (*get-ACS M*) *G*
⟨*proof*⟩
lemma *all-security-requirements-fulfilled-IFS-D*: *all-security-requirements-fulfilled M G* ⟹
all-security-requirements-fulfilled (*get-IFS M*) *G*
⟨*proof*⟩
lemma *all-security-requirements-fulfilled-mono-stateful-policy-to-network-graph*:
⟦ *valid-reqs M*; *E′* ⊆ *E*; *wf-graph* (⦇ *nodes* = *V*, *edges* = *Efix* ∪ *E* ⦈) ⟧ ⟹
all-security-requirements-fulfilled M
(*stateful-policy-to-network-graph* (⦇ *hosts* = *V*, *flows-fix* = *Efix*, *flows-state* = *E* ⦈))⟹
all-security-requirements-fulfilled M
(*stateful-policy-to-network-graph* (⦇ *hosts* = *V*, *flows-fix* = *Efix*, *flows-state* = *E′* ⦈))
⟨*proof*⟩

10.2 Sketch for generating a stateful policy from a simple directed policy

Having no stateful flows, we trivially get a valid stateful policy.

lemma *trivial-stateful-policy-compliance*:
$\llbracket$ *wf-graph* (| *nodes* = V, *edges* = E |); *valid-reqs M*; *all-security-requirements-fulfilled M* (| *nodes* = V, *edges* = E |) $\rrbracket \Longrightarrow$
stateful-policy-compliance (| *hosts* = V, *flows-fix* = E, *flows-state* = {} |) (| *nodes* = V, *edges* = E |) *M*
⟨*proof*⟩

trying better

First, filtering flows that cause no IFS violations

fun *filter-IFS-no-violations-accu* :: $'v$::*vertex graph* ⇒ $'v$ *SecurityInvariant-configured list* ⇒ ($'v$ × $'v$) *list* ⇒ ($'v$ × $'v$) *list* ⇒ ($'v$ × $'v$) *list* **where**
filter-IFS-no-violations-accu G M accu [] = *accu* |
filter-IFS-no-violations-accu G M accu (*e#Es*) = (*if*
all-security-requirements-fulfilled (*get-IFS M*) (*stateful-policy-to-network-graph* (| *hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *set* (*e#accu*) |))
then filter-IFS-no-violations-accu G M (*e#accu*) *Es*
else filter-IFS-no-violations-accu G M accu Es)
definition *filter-IFS-no-violations* :: $'v$::*vertex graph* ⇒ $'v$ *SecurityInvariant-configured list* ⇒ ($'v$ × $'v$) *list* ⇒ ($'v$ × $'v$) *list* **where**
filter-IFS-no-violations G M Es = *filter-IFS-no-violations-accu G M* [] *Es*

lemma *filter-IFS-no-violations-subseteq-input*: *set* (*filter-IFS-no-violations G M Es*) ⊆ *set Es*
⟨*proof*⟩
lemma *filter-IFS-no-violations-accu-correct-induction*: *valid-reqs* (*get-IFS M*) $\Longrightarrow$ *wf-graph* (| *nodes* = V, *edges* = E |) $\Longrightarrow$
all-security-requirements-fulfilled (*get-IFS M*) (*stateful-policy-to-network-graph* (| *hosts* = V, *flows-fix* = E, *flows-state* = *set* (*accu*) |)) $\Longrightarrow$
(*set accu*) ∪ (*set edgesList*) ⊆ E $\Longrightarrow$
all-security-requirements-fulfilled (*get-IFS M*) (*stateful-policy-to-network-graph* (| *hosts* = V, *flows-fix* = E, *flows-state* = *set* (*filter-IFS-no-violations-accu* (| *nodes* = V, *edges* = E |) *M accu edgesList*) |))
⟨*proof*⟩
lemma *filter-IFS-no-violations-correct*: $\llbracket$*valid-reqs* (*get-IFS M*); *wf-graph G*;
all-security-requirements-fulfilled (*get-IFS M*) *G*;
(*set edgesList*) ⊆ *edges G* $\rrbracket \Longrightarrow$
all-security-requirements-fulfilled (*get-IFS M*) (*stateful-policy-to-network-graph* (| *hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *set* (*filter-IFS-no-violations G M edgesList*) |))
⟨*proof*⟩
lemma *filter-IFS-no-violations-accu-no-IFS*: *valid-reqs* (*get-IFS M*) $\Longrightarrow$ *wf-graph G* $\Longrightarrow$ *get-IFS M* = [] $\Longrightarrow$
(*set accu*) ∪ (*set edgesList*) ⊆ *edges G* $\Longrightarrow$
filter-IFS-no-violations-accu G M accu edgesList = *rev*(*edgesList*)@*accu*
⟨*proof*⟩

lemma *filter-IFS-no-violations-accu-maximal-induction*: *valid-reqs* (*get-IFS M*) $\Longrightarrow$ *wf-graph* (| *nodes* = V, *edges* = E |) $\Longrightarrow$
set accu ⊆ E $\Longrightarrow$ *set edgesList* ⊆ E $\Longrightarrow$
∀ *e* ∈ E − (*set accu* ∪ *set edgesList*).

¬ *all-security-requirements-fulfilled* (*get-IFS M*) (*stateful-policy-to-network-graph* ⦇ *hosts* = *V*, *flows-fix* = *E*, *flows-state* = {*e*} ∪ (*set accu*) ⦈)
⟹
let stateful = *set* (*filter-IFS-no-violations-accu* ⦇ *nodes* = *V*, *edges* = *E* ⦈ *M accu edgesList*) *in*
(∀ *e* ∈ *E* − *stateful*.
¬ *all-security-requirements-fulfilled* (*get-IFS M*) (*stateful-policy-to-network-graph* ⦇ *hosts* = *V*, *flows-fix* = *E*, *flows-state* = {*e*} ∪ *stateful* ⦈))
⟨*proof*⟩

lemma *filter-IFS-no-violations-maximal*: ⟦*valid-reqs* (*get-IFS M*); *wf-graph G*;
(*set edgesList*) = *edges G* ⟧ ⟹
let stateful = *set* (*filter-IFS-no-violations G M edgesList*) *in*
∀ *e* ∈ *edges G* − *stateful*.
¬ *all-security-requirements-fulfilled* (*get-IFS M*) (*stateful-policy-to-network-graph* ⦇ *hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = {*e*} ∪ *stateful* ⦈)
⟨*proof*⟩

corollary *filter-IFS-no-violations-maximal-allsubsets*:
assumes *a1*: *valid-reqs* (*get-IFS M*)
and *a2*: *wf-graph G*
and *a4*: (*set edgesList*) = *edges G*
shows *let stateful* = *set* (*filter-IFS-no-violations G M edgesList*) *in*
∀ *E* ⊆ *edges G* − *stateful*. *E* ≠ {} ⟶
¬ *all-security-requirements-fulfilled* (*get-IFS M*) (*stateful-policy-to-network-graph* ⦇ *hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *E* ∪ *stateful* ⦈)
⟨*proof*⟩

thm *filter-IFS-no-violations-correct filter-IFS-no-violations-maximal*

Next

fun *filter-compliant-stateful-ACS-accu* :: *'v::vertex graph* ⇒ *'v SecurityInvariant-configured list* ⇒ (*'v* × *'v*) *list* ⇒ (*'v* × *'v*) *list* ⇒ (*'v* × *'v*) *list* **where**
filter-compliant-stateful-ACS-accu G M accu [] = *accu* |
filter-compliant-stateful-ACS-accu G M accu (*e#Es*) = (*if*
e ∉ *backflows* (*edges G*) ∧ (∀ *F* ∈ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* ⦇ *hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *set* (*e#accu*) ⦈). *F* ⊆ *backflows* (*set* (*e#accu*)))
then filter-compliant-stateful-ACS-accu G M (*e#accu*) *Es*
else filter-compliant-stateful-ACS-accu G M accu Es)

definition *filter-compliant-stateful-ACS* :: *'v::vertex graph* ⇒ *'v SecurityInvariant-configured list* ⇒ (*'v* × *'v*) *list* ⇒ (*'v* × *'v*) *list* **where**
filter-compliant-stateful-ACS G M Es = *filter-compliant-stateful-ACS-accu G M* [] *Es*

lemma *filter-compliant-stateful-ACS-subseteq-input*: *set* (*filter-compliant-stateful-ACS G M Es*) ⊆ *set Es*
⟨*proof*⟩

lemma *filter-compliant-stateful-ACS-accu-correct-induction*: *valid-reqs* (*get-ACS M*) ⟹ *wf-graph* ⦇ *nodes* = *V*, *edges* = *E* ⦈ ⟹
(*set accu*) ∪ (*set edgesList*) ⊆ *E* ⟹
∀ *F* ∈ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* ⦇ *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set* (*accu*) ⦈). *F* ⊆ *backflows* (*set accu*) ⟹
(∀ *a* ∈ *set accu*. *a* ∉ (*backflows E*)) ⟹
𝒯 = ⦇ *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set* (*filter-compliant-stateful-ACS-accu* ⦇ *nodes* = *V*, *edges* = *E* ⦈ *M accu edgesList*) ⦈ ⟹
∀ *F* ∈ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* 𝒯). *F* ⊆ *backflows* (*filternew-flows-state* 𝒯)

⟨proof⟩

lemma *filter-compliant-stateful-ACS-accu-no-side-effects*: *valid-reqs* (*get-ACS M*) ⟹ *wf-graph G* ⟹
∀ *F* ∈ *get-offending-flows* (*get-ACS M*) ⦇*nodes* = *nodes G*, *edges* = *edges G* ∪ *backflows* (*edges G*)⦈. *F* ⊆ (*backflows* (*edges G*)) − (*edges G*) ⟹
(*set accu*) ∪ (*set edgesList*) ⊆ *edges G* ⟹
(∀ *a* ∈ *set accu*. *a* ∉ (*backflows* (*edges G*))) ⟹
filter-compliant-stateful-ACS-accu G M accu edgesList = *rev*([*e* ← *edgesList*. *e* ∉ *backflows* (*edges G*)])@*accu*
⟨proof⟩

lemma *filter-compliant-stateful-ACS-correct*:
assumes *a1*: *valid-reqs* (*get-ACS M*)
and *a2*: *wf-graph G*
and *a3*: *set edgesList* ⊆ *edges G*
and *a4*: *all-security-requirements-fulfilled* (*get-ACS M*) *G*
and *a5*: 𝒯 = ⦇ *hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *set* (*filter-compliant-stateful-ACS G M edgesList*) ⦈
shows ∀ *F* ∈ *get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* 𝒯). *F* ⊆ *backflows* (*filternew-flows-state* 𝒯)
⟨proof⟩

lemma *filter-compliant-stateful-ACS-accu-induction-maximal*:⟦ *valid-reqs* (*get-ACS M*); *wf-graph* ⦇ *nodes* = *V*, *edges* = *E* ⦈;
(*set edgesList*) ⊆ *E*;
(*set accu*) ⊆ *E*;
stateful = *set* (*filter-compliant-stateful-ACS-accu* ⦇ *nodes* = *V*, *edges* = *E* ⦈ *M accu edgesList*);
∀ *e* ∈ *E* − (*set edgesList* ∪ *set accu* ∪ {*e* ∈ *E*. *e* ∈ *backflows E*}).
¬ ⋃(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* ⦇ *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*e*} ⦈))
⊆ *backflows* (*filternew-flows-state* ⦇ *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *set accu* ∪ {*e*} ⦈)
⟧ ⟹
∀ *e* ∈ *E* − (*stateful* ∪ {*e* ∈ *E*. *e* ∈ *backflows E*}). ~~*E* − (computed stateful flows plus trivial stateful flows)~~
¬ ⋃(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* ⦇ *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *stateful* ∪ {*e*} ⦈))
⊆ *backflows* (*filternew-flows-state* ⦇ *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *stateful* ∪ {*e*} ⦈)
⟨proof⟩

lemma *filter-compliant-stateful-ACS-maximal*: ⟦ *valid-reqs* (*get-ACS M*); *wf-graph* ⦇ *nodes* = *V*,

edges = *E* ⦈;
(*set edgesList*) = *E*;
stateful = *set* (*filter-compliant-stateful-ACS* ⦇ *nodes* = *V*, *edges* = *E* ⦈ *M edgesList*)
⟧ ⟹
∀ *e* ∈ *E* − (*stateful* ∪ {*e* ∈ *E*. *e* ∈ *backflows E*}). ~~E computed stateful flows plus trivial stateful flows~~
¬ ⋃ (*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* ⦇ *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *stateful* ∪ {*e*} ⦈))
⊆ *backflows* (*filternew-flows-state* ⦇ *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *stateful* ∪ {*e*} ⦈)
⟨*proof*⟩

lemma *filter-compliant-stateful-ACS-maximal-allsubsets*:
assumes *a1*: *valid-reqs* (*get-ACS M*) **and** *a2*: *wf-graph* ⦇ *nodes* = *V*, *edges* = *E* ⦈
and *a3*: (*set edgesList*) = *E*
and *a4*: *stateful* = *set* (*filter-compliant-stateful-ACS* ⦇ *nodes* = *V*, *edges* = *E* ⦈ *M edgesList*)
and *a5*: *X* ⊆ *E* − (*stateful* ∪ *backflows E*) **and** *a6*: *X* ≠ {}
shows
¬ ⋃(*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* ⦇ *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *stateful* ∪ *X* ⦈))
⊆ *backflows* (*filternew-flows-state* ⦇ *hosts* = *V*, *flows-fix* = *E*, *flows-state* = *stateful* ∪ *X* ⦈)
⟨*proof*⟩

filter-compliant-stateful-ACS is correct and maximal

thm *filter-compliant-stateful-ACS-correct filter-compliant-stateful-ACS-maximal*

