Network Security Policy Verification

Cornelius Diekmann

August 16, 2018

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

1 A type for vertices 5
2 Security Invariants 6
   2.1 Security Invariants with secure auto-completion of host attribute mappings 8
   2.2 Information Flow Security and Access Control 10
   2.3 Information Flow Security Strategy (IFS) 10
   2.4 Access Control Strategy (ACS) 11
3 SecurityInvariant Instantiation Helpers 12
   3.1 Offending Flows Not Empty Helper Lemmata 13
   3.2 Monotonicity of offending flows 16
4 Special Structures of Security Invariants 18
   4.1 Simple Edges Normal Form (ENF) 18
      4.1.1 Offending Flows 19
      4.1.2 Lemmata 20
      4.1.3 Instance Helper 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2.10</td>
<td>Example</td>
<td>55</td>
</tr>
<tr>
<td>6.3</td>
<td>SecurityInvariant PolEnforcePointExtended</td>
<td>56</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Preliminaries</td>
<td>56</td>
</tr>
<tr>
<td>6.3.2</td>
<td>ENF</td>
<td>57</td>
</tr>
<tr>
<td>6.3.3</td>
<td>SecurityInvariant PolEnforcePointExtended List Implementation</td>
<td>57</td>
</tr>
<tr>
<td>6.3.4</td>
<td>PolEnforcePoint packing</td>
<td>58</td>
</tr>
<tr>
<td>6.4</td>
<td>SecurityInvariant Sink (IFS)</td>
<td>59</td>
</tr>
<tr>
<td>6.4.1</td>
<td>Preliminaries</td>
<td>59</td>
</tr>
<tr>
<td>6.4.2</td>
<td>ENF</td>
<td>60</td>
</tr>
<tr>
<td>6.4.3</td>
<td>SecurityInvariant Sink (IFS) List Implementation</td>
<td>60</td>
</tr>
<tr>
<td>6.4.4</td>
<td>Sink packing</td>
<td>61</td>
</tr>
<tr>
<td>6.5</td>
<td>SecurityInvariant SubnetsInGW</td>
<td>62</td>
</tr>
<tr>
<td>6.5.1</td>
<td>Preliminaries</td>
<td>62</td>
</tr>
<tr>
<td>6.5.2</td>
<td>ENF</td>
<td>62</td>
</tr>
<tr>
<td>6.5.3</td>
<td>SecurityInvariant SubnetsInGw List Implementation</td>
<td>63</td>
</tr>
<tr>
<td>6.5.4</td>
<td>SubnetsInGW packing</td>
<td>64</td>
</tr>
<tr>
<td>6.6</td>
<td>SecurityInvariant CommunicationPartners</td>
<td>65</td>
</tr>
<tr>
<td>6.6.1</td>
<td>Preliminaries</td>
<td>65</td>
</tr>
<tr>
<td>6.6.2</td>
<td>ENRnr</td>
<td>66</td>
</tr>
<tr>
<td>6.6.3</td>
<td>SecurityInvariant CommunicationPartners List Implementation</td>
<td>67</td>
</tr>
<tr>
<td>6.6.4</td>
<td>CommunicationPartners packing</td>
<td>67</td>
</tr>
<tr>
<td>6.7</td>
<td>SecurityInvariant NoRefl</td>
<td>68</td>
</tr>
<tr>
<td>6.7.1</td>
<td>Preliminaries</td>
<td>68</td>
</tr>
<tr>
<td>6.7.2</td>
<td>SecurityInvariant NoRefl List Implementation</td>
<td>69</td>
</tr>
<tr>
<td>6.7.3</td>
<td>PolEnforcePoint packing</td>
<td>70</td>
</tr>
<tr>
<td>6.7.4</td>
<td>SecurityInvariant Tainting List Implementation</td>
<td>71</td>
</tr>
<tr>
<td>6.7.5</td>
<td>Tainting packing</td>
<td>71</td>
</tr>
<tr>
<td>6.7.6</td>
<td>Example</td>
<td>72</td>
</tr>
<tr>
<td>6.7.7</td>
<td>SecurityInvariant Tainting with Trust List Implementation</td>
<td>72</td>
</tr>
<tr>
<td>6.7.8</td>
<td>TaintingTrusted packing</td>
<td>73</td>
</tr>
<tr>
<td>6.7.9</td>
<td>Example</td>
<td>74</td>
</tr>
<tr>
<td>6.8