Getting those together. We cannot say *edgesList* = *E* here because one filters first. I guess filtering ACS first is easier, ...

definition *generate-valid-stateful-policy-IFSACS* :: *'v::vertex graph* ⇒ *'v SecurityInvariant-configured list* ⇒ (*'v* × *'v*) *list* ⇒ *'v stateful-policy* **where**
generate-valid-stateful-policy-IFSACS G M edgesList ≡ (*let filterIFS* = *filter-IFS-no-violations G M edgesList in*
(*let filterACS* = *filter-compliant-stateful-ACS G M filterIFS in* ⦇ *hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *set filterACS* ⦈))

lemma *generate-valid-stateful-policy-IFSACS-wf-stateful-policy*: **assumes** *wfG*: *wf-graph G*
and *edgesList*: (*set edgesList*) = *edges G*
shows *wf-stateful-policy* (*generate-valid-stateful-policy-IFSACS G M edgesList*)
⟨*proof*⟩

lemma *generate-valid-stateful-policy-IFSACS-select-simps*:
shows *hosts* (*generate-valid-stateful-policy-IFSACS G M edgesList*) = *nodes G*
and *flows-fix* (*generate-valid-stateful-policy-IFSACS G M edgesList*) = *edges G*
and *flows-state* (*generate-valid-stateful-policy-IFSACS G M edgesList*) ⊆ *set edgesList*
⟨*proof*⟩

lemma *generate-valid-stateful-policy-IFSACS-all-security-requirements-fulfilled-IFS*: **assumes** *validReqs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *high-level-policy-valid*: *all-security-requirements-fulfilled M G*
and *edgesList*: (*set edgesList*) ⊆ *edges G*

shows *all-security-requirements-fulfilled* (*get-IFS M*) (*stateful-policy-to-network-graph* (*generate-valid-stateful-polic*
G M edgesList))
⟨*proof*⟩

theorem *generate-valid-stateful-policy-IFSACS-stateful-policy-compliance*:
assumes *validReqs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *high-level-policy-valid*: *all-security-requirements-fulfilled M G*
and *edgesList*: (*set edgesList*) = *edges G*
and *Tau*: $\mathcal{T}$ = *generate-valid-stateful-policy-IFSACS G M edgesList*
shows *stateful-policy-compliance* $\mathcal{T}$ *G M*
⟨*proof*⟩

definition *generate-valid-stateful-policy-IFSACS-2* :: *'v::vertex graph* ⇒ *'v SecurityInvariant-configured list* ⇒ (*'v* × *'v*) *list* ⇒ *'v stateful-policy* **where**
generate-valid-stateful-policy-IFSACS-2 G M edgesList ≡
(| *hosts* = *nodes G*, *flows-fix* = *edges G*, *flows-state* = *set* (*filter-IFS-no-violations G M edgesList*) ∩ *set* (*filter-compliant-stateful-ACS G M edgesList*) |)

lemma *generate-valid-stateful-policy-IFSACS-2-wf-stateful-policy*: **assumes** *wfG*: *wf-graph G*
and *edgesList*: (*set edgesList*) = *edges G*
shows *wf-stateful-policy* (*generate-valid-stateful-policy-IFSACS-2 G M edgesList*)
⟨*proof*⟩

lemma *generate-valid-stateful-policy-IFSACS-2-select-simps*:
shows *hosts* (*generate-valid-stateful-policy-IFSACS-2 G M edgesList*) = *nodes G*
and *flows-fix* (*generate-valid-stateful-policy-IFSACS-2 G M edgesList*) = *edges G*
and *flows-state* (*generate-valid-stateful-policy-IFSACS-2 G M edgesList*) ⊆ *set edgesList*
⟨*proof*⟩

lemma *generate-valid-stateful-policy-IFSACS-2-all-security-requirements-fulfilled-IFS*: **assumes** *validReqs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *high-level-policy-valid*: *all-security-requirements-fulfilled M G*
and *edgesList*: (*set edgesList*) ⊆ *edges G*
shows *all-security-requirements-fulfilled* (*get-IFS M*) (*stateful-policy-to-network-graph* (*generate-valid-stateful-polic*
G M edgesList))
⟨*proof*⟩

lemma *generate-valid-stateful-policy-IFSACS-2-filter-compliant-stateful-ACS*:
assumes *validReqs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *high-level-policy-valid*: *all-security-requirements-fulfilled M G*
and *edgesList*: (*set edgesList*) ⊆ *edges G*
and *Tau*: $\mathcal{T}$ = *generate-valid-stateful-policy-IFSACS-2 G M edgesList*
shows ∀ *F*∈*get-offending-flows* (*get-ACS M*) (*stateful-policy-to-network-graph* $\mathcal{T}$). *F* ⊆ *backflows* (*filternew-flows-state* $\mathcal{T}$)
⟨*proof*⟩

theorem *generate-valid-stateful-policy-IFSACS-2-stateful-policy-compliance*:
assumes *validReqs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *high-level-policy-valid*: *all-security-requirements-fulfilled M G*
and *edgesList*: (*set edgesList*) = *edges G*
and *Tau*: $\mathcal{T}$ = *generate-valid-stateful-policy-IFSACS-2 G M edgesList*
shows *stateful-policy-compliance* $\mathcal{T}$ *G M*
⟨*proof*⟩

If there are no IFS requirements and the ACS requirements cause no side effects, effectively, the graph can be considered as undirected graph!

lemma *generate-valid-stateful-policy-IFSACS-2-noIFS-noACSsideeffects-imp-fullgraph*:
assumes *validReqs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *high-level-policy-valid*: *all-security-requirements-fulfilled M G*
and *edgesList*: (*set edgesList*) = *edges G*
and *no-ACS-sideeffects*: ∀ *F* ∈ *get-offending-flows* (*get-ACS M*) (|*nodes* = *nodes G*, *edges* = *edges G* ∪ *backflows* (*edges G*)|). *F* ⊆ (*backflows* (*edges G*)) − (*edges G*)
and *no-IFS*: *get-IFS M* = []
shows *stateful-policy-to-network-graph* (*generate-valid-stateful-policy-IFSACS-2 G M edgesList*) = *undirected G*
⟨*proof*⟩
lemma *generate-valid-stateful-policy-IFSACS-noIFS-noACSsideeffects-imp-fullgraph*:
assumes *validReqs*: *valid-reqs M*
and *wfG*: *wf-graph G*
and *high-level-policy-valid*: *all-security-requirements-fulfilled M G*
and *edgesList*: (*set edgesList*) = *edges G*
and *no-ACS-sideeffects*: ∀ *F* ∈ *get-offending-flows* (*get-ACS M*) (|*nodes* = *nodes G*, *edges* = *edges G* ∪ *backflows* (*edges G*)|). *F* ⊆ (*backflows* (*edges G*)) − (*edges G*)
and *no-IFS*: *get-IFS M* = []
shows *stateful-policy-to-network-graph* (*generate-valid-stateful-policy-IFSACS G M edgesList*) = *undirected G*
⟨*proof*⟩

end
theory *TopoS-Stateful-Policy-impl*
imports *TopoS-Composition-Theory-impl TopoS-Stateful-Policy-Algorithm*
begin

11 Stateful Policy – List Implementaion

record *'v stateful-list-policy* =
hostsL :: *'v list*
flows-fixL :: (*'v* × *'v*) *list*
flows-stateL :: (*'v* × *'v*) *list*

definition *stateful-list-policy-to-list-graph* :: *'v stateful-list-policy* ⇒ *'v list-graph* **where**
stateful-list-policy-to-list-graph $\mathcal{T}$ = (| *nodesL* = *hostsL* $\mathcal{T}$, *edgesL* = (*flows-fixL* $\mathcal{T}$) @ [*e* ← *flows-stateL* $\mathcal{T}$. *e* ∉ *set* (*flows-fixL* $\mathcal{T}$)] @ [*e* ← *backlinks* (*flows-stateL* $\mathcal{T}$). *e* ∉ *set* (*flows-fixL* $\mathcal{T}$)] |)

lemma *stateful-list-policy-to-list-graph-complies*:
list-graph-to-graph (*stateful-list-policy-to-list-graph* (| *hostsL* = *V*, *flows-fixL* = E_f, *flows-stateL* = E_σ |)) =
stateful-policy-to-network-graph (| *hosts* = *set V*, *flows-fix* = *set* E_f, *flows-state* = *set* E_σ |)
⟨*proof*⟩

lemma *wf-list-graph-stateful-list-policy-to-list-graph*:
wf-list-graph G ⟹ *distinct E* ⟹ *set E* ⊆ *set* (*edgesL G*) ⟹ *wf-list-graph* (*stateful-list-policy-to-list-graph* (|*hostsL* = *nodesL G*, *flows-fixL* = *edgesL G*, *flows-stateL* = *E*|))
⟨*proof*⟩

11.1 Algorithms

fun *filter-IFS-no-violations-accu* :: *'v list-graph* ⇒ *'v SecurityInvariant list* ⇒ (*'v* × *'v*) *list* ⇒ (*'v* × *'v*) *list* ⇒ (*'v* × *'v*) *list* **where**
filter-IFS-no-violations-accu G M accu [] = *accu* |
filter-IFS-no-violations-accu G M accu (*e*#*Es*) = (*if*
all-security-requirements-fulfilled (*TopoS-Composition-Theory-impl.get-IFS M*) (*stateful-list-policy-to-list-graph* (| *hostsL* = *nodesL G*, *flows-fixL* = *edgesL G*, *flows-stateL* = (*e*#*accu*) |))
then filter-IFS-no-violations-accu G M (*e*#*accu*) *Es*
else filter-IFS-no-violations-accu G M accu Es)
definition *filter-IFS-no-violations* :: *'v list-graph* ⇒ *'v SecurityInvariant list* ⇒ (*'v* × *'v*) *list* **where**
filter-IFS-no-violations G M = *filter-IFS-no-violations-accu G M* [] (*edgesL G*)

lemma *filter-IFS-no-violations-accu-distinct*: ⟦ *distinct* (*Es*@*accu*) ⟧ ⟹ *distinct* (*filter-IFS-no-violations-accu G M accu Es*)
⟨*proof*⟩

lemma *filter-IFS-no-violations-accu-complies*:
⟦∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*;
wf-list-graph G; *set Es* ⊆ *set* (*edgesL G*); *set accu* ⊆ *set* (*edgesL G*); *distinct* (*Es*@*accu*) ⟧ ⟹
filter-IFS-no-violations-accu G (*get-impl M*) *accu Es* = *TopoS-Stateful-Policy-Algorithm.filter-IFS-no-violations-accu* (*list-graph-to-graph G*) (*get-spec M*) *accu Es*
⟨*proof*⟩

lemma *filter-IFS-no-violations-complies*:
⟦ ∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*; *wf-list-graph G* ⟧ ⟹
filter-IFS-no-violations G (*get-impl M*) = *TopoS-Stateful-Policy-Algorithm.filter-IFS-no-violations* (*list-graph-to-graph G*) (*get-spec M*) (*edgesL G*)
⟨*proof*⟩

fun *filter-compliant-stateful-ACS-accu* :: *'v list-graph* ⇒ *'v SecurityInvariant list* ⇒ (*'v* × *'v*) *list*

⇒ ('v × 'v) *list* ⇒ ('v × 'v) *list* **where**
filter-compliant-stateful-ACS-accu G M accu [] = *accu* |
filter-compliant-stateful-ACS-accu G M accu (*e#Es*) = (*if*
e ∉ *set* (*backlinks* (*edgesL G*)) ∧ (∀ *F* ∈ *set* (*implc-get-offending-flows* (*get-ACS M*) (*stateful-list-policy-to-list-graph* (| *hostsL* = *nodesL G*, *flows-fixL* = *edgesL G*, *flows-stateL* = (*e#accu*) |))). *set F* ⊆ *set* (*backlinks* (*e#accu*)))
then filter-compliant-stateful-ACS-accu G M (*e#accu*) *Es*
else filter-compliant-stateful-ACS-accu G M accu Es)

definition *filter-compliant-stateful-ACS* :: 'v *list-graph* ⇒ 'v *SecurityInvariant list* ⇒ ('v × 'v) *list* **where**
filter-compliant-stateful-ACS G M = *filter-compliant-stateful-ACS-accu G M* [] (*edgesL G*)

lemma *filter-compliant-stateful-ACS-accu-complies*:
⟦∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*;
wf-list-graph G; *set Es* ⊆ *set* (*edgesL G*); *set accu* ⊆ *set* (*edgesL G*); *distinct* (*Es@accu*) ⟧ ⟹
filter-compliant-stateful-ACS-accu G (*get-impl M*) *accu Es* = *TopoS-Stateful-Policy-Algorithm.filter-compliant-statefu* (*list-graph-to-graph G*) (*get-spec M*) *accu Es*
⟨*proof*⟩

lemma *filter-compliant-stateful-ACS-cont-complies*:
⟦ ∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*; *wf-list-graph G*; *set Es* ⊆ *set* (*edgesL G*); *distinct Es* ⟧ ⟹
filter-compliant-stateful-ACS-accu G (*get-impl M*) [] *Es* = *TopoS-Stateful-Policy-Algorithm.filter-compliant-stateful-A* (*list-graph-to-graph G*) (*get-spec M*) *Es*
⟨*proof*⟩

lemma *filter-compliant-stateful-ACS-complies*:
⟦ ∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*; *wf-list-graph G* ⟧ ⟹
filter-compliant-stateful-ACS G (*get-impl M*) = *TopoS-Stateful-Policy-Algorithm.filter-compliant-stateful-ACS* (*list-graph-to-graph G*) (*get-spec M*) (*edgesL G*)
⟨*proof*⟩

definition *generate-valid-stateful-policy-IFSACS* :: 'v *list-graph* ⇒ 'v *SecurityInvariant list* ⇒ 'v *stateful-list-policy* **where**
generate-valid-stateful-policy-IFSACS G M = (*let filterIFS* = *filter-IFS-no-violations G M in*
(*let filterACS* = *filter-compliant-stateful-ACS-accu G M* [] *filterIFS in* (| *hostsL* = *nodesL G*, *flows-fixL* = *edgesL G*, *flows-stateL* = *filterACS* |)))

fun *inefficient-list-intersect* :: 'a *list* ⇒ 'a *list* ⇒ 'a *list* **where**
inefficient-list-intersect [] *bs* = [] |
inefficient-list-intersect (*a#as*) *bs* = (*if a* ∈ *set bs then a#*(*inefficient-list-intersect as bs*) *else inefficient-list-intersect as bs*)

lemma *inefficient-list-intersect-correct*: *set* (*inefficient-list-intersect a b*) = (*set a*) ∩ (*set b*)
⟨*proof*⟩

definition *generate-valid-stateful-policy-IFSACS-2* :: *'v list-graph* ⇒ *'v SecurityInvariant list* ⇒ *'v stateful-list-policy* **where**
generate-valid-stateful-policy-IFSACS-2 G M =
(| *hostsL* = *nodesL G*, *flows-fixL* = *edgesL G*, *flows-stateL* = *inefficient-list-intersect* (*filter-IFS-no-violations G M*) (*filter-compliant-stateful-ACS G M*) |)

lemma *generate-valid-stateful-policy-IFSACS-2-complies*: ⟦∀ (*m-impl*, *m-spec*) ∈ *set M*. *SecurityInvariant-complies-formal-def m-impl m-spec*;
wf-list-graph G;
valid-reqs (*get-spec M*);
TopoS-Composition-Theory.all-security-requirements-fulfilled (*get-spec M*) (*list-graph-to-graph G*);
$\mathcal{T}$ = (*generate-valid-stateful-policy-IFSACS-2 G* (*get-impl M*))⟧ ⟹
stateful-policy-compliance (|*hosts* = *set* (*hostsL* $\mathcal{T}$), *flows-fix* = *set* (*flows-fixL* $\mathcal{T}$), *flows-state* = *set* (*flows-stateL* $\mathcal{T}$) |) (*list-graph-to-graph G*) (*get-spec M*)
⟨*proof*⟩

end
theory *METASINVAR-SystemBoundary*
imports *SINVAR-BLPtrusted-impl*
SINVAR-SubnetsInGW-impl
../TopoS-Composition-Theory-impl
begin

11.1.1 Meta SecurityInvariant: System Boundaries

datatype *system-components* = *SystemComponent*
| *SystemBoundaryInput*
| *SystemBoundaryOutput*
| *SystemBoundaryInputOutput*

fun *system-components-to-subnets* :: *system-components* ⇒ *subnets* **where**
system-components-to-subnets SystemComponent = *Member* |
system-components-to-subnets SystemBoundaryInput = *InboundGateway* |
system-components-to-subnets SystemBoundaryOutput = *Member* |
system-components-to-subnets SystemBoundaryInputOutput = *InboundGateway*

fun *system-components-to-blp* :: *system-components* ⇒ *SINVAR-BLPtrusted.node-config* **where**
system-components-to-blp SystemComponent = (| *security-level* = *1*, *trusted* = *False* |) |
system-components-to-blp SystemBoundaryInput = (| *security-level* = *1*, *trusted* = *False* |) |
system-components-to-blp SystemBoundaryOutput = (| *security-level* = *0*, *trusted* = *True* |) |
system-components-to-blp SystemBoundaryInputOutput = (| *security-level* = *0*, *trusted* = *True* |)

definition *new-meta-system-boundary* :: (*'v::vertex* × *system-components*) *list* ⇒ *string* ⇒ (*'v SecurityInvariant*) *list* **where**
new-meta-system-boundary C description = [
new-configured-list-SecurityInvariant SINVAR-LIB-SubnetsInGW (|
node-properties = *map-of* (*map* (λ(*v*,*c*). (*v*, *system-components-to-subnets c*)) *C*)
|) (*description* @ '' (*ACS*)'')

,
new-configured-list-SecurityInvariant SINVAR-LIB-BLPtrusted ⦇
node-properties = *map-of* (*map* (λ(*v*,*c*). (*v*, *system-components-to-blp c*)) *C*)
⦈ (*description* @ ″ (*IFS*)″)
]

lemma *system-components-to-subnets*:
SINVAR-SubnetsInGW.allowed-subnet-flow
SINVAR-SubnetsInGW.default-node-properties
(*system-components-to-subnets c*) ⟷
c = *SystemBoundaryInput* ∨ *c* = *SystemBoundaryInputOutput*
⟨*proof*⟩

lemma *system-components-to-blp*:
(¬ *trusted SINVAR-BLPtrusted.default-node-properties* ⟶
security-level (*system-components-to-blp c*) ≤ *security-level SINVAR-BLPtrusted.default-node-properties*)
⟷
c = *SystemBoundaryOutput* ∨ *c* = *SystemBoundaryInputOutput*
⟨*proof*⟩

lemma *all-security-requirements-fulfilled* (*new-meta-system-boundary C description*) *G* ⟷
(∀ (v_1, v_2) ∈ *set* (*edgesL G*). *case* ((*map-of C*) v_1, (*map-of C*) v_2)
of
— No restrictions outside of the component
(*None*, *None*) ⇒ *True*

— no restrictions inside the component
| (*Some c1*, *Some c2*) ⇒ *True*

— System Boundaries Input
| (*None*, *Some SystemBoundaryInputOutput*) ⇒ *True*
| (*None*, *Some SystemBoundaryInput*) ⇒ *True*

— System Boundaries Output
| (*Some SystemBoundaryOutput*, *None*) ⇒ *True*
| (*Some SystemBoundaryInputOutput*, *None*) ⇒ *True*

— everything else is prohibited
| - ⇒ *False*
)
⟨*proof*⟩

value[*code*] *let nodes* = [*1*,*2*,*3*,*4*,*8*,*9*,*10*];
sinvars = *new-meta-system-boundary*
[(*1*::*int*, *SystemBoundaryInput*),
(*2*, *SystemComponent*),
(*3*, *SystemBoundaryOutput*),
(*4*, *SystemBoundaryInputOutput*)
] ″*foobar*″
in generate-valid-topology sinvars ⦇*nodesL* = *nodes*, *edgesL* = *List.product nodes nodes* ⦈

end
theory *TopoS-Impl*
imports *TopoS-Library TopoS-Composition-Theory-impl*

Security-Invariants/METASINVAR-SystemBoundary

Lib/ML-GraphViz
TopoS-Stateful-Policy-impl
begin

12 ML Visualization Interface

definition *print-offending-flows-debug* ::
'v SecurityInvariant list ⇒ *'v list-graph* ⇒ (*string* × (*'v* × *'v*) *list list*) *list* **where**
print-offending-flows-debug M G = *map*
(*λm.*
(*implc-description m* @ '' ('' @ *implc-type m* @ '')''
, *implc-offending-flows m G*)
) *M*

⟨*ML*⟩

12.1 Utility Functions

fun *rembiflowdups* :: (*'a* × *'a*) *list* ⇒ (*'a* × *'a*) *list* **where**
rembiflowdups [] = [] |
rembiflowdups ((*s*,*r*)#*as*) = (*if* (*s*,*r*) ∈ *set as* ∨ (*r*,*s*) ∈ *set as then rembiflowdups as else* (*s*,*r*)#*rembiflowdups as*)

lemma *rembiflowdups-complete*: ⟦ ∀ (*s*,*r*) ∈ *set x*. (*r*,*s*) ∈ *set x* ⟧ ⟹ *set* (*rembiflowdups x*) ∪ *set* (*backlinks* (*rembiflowdups x*)) = *set x*
⟨*proof*⟩

only for prettyprinting

definition *filter-for-biflows*:: (*'a* × *'a*) *list* ⇒ (*'a* × *'a*) *list* **where**
filter-for-biflows E ≡ [*e* ← *E*. (*snd e*, *fst e*) ∈ *set E*]

definition *filter-for-uniflows*:: (*'a* × *'a*) *list* ⇒ (*'a* × *'a*) *list* **where**
filter-for-uniflows E ≡ [*e* ← *E*. (*snd e*, *fst e*) ∉ *set E*]

lemma *filter-for-biflows-correct*: ∀ (*s*,*r*) ∈ *set* (*filter-for-biflows E*). (*r*,*s*) ∈ *set* (*filter-for-biflows E*)
⟨*proof*⟩

lemma *filter-for-biflows-un-filter-for-uniflows*: *set* (*filter-for-biflows E*) ∪ *set* (*filter-for-uniflows E*) = *set E*
⟨*proof*⟩

definition *partition-by-biflows* :: (*'a* × *'a*) *list* ⇒ ((*'a* × *'a*) *list* × (*'a* × *'a*) *list*) **where**
partition-by-biflows E ≡ (*rembiflowdups* (*filter-for-biflows E*), *remdups* (*filter-for-uniflows E*))

lemma *partition-by-biflows-correct*: *case partition-by-biflows E of* (*biflows*, *uniflows*) ⇒ *set biflows* ∪ *set* (*backlinks* (*biflows*)) ∪ *set uniflows* = *set E*
⟨*proof*⟩

lemma *partition-by-biflows* [(*1*::*int*, *1*::*int*), (*1*,*2*), (*2*, *1*), (*1*,*3*)] = ([(*1*, *1*), (*2*, *1*)], [(*1*, *3*)]) ⟨*proof*⟩

⟨*ML*⟩

definition *internal-get-invariant-types-list*:: *'a SecurityInvariant list* ⇒ *string list* **where**
internal-get-invariant-types-list M ≡ *map implc-type M*

definition *internal-node-configs* :: *'a list-graph* ⇒ (*'a* ⇒ *'b*) ⇒ (*'a* × *'b*) *list* **where**
internal-node-configs G config ≡ *zip* (*nodesL G*) (*map config* (*nodesL G*))

⟨*ML*⟩

end

13 Network Security Policy Verification

theory *Network-Security-Policy-Verification*
imports
TopoS-Interface
TopoS-Interface-impl
TopoS-Library
TopoS-Composition-Theory
TopoS-Stateful-Policy
TopoS-Composition-Theory-impl
TopoS-Stateful-Policy-Algorithm
TopoS-Stateful-Policy-impl
TopoS-Impl
begin

14 A small Tutorial

We demonstrate usage of the executable theory.

Everything that is indented and starts with 'Interlude:' summarizes the main correctness proofs and can be skipped if only the implementation is concerned

14.1 Policy

The secuity policy is a directed graph.

definition *policy* :: *nat list-graph* **where**
policy ≡ ⦇ *nodesL* = [*1*,*2*,*3*],

edgesL = [(*1*,*2*), (*2*,*2*), (*2*,*3*)] ⦈

It is syntactically well-formed

lemma *wf-list-graph-policy*: *wf-list-graph policy* ⟨*proof*⟩

In contrast, this is not a syntactically well-formed graph.

lemma ¬ *wf-list-graph* ⦇ *nodesL* = [*1*,*2*]::*nat list*, *edgesL* = [(*1*,*2*), (*2*,*2*), (*2*,*3*)] ⦈ ⟨*proof*⟩

Our *policy* has three rules.

lemma *length* (*edgesL policy*) = *3* ⟨*proof*⟩

14.2 Security Invariants

We construct a security invariant. Node *2* has confidential data

definition *BLP-security-levels* :: *nat* ⇀ *SINVAR-BLPtrusted.node-config***where**
BLP-security-levels ≡ [*2* ↦ ⦇ *security-level* = *1*, *trusted* = *False* ⦈]

definition *BLP-m*::(*nat SecurityInvariant*) **where**
BLP-m ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-BLPtrusted* ⦇
node-properties = *BLP-security-levels*
⦈ *''Two has confidential information''*

Interlude: *BLP-m* is a valid implementation of a SecurityInvariant

definition *BLP-m-spec* :: *nat SecurityInvariant-configured option***where**
BLP-m-spec ≡ *new-configured-SecurityInvariant* (
SINVAR-BLPtrusted.sinvar,
SINVAR-BLPtrusted.default-node-properties,
SINVAR-BLPtrusted.receiver-violation,
SecurityInvariant.node-props SINVAR-BLPtrusted.default-node-properties ⦇
node-properties = *BLP-security-levels*
⦈)

Fist, we need to show that the formal definition obeys all requirements, *new-configured-SecurityInvariant* verifies this. To double check, we manually give the configuration.

lemma *BLP-m-spec*: **assumes** *nP* = (λ *v*. (*case BLP-security-levels v of Some c* ⇒ *c* | *None* ⇒ *SINVAR-BLPtrusted.default-node-properties*))
shows *BLP-m-spec* = *Some* ⦇
c-sinvar = (λ*G*. *SINVAR-BLPtrusted.sinvar G nP*),
c-offending-flows = (λ*G*. *SecurityInvariant-withOffendingFlows.set-offending-flows SINVAR-BLPtrusted.sinvar G nP*),
c-isIFS = *SINVAR-BLPtrusted.receiver-violation*
⦈ (**is** *BLP-m-spec* = *Some ?Spec*)
⟨*proof*⟩
lemma *valid-reqs-BLP*: *valid-reqs* [*the BLP-m-spec*]
⟨*proof*⟩

Interlude: While *BLP-m* is executable code, we will now show that this executable code complies with its formal definition.

lemma *complies-BLP*: *SecurityInvariant-complies-formal-def BLP-m* (*the BLP-m-spec*)
⟨*proof*⟩

We define the list of all security invariants of type *nat SecurityInvariant list*. The type *nat* is because the policy's nodes are of type *nat*.

definition *security-invariants* = [*BLP-m*]

We can see that the policy does not fulfill the security invariants.

lemma ¬ *all-security-requirements-fulfilled security-invariants policy* ⟨*proof*⟩

We ask why. Obviously, node 2 leaks confidential data to node 3.

value *implc-get-offending-flows security-invariants policy*
lemma *implc-get-offending-flows security-invariants policy* = [[(*2*, *3*)]] ⟨*proof*⟩

Interlude: the implementation *implc-get-offending-flows* corresponds to the formal definition *get-offending-flows*

lemma *set* ' *set* (*implc-get-offending-flows* (*get-impl* [(*BLP-m*, *the BLP-m-spec*)]) *policy*) = *get-offending-flows* (*get-spec* [(*BLP-m*, *the BLP-m-spec*)]) (*list-graph-to-graph policy*)
⟨*proof*⟩

Visualization of the violation (only in interactive mode)

⟨*ML*⟩

Experimental: the config (only one) can be added to the end.

⟨*ML*⟩

The policy has a flaw. We throw it away and generate a new one which fulfills the invariants.

definition *max-policy* = *generate-valid-topology security-invariants* (|*nodesL* = *nodesL policy*, *edgesL* = *List.product* (*nodesL policy*) (*nodesL policy*) |)

Interlude: the implementation *implc-get-offending-flows* corresponds to the formal definition *get-offending-flows*

thm *generate-valid-topology-complies*

Interlude: the formal definition is sound

thm *generate-valid-topology-sound*

Here, it is also complete

lemma *wf-graph G* ⟹ *max-topo* [*the BLP-m-spec*] (*TopoS-Composition-Theory.generate-valid-topology* [*the BLP-m-spec*] (*fully-connected G*))
⟨*proof*⟩

Calculating the maximum policy

value *max-policy*
lemma *max-policy* = (|*nodesL* = [*1*, *2*, *3*], *edgesL* = [(*1*, *1*), (*1*, *2*), (*1*, *3*), (*2*, *2*), (*3*, *1*), (*3*, *2*), (*3*, *3*)]|) ⟨*proof*⟩

Visualizing the maximum policy (only in interactive mode)

⟨*ML*⟩

Of course, all security invariants hold for the maximum policy.

lemma *all-security-requirements-fulfilled security-invariants max-policy* ⟨*proof*⟩

14.3 A stateful implementation

We generate a stateful policy

definition *stateful-policy* = *generate-valid-stateful-policy-IFSACS-2 policy security-invariants*

When thinking about it carefully, no flow can be stateful without introducing an information leakage here!

value *stateful-policy*
lemma *stateful-policy* = (|*hostsL* = [*1*, *2*, *3*], *flows-fixL* = [(*1*, *2*), (*2*, *2*), (*2*, *3*)], *flows-stateL* = []|)
⟨*proof*⟩

Interlude: the stateful policy we are computing fulfills all the necessary properties

thm *generate-valid-stateful-policy-IFSACS-2-complies*

thm *filter-compliant-stateful-ACS-correct filter-compliant-stateful-ACS-maximal*
thm *filter-IFS-no-violations-correct filter-IFS-no-violations-maximal*

Visualizing the stateful policy (only in interactive mode)

⟨*ML*⟩

This is how it would look like if (*3*::′*a*, *1*) were a stateful flow

⟨*ML*⟩

hide-const *policy security-invariants max-policy stateful-policy*

end
theory *Example-BLP*
imports *TopoS-Library*
begin

definition *BLPexample1*::*bool* **where**
BLPexample1 ≡ (*nm-eval SINVAR-LIB-BLPbasic*) *fabNet* (| *node-properties* = [″*PresenceSensor*″ ↦ *2*,
″*Webcam*″ ↦ *3*,
″*SensorSink*″ ↦ *3*,
″*Statistics*″ ↦ *3*] |)
definition *BLPexample3*::(*string* × *string*) *list list* **where**
BLPexample3 ≡ (*nm-offending-flows SINVAR-LIB-BLPbasic*) *fabNet* ((*nm-node-props SINVAR-LIB-BLPbasic*) *sensorProps-NMParams-try3*)

value *BLPexample1*
value *BLPexample3*

end
theory *TopoS-generateCode*
imports
TopoS-Library
Example-BLP
begin

⟨*ML*⟩

export-code
— generic network security invariants
SINVAR-LIB-BLPbasic
SINVAR-LIB-Dependability
SINVAR-LIB-DomainHierarchyNG
SINVAR-LIB-Subnets
SINVAR-LIB-BLPtrusted
SINVAR-LIB-PolEnforcePointExtended
SINVAR-LIB-Sink
SINVAR-LIB-NonInterference
SINVAR-LIB-SubnetsInGW
SINVAR-LIB-CommunicationPartners
— accessors to the packed invariants
nm-eval
nm-node-props
nm-offending-flows
nm-sinvar
nm-default
nm-receiver-violation nm-name
— TopoS Params
node-properties
— Finite Graph functions
FiniteListGraph.wf-list-graph
FiniteListGraph.add-node
FiniteListGraph.delete-node
FiniteListGraph.add-edge
FiniteListGraph.delete-edge
FiniteListGraph.delete-edges
— Examples
BLPexample1 BLPexample3
in *Scala*

end
theory *SINVAR-Examples*
imports
TopoS-Interface
TopoS-Interface-impl
TopoS-Library
TopoS-Composition-Theory
TopoS-Stateful-Policy
TopoS-Composition-Theory-impl
TopoS-Stateful-Policy-Algorithm
TopoS-Stateful-Policy-impl
TopoS-Impl
begin

⟨*ML*⟩

definition *make-policy* :: (*'a SecurityInvariant*) *list* ⇒ *'a list* ⇒ *'a list-graph* **where**
make-policy sinvars V ≡ *generate-valid-topology sinvars* (|*nodesL* = *V*, *edgesL* = *List.product V V* |)

context begin
private definition *SINK-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-Sink* (|
node-properties = [*''Bot1''* ↦ *Sink*,
''Bot2'' ↦ *Sink*,
''MissionControl1'' ↦ *SinkPool*,
''MissionControl2'' ↦ *SinkPool*
]
|) *''bots and control are infromation sink''*
value[*code*] *make-policy* [*SINK-m*] [*''INET''*, *''Supervisor''*, *''Bot1''*, *''Bot2''*, *''MissionControl1''*, *''MissionControl2''*]
⟨*ML*⟩
end

context begin
private definition *ACL-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-CommunicationPartners* (|
node-properties = [*''db1''* ↦ *Master* [*''h1''*, *''h2''*],
''db2'' ↦ *Master* [*''db1''*],
''h1'' ↦ *Care*,
''h2'' ↦ *Care*
]
|) *''ACL for databases''*
value[*code*] *make-policy* [*ACL-m*] [*''db1''*, *''db2''*, *''h1''*, *''h2''*, *''h3''*]
⟨*ML*⟩
end

definition *CommWith-m*::(*nat SecurityInvariant*) **where**
CommWith-m ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-ACLcommunicateWith* (|
node-properties = [
1 ↦ [*2*,*3*],
2 ↦ [*3*]]
|) *''One can only talk to 2,3''*

Experimental: the config (only one) can be added to the end.

⟨*ML*⟩

value[*code*] *make-policy* [*CommWith-m*] [*1*,*2*,*3*]
value[*code*] *implc-offending-flows CommWith-m* (|*nodesL* = [*1*,*2*,*3*,*4*], *edgesL* = *List.product* [*1*,*2*,*3*,*4*] [*1*,*2*,*3*,*4*] |)

value[*code*] *make-policy* [*CommWith-m*] [*1*,*2*,*3*,*4*]

⟨*ML*⟩

lemma *implc-offending-flows* (*new-configured-list-SecurityInvariant SINVAR-LIB-ACLcommunicateWith* (|
node-properties = [
1::*nat* ↦ [*1*,*2*,*3*],
2 ↦ [*1*,*2*,*3*,*4*],
3 ↦ [*1*,*2*,*3*,*4*],
4 ↦ [*1*,*2*,*3*,*4*]]
|) *''usefull description here''*) (|*nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*2*), (*1*,*3*), (*2*,*3*), (*3*, *4*)]
|) =
[[(*1*, *2*), (*1*, *3*)], [(*1*, *3*), (*2*, *3*)], [(*3*, *4*)]] ⟨*proof*⟩

context begin
private definition *G-dep* :: *nat list-graph* **where**
G-dep ≡ (|*nodesL* = [*1*::*nat*,*2*,*3*,*4*,*5*,*6*,*7*], *edgesL* = [(*1*,*2*), (*2*,*1*), (*2*,*3*),
(*4*,*5*), (*5*,*6*), (*6*,*7*)] |)
private lemma *wf-list-graph G-dep* ⟨*proof*⟩ **definition** *DEP-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-Dependability* (|
node-properties = *Some* ∘ *dependability-fix-nP G-dep* (λ-. *0*)
|) *''automatically computed dependability invariant''*
⟨*ML*⟩

Connecting (*3*::*'a*, *4*::*'b*). This causes only one offedning flow at (*3*::*'a*, *4*::*'b*).

⟨*ML*⟩

We try to increase the dependability level at *3*::*'a*. Suddenly, offending flows everywhere.

⟨*ML*⟩
lemma *implc-offending-flows* (*new-configured-list-SecurityInvariant SINVAR-LIB-Dependability* (|
node-properties = *Some* ∘ ((*dependability-fix-nP G-dep* (λ-. *0*))(*3* := *2*))
|) *''changed deps''*)
(*G-dep*(|*edgesL* := (*3*,*4*)#*edgesL G-dep*|)) =
[[(*3*, *4*)], [(*1*, *2*), (*2*, *1*), (*5*, *6*)], [(*1*, *2*), (*4*, *5*)], [(*2*, *1*), (*4*, *5*)], [(*2*, *3*), (*4*, *5*)], [(*2*, *3*), (*5*, *6*)]]
⟨*proof*⟩

If we recompute the dependability levels for the changed graph, we see that suddenly, The level at *1* and *2*::*'a* increased, though we only added the edge (*3*::*'a*, *4*::*'b*). This hints that we connected the graph. If an attacker can now compromise *1*, she may be able to peek much deeper into the network.

⟨*ML*⟩

Dependability is reflexive, a host can depend on itself.

⟨*ML*⟩

end

context begin
private definition *G-noninter* :: *nat list-graph* **where**
G-noninter ≡ (|*nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*2*), (*1*,*3*), (*2*,*3*), (*3*, *4*)] |)
private lemma *wf-list-graph G-noninter* ⟨*proof*⟩ **definition** *NonI-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-NonInterference* (|
node-properties = [
1::*nat* ↦ *Interfering*,
2 ↦ *Unrelated*,
3 ↦ *Unrelated*,
4 ↦ *Interfering*]
|) ''*One and Four interfere*''
⟨*ML*⟩

lemma *implc-offending-flows NonI-m G-noninter* = [[(*1*, *2*), (*1*, *3*)], [(*1*, *3*), (*2*, *3*)], [(*3*, *4*)]]
⟨*proof*⟩

⟨*ML*⟩

lemma *implc-offending-flows NonI-m* (|*nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*2*), (*1*,*3*), (*2*,*3*), (*4*, *3*)] |) =
[[(*1*, *2*), (*1*, *3*)], [(*1*, *3*), (*2*, *3*)], [(*4*, *3*)]]
⟨*proof*⟩

In comparison, *SINVAR-LIB-ACLcommunicateWith* is less strict. Changing the direction of the edge (*3*::'*a*, *4*::'*b*) removes the access from *1* to *4*::'*a* and the invariant holds.

lemma *implc-offending-flows* (*new-configured-list-SecurityInvariant SINVAR-LIB-ACLcommunicateWith* (|
node-properties = [
1::*nat* ↦ [*1*,*2*,*3*],
2 ↦ [*1*,*2*,*3*,*4*],
3 ↦ [*1*,*2*,*3*,*4*],
4 ↦ [*1*,*2*,*3*,*4*]]
|) ''*One must not access Four*'') (|*nodesL* = [*1*::*nat*,*2*,*3*,*4*], *edgesL* = [(*1*,*2*), (*1*,*3*), (*2*,*3*), (*4*, *3*)] |) = [] ⟨*proof*⟩
end

context begin
private definition *subnets-host-attributes* ≡ [
''*v11*'' ↦ *Subnet 1*,
''*v12*'' ↦ *Subnet 1*,
''*v13*'' ↦ *Subnet 1*,
''*v1b*'' ↦ *BorderRouter 1*,
''*v21*'' ↦ *Subnet 2*,
''*v22*'' ↦ *Subnet 2*,
''*v23*'' ↦ *Subnet 2*,
''*v2b*'' ↦ *BorderRouter 2*,
''*v3b*'' ↦ *BorderRouter 3*
]
private definition *Subnets-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-Subnets* (|

node-properties = *subnets-host-attributes*
⦈ ''*Collaborating hosts*''

private definition *subnet-hosts* ≡ [''*v11*'', ''*v12*'', ''*v13*'', ''*v1b*'',
''*v21*'', ''*v22*'', ''*v23*'', ''*v2b*'',
''*v3b*'', ''*vo*'']

private lemma *dom* (*subnets-host-attributes*) ⊆ *set* (*subnet-hosts*)
⟨*proof*⟩
value[*code*] *make-policy* [*Subnets-m*] *subnet-hosts*
⟨*ML*⟩

Emulating the same but with accessible members with SubnetsInGW and ACLs

private definition *SubnetsInGW-ACL-ms* ≡ [*new-configured-list-SecurityInvariant SINVAR-LIB-SubnetsInGW* ⦇
node-properties = [''*v11*'' ↦ *Member*, ''*v12*'' ↦ *Member*, ''*v13*'' ↦ *Member*, ''*v1b*'' ↦ *InboundGateway*]
⦈ ''*v1 subnet*'',
new-configured-list-SecurityInvariant SINVAR-LIB-CommunicationPartners ⦇
node-properties = [''*v1b*'' ↦ *Master* [''*v11*'', ''*v12*'', ''*v13*'', ''*v2b*'', ''*v3b*''],
''*v11*'' ↦ *Care*,
''*v12*'' ↦ *Care*,
''*v13*'' ↦ *Care*,
''*v2b*'' ↦ *Care*,
''*v3b*'' ↦ *Care*
]
⦈ ''*v1b ACL*'',
new-configured-list-SecurityInvariant SINVAR-LIB-SubnetsInGW ⦇
node-properties = [''*v21*'' ↦ *Member*, ''*v22*'' ↦ *Member*, ''*v23*'' ↦ *Member*, ''*v2b*'' ↦ *InboundGateway*]
⦈ ''*v2 subnet*'',
new-configured-list-SecurityInvariant SINVAR-LIB-CommunicationPartners ⦇
node-properties = [''*v2b*'' ↦ *Master* [''*v21*'', ''*v22*'', ''*v23*'', ''*v1b*'', ''*v3b*''],
''*v21*'' ↦ *Care*,
''*v22*'' ↦ *Care*,
''*v23*'' ↦ *Care*,
''*v1b*'' ↦ *Care*,
''*v3b*'' ↦ *Care*
]
⦈ ''*v2b ACL*'',
~~*new-configured-list-SecurityInvariant SINVAR-LIB-SubnetsInGW* ⦇ *node-properties* = [''*v3b*'' ↦ *InboundGateway*] ⦈ ''*v3b*'',~~
new-configured-list-SecurityInvariant SINVAR-LIB-CommunicationPartners ⦇
node-properties = [''*v3b*'' ↦ *Master* [''*v1b*'', ''*v2b*''],
''*v1b*'' ↦ *Care*,
''*v2b*'' ↦ *Care*
]
⦈ ''*v3b ACL*'']

value[*code*] *make-policy SubnetsInGW-ACL-ms subnet-hosts*

lemma *set* (*edgesL* (*make-policy* [*Subnets-m*] *subnet-hosts*)) ⊆ *set* (*edgesL* (*make-policy SubnetsInGW-ACL-ms subnet-hosts*)) ⟨*proof*⟩

lemma [*e* <− *edgesL* (*make-policy SubnetsInGW-ACL-ms subnet-hosts*). *e* ∉ *set* (*edgesL* (*make-policy* [*Subnets-m*] *subnet-hosts*))] =
[(''*v1b*'', ''*v11*''), (''*v1b*'', ''*v12*''), (''*v1b*'', ''*v13*''), (''*v2b*'', ''*v21*''), (''*v2b*'', ''*v22*''), (''*v2b*'', ''*v23*'')]
⟨*proof*⟩

⟨ML⟩
end

context begin
private definition *secgwext-host-attributes* ≡ [
''hypervisor'' ↦ *PolEnforcePoint*,
''securevm1'' ↦ *DomainMember*,
''securevm2'' ↦ *DomainMember*,
''publicvm1'' ↦ *AccessibleMember*,
''publicvm2'' ↦ *AccessibleMember*
]
private definition *SecGwExt-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-PolEnforcePointExtended*
⦇
node-properties = *secgwext-host-attributes*
⦈ *''secure hypervisor mediates accesses between secure VMs''*
private definition *secgwext-hosts* ≡ [*''hypervisor''*, *''securevm1''*, *''securevm2''*,
''publicvm1'', *''publicvm2''*,
''INET'']

private lemma *dom* (*secgwext-host-attributes*) ⊆ *set* (*secgwext-hosts*)
⟨*proof*⟩
value[*code*] *make-policy* [*SecGwExt-m*] *secgwext-hosts*
⟨ML⟩
end

end

15 Example: Imaginary Factory Network

theory *Imaginary-Factory-Network*
imports ../*TopoS-Impl*
begin

In this theory, we give an an example of an imaginary factory network. The example was chosen to show the interplay of several security invariants and to demonstrate their configuration effort.

The specified security invariants deliberately include some minor specification problems. These problems will be used to demonstrate the inner workings of the algorithms and to visualize why some computed results will deviate from the expected results.

The described scenario is an imaginary factory network. It consists of sensors and actuators in a cyber-physical system. The on-site production units of the factory are completely automated and there are no humans in the production area. Sensors are monitoring the building. The production units are two robots (abbreviated bots) which manufacture the actual goods. The robots are controlled by two control systems.

The network consists of the following hosts which are responsible for monitoring the building.

- Statistics: A server which collects, processes, and stores all data from the sensors.

- SensorSink: A device which receives the data from the PresenceSensor, Webcam, TempSensor, and FireSensor. It sends the data to the Statistics server.
- PresenceSensor: A sensor which detects whether a human is in the building.
- Webcam: A camera which monitors the building indoors.
- TempSensor: A sensor which measures the temperature in the building.
- FireSensor: A sensor which detects fire and smoke.

The following hosts are responsible for the production line.

- MissionControl1: An automation device which drives and controls the robots.
- MissionControl2: An automation device which drives and controls the robots. It contains the logic for a secret production step, carried out only by Robot2.
- Watchdog: Regularly checks the health and technical readings of the robots.
- Robot1: Production robot unit 1.
- Robot2: Production robot unit 2. Does a secret production step.
- AdminPc: A human administrator can log into this machine to supervise or troubleshoot the production.

We model one additional special host.

- INET: A symbolic host which represents all hosts which are not part of this network.

The security policy is defined below.

```
definition policy :: string list-graph where
  policy ≡ ⦇ nodesL = [''Statistics'',
                    ''SensorSink'',
                    ''PresenceSensor'',
                    ''Webcam'',
                    ''TempSensor'',
                    ''FireSensor'',
                    ''MissionControl1'',
                    ''MissionControl2'',
                    ''Watchdog'',
                    ''Robot1'',
                    ''Robot2'',
                    ''AdminPc'',
                    ''INET''],
           edgesL = [(''PresenceSensor'', ''SensorSink''),
                    (''Webcam'', ''SensorSink''),
                    (''TempSensor'', ''SensorSink''),
                    (''FireSensor'', ''SensorSink''),
                    (''SensorSink'', ''Statistics''),
                    (''MissionControl1'', ''Robot1''),
                    (''MissionControl1'', ''Robot2''),
```

("MissionControl2", "Robot2"),
("AdminPc", "MissionControl2"),
("AdminPc", "MissionControl1"),
("Watchdog", "Robot1"),
("Watchdog", "Robot2")
] ⦈

lemma *wf-list-graph policy* ⟨*proof*⟩

⟨*ML*⟩

The idea behind the policy is the following. The sensors on the left can all send their readings in an unidirectional fashion to the sensor sink, which forwards the data to the statistics server. In the production line, on the right, all devices will set up stateful connections. This means, once a connection is established, packet exchange can be bidirectional. This makes sure that the watchdog will receive the health information from the robots, the mission control machines will receive the current state of the robots, and the administrator can actually log into the mission control machines. The policy should only specify who is allowed to set up the connections. We will elaborate on the stateful implementation in `../TopoS_Stateful_Policy.thy` and `../TopoS_Stateful_Policy_Algorithm.thy`.

15.1 Specification of Security Invariants

Several security invariants are specified.

Privacy for employees. The sensors in the building may record any employee. Due to privacy requirements, the sensor readings, processing, and storage of the data is treated with high security levels. The presence sensor does not allow do identify an individual employee, hence produces less critical data, hence has a lower level.

context begin
  private definition *BLP-privacy-host-attributes* ≡ ["Statistics" ↦ 3,
    "SensorSink" ↦ 3,
    "PresenceSensor" ↦ 2, — less critical data
    "Webcam" ↦ 3
    ]
  private lemma *dom* (*BLP-privacy-host-attributes*) ⊆ *set* (*nodesL policy*)
    ⟨*proof*⟩
  definition *BLP-privacy-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-BLPbasic* ⦇
    node-properties = *BLP-privacy-host-attributes* ⦈ "*confidential sensor data*"
end

Secret corporate knowledge and intellectual property: The production process is a corporate trade secret. The mission control devices have the trade secretes in their program. The important and secret step is done by MissionControl2.

context begin
  private definition *BLP-tradesecrets-host-attributes* ≡ ["MissionControl1" ↦ 1,
    "MissionControl2" ↦ 2,
    "Robot1" ↦ 1,
    "Robot2" ↦ 2
    ]

private lemma *dom* (*BLP-tradesecrets-host-attributes*) ⊆ *set* (*nodesL policy*)
⟨*proof*⟩
definition *BLP-tradesecrets-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-BLPbasic* ⦇
node-properties = *BLP-tradesecrets-host-attributes* ⦈ ''*trade secrets*''
end

Note that Invariant 1 and Invariant 2 are two distinct specifications. They specify individual security goals independent of each other. For example, in Invariant 1,''*MissionControl2*'' has the security level ⊥ and in Invariant 2, ''*PresenceSensor*'' has security level ⊥. Consequently, both cannot interact.

Privacy for employees, exporting aggregated data: Monitoring the building while both ensuring privacy of the employees is an important goal for the company. While the presence sensor only collects the single-bit information whether a human is present, the webcam allows to identify individual employees. The data collected by the presence sensor is classified as secret while the data produced by the webcam is top secret. The sensor sink only has the secret security level, hence it is not allowed to process the data generated by the webcam. However, the sensor sink aggregates all data and only distributes a statistical average which does not allow to identify individual employees. It does not store the data over long periods. Therefore, it is marked as trusted and may thus receive the webcam's data. The statistics server, which archives all the data, is considered top secret.

context begin
private definition *BLP-employee-export-host-attributes* ≡
[''*Statistics*'' ↦ ⦇ *security-level* = *3*, *trusted* = *False* ⦈,
''*SensorSink*'' ↦ ⦇ *security-level* = *2*, *trusted* = *True* ⦈,
''*PresenceSensor*'' ↦ ⦇ *security-level* = *2*, *trusted* = *False* ⦈,
''*Webcam*'' ↦ ⦇ *security-level* = *3*, *trusted* = *False* ⦈
]
private lemma *dom* (*BLP-employee-export-host-attributes*) ⊆ *set* (*nodesL policy*)
⟨*proof*⟩
definition *BLP-employee-export-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-BLPtrusted*
⦇
node-properties = *BLP-employee-export-host-attributes* ⦈ ''*employee data* (*privacy*)''

end

Who can access bot2? Robot2 carries out a mission-critical production step. It must be made sure that Robot2 only receives packets from Robot1, the two mission control devices and the watchdog.

context begin
private definition *ACL-bot2-host-attributues* ≡
[''*Robot2*'' ↦ *Master* [''*Robot1*'',
''*MissionControl1*'',
''*MissionControl2*'',
''*Watchdog*''],
''*MissionControl1*'' ↦ *Care*,
''*MissionControl2*'' ↦ *Care*,
''*Watchdog*'' ↦ *Care*
]
private lemma *dom* (*ACL-bot2-host-attributues*) ⊆ *set* (*nodesL policy*)
⟨*proof*⟩
definition *ACL-bot2-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-CommunicationPartners*

$(\!|$*node-properties* = *ACL-bot2-host-attributues* $|\!)$ ''*Robot2 ACL*''

Note that Robot1 is in the access list of Robot2 but it does not have the *Care* attribute. This means, Robot1 can never access Robot2. A tool could automatically detect such inconsistencies and emit a warning. However, a tool should only emit a warning (not an error) because this setting can be desirable.

In our factory, this setting is currently desirable: Three months ago, Robot1 had an irreparable hardware error and needed to be removed from the production line. When removing Robot1 physically, all its host attributes were also deleted. The access list of Robot2 was not changed. It was planned that Robot1 will be replaced and later will have the same access rights again. A few weeks later, a replacement for Robot1 arrived. The replacement is also called Robot1. The new robot arrived neither configured nor tested for the production. After carefully testing Robot1, Robot1 has been given back the host attributes for the other security invariants. Despite the ACL entry of Robot2, when Robot1 was added to the network, because of its missing *Care* attribute, it was not given automatically access to Robot2. This prevented that Robot1 would accidentally impact Robot2 without being fully configured. In our scenario, once Robot1 will be fully configured, tested, and verified, it will be given the *Care* attribute back.

In general, this design choice of the invariant template prevents that a newly added host may inherit access rights due to stale entries in access lists. At the same time, it does not force administrators to clean up their access lists because a host may only be removed temporarily and wants to be given back its access rights later on. Note that managing access lists scales quadratically in the number of hosts. In contrast, the *Care* attribute can be considered as a Boolean flag which allows to temporarily enable or disable the access rights of a host locally without touching the carefully constructed access lists of other hosts. It also prevents that new hosts which have the name of hosts removed long ago (but where stale access rights were not cleaned up) accidentally inherit their access rights.

end

Hierarchy of fab robots: The production line is designed according to a strict command hierarchy. On top of the hierarchy are control terminals which allow a human operator to intervene and supervise the production process. On the level below, one distinguishes between supervision devices and control devices. The watchdog is a typical supervision device whereas the mission control devices are control devices. Directly below the control devices are the robots. This is the structure that is necessary for the example. However, the company defined a few more sub-departments for future use. The full domain hierarchy tree is visualized below.

Apart from the watchdog, only the following linear part of the tree is used: ''*Robots*'' $\sqsubseteq$ ''*ControlDevices*'' $\sqsubseteq$ ''*ControlTerminal*''. Because the watchdog is in a different domain, it needs a trust level of 1 to access the robots it is monitoring.

context begin
private definition *DomainHierarchy-host-attributes* ≡
[(''*MissionControl1*'',
DN (''*ControlTerminal*''−−''*ControlDevices*''−−*Leaf*, *0*)),
(''*MissionControl2*'',
DN (''*ControlTerminal*''−−''*ControlDevices*''−−*Leaf*, *0*)),
(''*Watchdog*'',
DN (''*ControlTerminal*''−−''*Supervision*''−−*Leaf*, *1*)),
(''*Robot1*'',

```
          DN (''ControlTerminal''--''ControlDevices''--''Robots''--Leaf, 0)),
        (''Robot2'',
          DN (''ControlTerminal''--''ControlDevices''--''Robots''--Leaf, 0)),
        (''AdminPc'',
          DN (''ControlTerminal''--Leaf, 0))
        ]
  private lemma dom (map-of DomainHierarchy-host-attributes) ⊆ set (nodesL policy)
    ⟨proof⟩

  lemma DomainHierarchyNG-sanity-check-config
    (map snd DomainHierarchy-host-attributes)
        (
        Department ''ControlTerminal'' [
          Department ''ControlDevices'' [
            Department ''Robots'' [],
            Department ''OtherStuff'' [],
            Department ''ThirdSubDomain'' []
          ],
          Department ''Supervision'' [
            Department ''S1'' [],
            Department ''S2'' []
          ]
        ]) ⟨proof⟩

  definition Control-hierarchy-m ≡ new-configured-list-SecurityInvariant
                                    SINVAR-LIB-DomainHierarchyNG
                                    (| node-properties = map-of DomainHierarchy-host-attributes |)
                                    ''Production device hierarchy''
end
```

Sensor Gateway: The sensors should not communicate among each other; all accesses must be mediated by the sensor sink.

```
context begin
  private definition PolEnforcePoint-host-attributes ≡
        [''SensorSink'' ↦ PolEnforcePoint,
         ''PresenceSensor'' ↦ DomainMember,
         ''Webcam'' ↦ DomainMember,
         ''TempSensor'' ↦ DomainMember,
         ''FireSensor'' ↦ DomainMember
        ]
  private lemma dom PolEnforcePoint-host-attributes ⊆ set (nodesL policy)
    ⟨proof⟩
  definition PolEnforcePoint-m ≡ new-configured-list-SecurityInvariant
                                  SINVAR-LIB-PolEnforcePointExtended
                                    (| node-properties = PolEnforcePoint-host-attributes |)
                                  ''sensor slaves''
end
```

Production Robots are an information sink: The actual control program of the robots is a corporate trade secret. The control commands must not leave the robots. Therefore, they are declared information sinks. In addition, the control command must not leave the mission control devices. However, the two devices could possibly interact to synchronize and they must send their commands to the robots. Therefore, they are labeled as sink pools.

```
context begin
  private definition SinkRobots-host-attributes ≡
              [''MissionControl1'' ↦ SinkPool,
               ''MissionControl2'' ↦ SinkPool,
               ''Robot1'' ↦ Sink,
               ''Robot2'' ↦ Sink
               ]
  private lemma dom SinkRobots-host-attributes ⊆ set (nodesL policy)
    ⟨proof⟩
  definition SinkRobots-m ≡ new-configured-list-SecurityInvariant
                              SINVAR-LIB-Sink
                                (| node-properties = SinkRobots-host-attributes |)
                           ''non−leaking production units''
end
```

Subnet of the fab: The sensors, including their sink and statistics server are located in their own subnet and must not be accessible from elsewhere. Also, the administrator's PC is in its own subnet. The production units (mission control and robots) are already isolated by the DomainHierarchy and are not added to a subnet explicitly.

```
context begin
  private definition Subnets-host-attributes ≡
              [''Statistics'' ↦ Subnet 1,
               ''SensorSink'' ↦ Subnet 1,
               ''PresenceSensor'' ↦ Subnet 1,
               ''Webcam'' ↦ Subnet 1,
               ''TempSensor'' ↦ Subnet 1,
               ''FireSensor'' ↦ Subnet 1,
               ''AdminPc'' ↦ Subnet 4
               ]
  private lemma dom Subnets-host-attributes ⊆ set (nodesL policy)
    ⟨proof⟩
  definition Subnets-m ≡ new-configured-list-SecurityInvariant
                              SINVAR-LIB-Subnets
                                (| node-properties = Subnets-host-attributes |)
                         ''network segmentation''
end
```

Access Gateway for the Statistics server: The statistics server is further protected from external accesses. Another, smaller subnet is defined with the only member being the statistics server. The only way it may be accessed is via that sensor sink.

```
context begin
  private definition SubnetsInGW-host-attributes ≡
              [''Statistics'' ↦ Member,
               ''SensorSink'' ↦ InboundGateway
               ]
  private lemma dom SubnetsInGW-host-attributes ⊆ set (nodesL policy)
    ⟨proof⟩
  definition SubnetsInGW-m ≡ new-configured-list-SecurityInvariant
                              SINVAR-LIB-SubnetsInGW
                                (| node-properties = SubnetsInGW-host-attributes |)
                           ''Protectting statistics srv''
end
```

NonInterference (for the sake of example): The fire sensor is managed by an external company and has a built-in GSM module to call the fire fighters in case of an emergency. This additional, out-of-band connectivity is not modeled. However, the contract defines that the company's administrator must not interfere in any way with the fire sensor.

context begin
private definition *NonInterference-host-attributes* ≡
[''*Statistics*'' ↦ *Unrelated*,
''*SensorSink*'' ↦ *Unrelated*,
''*PresenceSensor*'' ↦ *Unrelated*,
''*Webcam*'' ↦ *Unrelated*,
''*TempSensor*'' ↦ *Unrelated*,
''*FireSensor*'' ↦ *Interfering*, — (!)
''*MissionControl1*'' ↦ *Unrelated*,
''*MissionControl2*'' ↦ *Unrelated*,
''*Watchdog*'' ↦ *Unrelated*,
''*Robot1*'' ↦ *Unrelated*,
''*Robot2*'' ↦ *Unrelated*,
''*AdminPc*'' ↦ *Interfering*, — (!)
''*INET*'' ↦ *Unrelated*
]
private lemma *dom NonInterference-host-attributes* ⊆ *set* (*nodesL policy*)
⟨*proof*⟩
definition *NonInterference-m* ≡ *new-configured-list-SecurityInvariant SINVAR-LIB-NonInterference*
(| *node-properties* = *NonInterference-host-attributes* |)
''*for the sake of an acdemic example*!''
end

As discussed, this invariant is very strict and rather theoretical. It is not ENF-structured and may produce an exponential number of offending flows. Therefore, we exclude it by default from our algorithms.

definition *invariants* ≡ [*BLP-privacy-m*, *BLP-tradesecrets-m*, *BLP-employee-export-m*,
ACL-bot2-m, *Control-hierarchy-m*,
PolEnforcePoint-m, *SinkRobots-m*, *Subnets-m*, *SubnetsInGW-m*]

We have excluded *NonInterference-m* because of its infeasible runtime.

lemma *length invariants* = *9* ⟨*proof*⟩

15.2 Policy Verification

The given policy fulfills all the specified security invariants. Also with *NonInterference-m*, the policy fulfills all security invariants.

lemma *all-security-requirements-fulfilled* (*NonInterference-m#invariants*) *policy* ⟨*proof*⟩
⟨*ML*⟩

definition *make-policy* :: (*'a SecurityInvariant*) *list* ⇒ *'a list* ⇒ *'a list-graph* **where**
make-policy sinvars Vs ≡ *generate-valid-topology sinvars* (|*nodesL* = *Vs*, *edgesL* = *List.product Vs Vs* |)

definition *make-policy-efficient* :: (*'a SecurityInvariant*) *list* ⇒ *'a list* ⇒ *'a list-graph* **where**
make-policy-efficient sinvars Vs ≡ *generate-valid-topology-some sinvars* (|*nodesL* = *Vs*, *edgesL* = *List.product Vs Vs* |)

The question, "how good are the specified security invariants?" remains. Therefore, we use the algorithm from *make-policy* to generate a policy. Then, we will compare our policy with the automatically generated one. If we exclude the NonInterference invariant from the policy construction, we know that the resulting policy must be maximal. Therefore, the computed policy reflects the view of the specified security invariants. By maximality of the computed policy and monotonicity, we know that our manually specified policy must be a subset of the computed policy. This allows to compare the manually-specified policy to the policy implied by the security invariants: If there are too many flows which are allowed according to the computed policy but which are not in our manually-specified policy, we can conclude that our security invariants are not strict enough.

value[*code*] *make-policy invariants* (*nodesL policy*)
lemma *make-policy invariants* (*nodesL policy*) =
(|*nodesL* =
[''*Statistics*'', ''*SensorSink*'', ''*PresenceSensor*'', ''*Webcam*'', ''*TempSensor*'',
''*FireSensor*'', ''*MissionControl1*'', ''*MissionControl2*'', ''*Watchdog*'', ''*Robot1*'',
''*Robot2*'', ''*AdminPc*'', ''*INET*''],
edgesL =
[(''*Statistics*'', ''*Statistics*''), (''*SensorSink*'', ''*Statistics*''),
(''*SensorSink*'', ''*SensorSink*''), (''*SensorSink*'', ''*Webcam*''),
(''*PresenceSensor*'', ''*SensorSink*''), (''*PresenceSensor*'', ''*PresenceSensor*''),
(''*Webcam*'', ''*SensorSink*''), (''*Webcam*'', ''*Webcam*''),
(''*TempSensor*'', ''*SensorSink*''), (''*TempSensor*'', ''*TempSensor*''),
(''*TempSensor*'', ''*INET*''), (''*FireSensor*'', ''*SensorSink*''),
(''*FireSensor*'', ''*FireSensor*''), (''*FireSensor*'', ''*INET*''),
(''*MissionControl1*'', ''*MissionControl1*''),
(''*MissionControl1*'', ''*MissionControl2*''), (''*MissionControl1*'', ''*Robot1*''),
(''*MissionControl1*'', ''*Robot2*''), (''*MissionControl2*'', ''*MissionControl2*''),
(''*MissionControl2*'', ''*Robot2*''), (''*Watchdog*'', ''*MissionControl1*''),
(''*Watchdog*'', ''*MissionControl2*''), (''*Watchdog*'', ''*Watchdog*''),
(''*Watchdog*'', ''*Robot1*''), (''*Watchdog*'', ''*Robot2*''), (''*Watchdog*'', ''*INET*''),
(''*Robot1*'', ''*Robot1*''), (''*Robot2*'', ''*Robot2*''), (''*AdminPc*'', ''*MissionControl1*''),
(''*AdminPc*'', ''*MissionControl2*''), (''*AdminPc*'', ''*Watchdog*''),
(''*AdminPc*'', ''*Robot1*''), (''*AdminPc*'', ''*AdminPc*''), (''*AdminPc*'', ''*INET*''),
(''*INET*'', ''*INET*'')]|) ⟨*proof*⟩

Additional flows which would be allowed but which are not in the policy

lemma *set* [*e* ← *edgesL* (*make-policy invariants* (*nodesL policy*)). *e* ∉ *set* (*edgesL policy*)] =
set [(*v*,*v*). *v* ← (*nodesL policy*)] ∪
set [(''*SensorSink*'', ''*Webcam*''),
(''*TempSensor*'', ''*INET*''),
(''*FireSensor*'', ''*INET*''),
(''*MissionControl1*'', ''*MissionControl2*''),
(''*Watchdog*'', ''*MissionControl1*''),
(''*Watchdog*'', ''*MissionControl2*''),
(''*Watchdog*'', ''*INET*''),
(''*AdminPc*'', ''*Watchdog*''),
(''*AdminPc*'', ''*Robot1*''),
(''*AdminPc*'', ''*INET*'')] ⟨*proof*⟩

We visualize this comparison below. The solid edges correspond to the manually-specified policy. The dotted edges correspond to the flow which would be additionally permitted by the computed policy.

⟨*ML*⟩

The comparison reveals that the following flows would be additionally permitted. We will discuss whether this is acceptable or if the additional permission indicates that we probably forgot to specify an additional security goal.

- All reflexive flows, i.e. all host can communicate with themselves. Since each host in the policy corresponds to one physical entity, there is no need to explicitly prohibit or allow in-host communication.
- The *''SensorSink''* may access the *''Webcam''*. Both share the same security level, there is no problem with this possible information flow. Technically, a bi-directional connection may even be desirable, since this allows the sensor sink to influence the video stream, e.g. request a lower bit rate if it is overloaded.
- Both the *''TempSensor''* and the *''FireSensor''* may access the Internet. No security levels or other privacy concerns are specified for them. This may raise the question whether this data is indeed public. It is up to the company to decide that this data should also be considered confidential.
- *''MissionControl1''* can send to *''MissionControl2''*. This may be desirable since it was stated anyway that the two may need to cooperate. Note that the opposite direction is definitely prohibited since the critical and secret production step only known to *''MissionControl2''* must not leak.
- The *''Watchdog''* may access *''MissionControl1''*, *''MissionControl2''*, and the *''INET''*. While it may be acceptable that the watchdog which monitors the robots may also access the control devices, it should raise a concern that the watchdog may freely send data to the Internet. Indeed, the watchdog can access devices which have corporate trade secrets stored but it was never specified that the watchdog should be treated confidentially. Note that in the current setting, the trade secrets will never leave the robots. This is because the policy only specifies a unidirectional information flow from the watchdog to the robots; the robots will not leak any information back to the watchdog. This also means that the watchdog cannot actually monitor the robots. Later, when implementing the scenario, we will see that the simple, hand-waving argument "the watchdog connects to the robots and the robots send back their data over the established connection" will not work because of this possible information leak.
- The *''AdminPc''* is allowed to access the *''Watchdog''*, *''Robot1''*, and the *''INET''*. Since this machine is trusted anyway, the company does not see a problem with this.

without *NonInterference-m*

lemma *all-security-requirements-fulfilled invariants* (*make-policy invariants* (*nodesL policy*)) ⟨*proof*⟩

Side note: what if we exclude subnets?

⟨*ML*⟩

15.3 About NonInterference

The NonInterference template was deliberately selected for our scenario as one of the 'problematic' and rather theoretical invariants. Our framework allows to specify almost arbitrary invariant templates. We concluded that all non-ENF-structured invariants which may produce an exponential number of offending flows are problematic for practical use. This includes "Comm. With" (`../Security_Invariants/SINVAR_ACLcommunicateWith.thy`), "Not Comm. With" (`../Security_Invariants/SINVAR_ACLnotCommunicateWith.thy`), Dependability (`../Security_Invariants/SINVAR_Dependability.thy`), and NonInterference (`../Security_Invariants/SINVAR_NonInterference.thy`). In this section, we discuss the consequences of the NonInterference invariant for automated policy construction. We will conclude that, though we can solve all technical challenges, said invariants are —due to their inherent ambiguity— not very well suited for automated policy construction.

The computed maximum policy does not fulfill invariant 10 (NonInterference). This is because the fire sensor and the administrator's PC may be indirectly connected over the Internet.

lemma ¬ *all-security-requirements-fulfilled* (*NonInterference-m#invariants*) (*make-policy invariants* (*nodesL policy*)) ⟨*proof*⟩

Since the NonInterference template may produce an exponential number of offending flows, it is infeasible to try our automated policy construction algorithm with it. We have tried to do so on a machine with 128GB of memory but after a few minutes, the computation ran out of memory. On said machine, we were unable to run our policy construction algorithm with the NonInterference invariant for more that five hosts.

Algorithm *make-policy-efficient* improves the policy construction algorithm. The new algorithm instantly returns a solution for this scenario with a very small memory footprint.

The more efficient algorithm does not need to construct the complete set of offending flows

value[*code*] *make-policy-efficient* (*invariants*@[*NonInterference-m*]) (*nodesL policy*)
value[*code*] *make-policy-efficient* (*NonInterference-m#invariants*) (*nodesL policy*)

lemma *make-policy-efficient* (*invariants*@[*NonInterference-m*]) (*nodesL policy*) =
make-policy-efficient (*NonInterference-m#invariants*) (*nodesL policy*) ⟨*proof*⟩

But *NonInterference-m* insists on removing something, which would not be necessary.

lemma *make-policy invariants* (*nodesL policy*) ≠ *make-policy-efficient* (*NonInterference-m#invariants*) (*nodesL policy*) ⟨*proof*⟩

lemma *set* (*edgesL* (*make-policy-efficient* (*NonInterference-m#invariants*) (*nodesL policy*)))
⊆
set (*edgesL* (*make-policy invariants* (*nodesL policy*))) ⟨*proof*⟩

This is what it wants to be gone.

lemma [*e* ← *edgesL* (*make-policy invariants* (*nodesL policy*)).
e ∉ *set* (*edgesL* (*make-policy-efficient* (*NonInterference-m#invariants*) (*nodesL policy*)))]
=
[(''*AdminPc*'', ''*MissionControl1*''), (''*AdminPc*'', ''*MissionControl2*''),
(''*AdminPc*'', ''*Watchdog*''), (''*AdminPc*'', ''*Robot1*''), (''*AdminPc*'', ''*INET*'')]
⟨*proof*⟩

lemma [*e* ← *edgesL* (*make-policy invariants* (*nodesL policy*)).
e ∉ *set* (*edgesL* (*make-policy-efficient* (*NonInterference-m#invariants*) (*nodesL policy*)))]
=

[$e \leftarrow$ *edgesL* (*make-policy invariants* (*nodesL policy*)). *fst* $e = ''AdminPc'' \wedge$ *snd* $e \neq ''AdminPc'']$
⟨*proof*⟩
⟨*ML*⟩

However, it is an inherent property of the NonInterferance template (and similar templates), that the set of offending flows is not uniquely defined. Consequently, since several solutions are possible, even our new algorithm may not be able to compute one maximum solution. It would be possible to construct some maximal solution, however, this would require to enumerate all offending flows, which is infeasible. Therefore, our algorithm can only return some (valid but probably not maximal) solution for non-END-structured invariants.

As a human, we know the scenario and the intention behind the policy. Probably, the best solution for policy construction with the NonInterferance property would be to restrict outgoing edges from the fire sensor. If we consider the policy above which was constructed without NonInterference, if we cut off the fire sensor from the Internet, we get a valid policy for the NonInterference property. Unfortunately, an algorithm does not have the information of which flows we would like to cut first and the algorithm needs to make some choice. In this example, the algorithm decides to isolate the administrator's PC from the rest of the world. This is also a valid solution. We could change the order of the elements to tell the algorithm which edges we would rather sacrifice than others. This may help but requires some additional input. The author personally prefers to construct only maximum policies with Φ-structured invariants and afterwards fix the policy manually for the remaining non-Φ-structured invariants. Though our new algorithm gives better results and returns instantly, the very nature of invariant templates with an exponential number of offending flows tells that these invariants are problematic for automated policy construction.

15.4 Stateful Implementation

In this section, we will implement the policy and deploy it in a network. As the scenario description stated, all devices in the production line should establish stateful connections which allows – once the connection is established – packets to travel in both directions. This is necessary for the watchdog, the mission control devices, and the administrator's PC to actually perform their task.

We compute a stateful implementation. Below, the stateful implementation is visualized. It consists of the policy as visualized above. In addition, dotted edges visualize where answer packets are permitted.

definition *stateful-policy = generate-valid-stateful-policy-IFSACS policy invariants*
lemma *stateful-policy =*
⦇*hostsL = nodesL policy*,
  flows-fixL = edgesL policy,
  flows-stateL =
    [(*''Webcam''*, *''SensorSink''*),
     (*''SensorSink''*, *''Statistics''*)]⦈ ⟨*proof*⟩

⟨*ML*⟩

As can be seen, only the flows (*''Webcam''*, *''SensorSink''*) and (*''SensorSink''*, *''Statistics''*) are allowed to be stateful. This setup cannot be practically deployed because the watchdog, the mission control devices, and the administrator's PC also need to set up stateful connections. Previous section's discussion already hinted at this problem. The reason why the desired stateful connections are not permitted is due to information leakage. In detail: *BLP-tradesecrets-m* and *SinkRobots-m* are responsible. Both invariants prevent that any data leaves the robots and the mission control devices. To verify this suspicion, the two invariants are removed and the stateful flows are computed again. The result visualized is below.

lemma *generate-valid-stateful-policy-IFSACS policy*
[BLP-privacy-m, BLP-employee-export-m,
ACL-bot2-m, Control-hierarchy-m,
PolEnforcePoint-m, Subnets-m, SubnetsInGW-m] =
(|hostsL = nodesL policy,
flows-fixL = edgesL policy,
flows-stateL =
[("Webcam", "SensorSink"),
("SensorSink", "Statistics"),
("MissionControl1", "Robot1"),
("MissionControl1", "Robot2"),
("MissionControl2", "Robot2"),
("AdminPc", "MissionControl2"),
("AdminPc", "MissionControl1"),
("Watchdog", "Robot1"),
("Watchdog", "Robot2")]|) ⟨*proof*⟩

This stateful policy could be transformed into a fully functional implementation. However, there would be no security invariants specified which protect the trade secrets. Without those two invariants, the invariant specification is too permissive. For example, if we recompute the maximum policy, we can see that the robots and mission control can leak any data to the Internet. Even without the maximum policy, in the stateful policy above, it can be seen that MissionControl1 can exfiltrate information from robot 2, once it establishes a stateful connection.

Without the two invariants, the security goals are way too permissive!

lemma *set [e ← edgesL (make-policy [BLP-privacy-m, BLP-employee-export-m,*
ACL-bot2-m, Control-hierarchy-m,
PolEnforcePoint-m, Subnets-m, SubnetsInGW-m] (nodesL policy)). e ∉ set (edgesL policy)] =
set [(v,v). v ← (nodesL policy)] ∪
set [("SensorSink", "Webcam"),
("TempSensor", "INET"),
("FireSensor", "INET"),
("MissionControl1", "MissionControl2"),
("Watchdog", "MissionControl1"),
("Watchdog", "MissionControl2"),
("Watchdog", "INET"),
("AdminPc", "Watchdog"),
("AdminPc", "Robot1"),
("AdminPc", "INET")] ∪
set [("MissionControl1", "INET"),
("MissionControl2", "MissionControl1"),
("MissionControl2", "Robot1"),
("MissionControl2", "INET"),
("Robot1", "INET"),
("Robot2", "Robot1"),
("Robot2", "INET")] ⟨*proof*⟩

⟨*ML*⟩

Therefore, the two invariants are not removed but repaired. The goal is to allow the watchdog, administrator's pc, and the mission control devices to set up stateful connections without leaking corporate trade secrets to the outside.

First, we repair *BLP-tradesecrets-m*. On the one hand, the watchdog should be able to send packets both *''Robot1''* and *''Robot2''*. *''Robot1''* has a security level of *1* and *''Robot2''* has a security level of *2*. Consequently, in order to be allowed to send packets to both, *''Watchdog''* must have a security level not higher than *1*. On the other hand, the *''Watchdog''* should be able to receive packets from both. By the same argument, it must have a security level of at least *2*. Consequently, it is impossible to express the desired meaning in the BLP basic template. There are only two solutions to the problem: Either the company installs one watchdog for each security level, or the watchdog must be trusted. We decide for the latter option and upgrade the template to the Bell LaPadula model with trust. We define the watchdog as trusted with a security level of *1*. This means, it can receive packets from and send packets to both robots but it cannot leak information to the outside world. We do the same for the *''AdminPc''*.

Then, we repair *SinkRobots-m*. We realize that the following set set of hosts forms one big pool of devices which must all somehow interact but where information must not leave the pool: The administrator's PC, the mission control devices, the robots, and the watchdog. Therefore, all those devices are configured to be in the same *SinkPool*.

```
definition invariants-tuned ≡ [BLP-privacy-m, BLP-employee-export-m,
      ACL-bot2-m, Control-hierarchy-m,
      PolEnforcePoint-m, Subnets-m, SubnetsInGW-m,
      new-configured-list-SecurityInvariant SINVAR-LIB-Sink
        (| node-properties = [''MissionControl1'' ↦ SinkPool,
                              ''MissionControl2'' ↦ SinkPool,
                              ''Robot1'' ↦ SinkPool,
                              ''Robot2'' ↦ SinkPool,
                              ''Watchdog'' ↦ SinkPool,
                              ''AdminPc'' ↦ SinkPool
                              ] |)
        ''non−leaking production units'',
      new-configured-list-SecurityInvariant SINVAR-LIB-BLPtrusted
        (| node-properties = [''MissionControl1'' ↦ (| security-level = 1, trusted = False |),
                              ''MissionControl2'' ↦ (| security-level = 2, trusted = False |),
                              ''Robot1'' ↦ (| security-level = 1, trusted = False |),
                              ''Robot2'' ↦ (| security-level = 2, trusted = False |),
                              ''Watchdog'' ↦ (| security-level = 1, trusted = True |),
                              — trust because bot2 must send to it. security-level 1 to interact with
bot 1
                              ''AdminPc'' ↦ (| security-level = 1, trusted = True |)
                              ] |)
        ''trade secrets''
      ]
```

lemma *all-security-requirements-fulfilled invariants-tuned policy* ⟨*proof*⟩

definition *stateful-policy-tuned = generate-valid-stateful-policy-IFSACS policy invariants-tuned*

The computed stateful policy is visualized below.

lemma *stateful-policy-tuned*
$=$
$(\!|$*hostsL* $=$ *nodesL policy*,
flows-fixL $=$ *edgesL policy*,
flows-stateL $=$
[(''*Webcam*'', ''*SensorSink*''),
(''*SensorSink*'', ''*Statistics*''),
(''*MissionControl1*'', ''*Robot1*''),
(''*MissionControl2*'', ''*Robot2*''),
(''*AdminPc*'', ''*MissionControl2*''),
(''*AdminPc*'', ''*MissionControl1*''),
(''*Watchdog*'', ''*Robot1*''),
(''*Watchdog*'', ''*Robot2*'')]$|\!)$ ⟨*proof*⟩

We even get a better (i.e. stricter) maximum policy

lemma *set* (*edgesL* (*make-policy invariants-tuned* (*nodesL policy*))) $\subset$
set (*edgesL* (*make-policy invariants* (*nodesL policy*))) ⟨*proof*⟩
lemma *set* [*e* $\leftarrow$ *edgesL* (*make-policy invariants-tuned* (*nodesL policy*)). *e* $\notin$ *set* (*edgesL policy*)] $=$
set [(*v*,*v*). *v* $\leftarrow$ (*nodesL policy*)] $\cup$
set [(''*SensorSink*'', ''*Webcam*''),
(''*TempSensor*'', ''*INET*''),
(''*FireSensor*'', ''*INET*''),
(''*MissionControl1*'', ''*MissionControl2*''),
(''*Watchdog*'', ''*MissionControl1*''),
(''*Watchdog*'', ''*MissionControl2*''),
(''*AdminPc*'', ''*Watchdog*''),
(''*AdminPc*'', ''*Robot1*'')] ⟨*proof*⟩

It can be seen that all connections which should be stateful are now indeed stateful. In addition, it can be seen that MissionControl1 cannot set up a stateful connection to Bot2. This is because MissionControl1 was never declared a trusted device and the confidential information in MissionControl2 and Robot2 must not leak.

The improved invariant definition even produces a better (i.e. stricter) maximum policy.

15.5 Iptables Implementation

firewall – classical use case

⟨*ML*⟩

Using, https://github.com/diekmann/Iptables_Semantics, the iptables ruleset is indeed correct.

end

References

[1] C. Diekmann, L. Hupel, and G. Carle. Directed Security Policies: A Stateful Network Implementation. In J. Pang and Y. Liu, editors, *Engineering Safety and Security Systems*, volume 150 of *Electronic Proceedings in Theoretical Computer Science*, pages 20–34, Singapore, May 2014. Open Publishing Association.

[2] C. Diekmann, A. Korsten, and G. Carle. Demonstrating *topoS*: Theorem-Prover-Based Synthesis of Secure Network Configurations. In *2nd International Workshop on Management of SDN and NFV Systems, manSDN/NFV*, Barcelona, Spain, Nov. 2015.

[3] C. Diekmann, S.-A. Posselt, H. Niedermayer, H. Kinkelin, O. Hanka, and G. Carle. Verifying Security Policies using Host Attributes. In *FORTE – 34th IFIP International Conference on Formal Techniques for Distributed Objects, Components and Systems*, Berlin, Germany, June 2014.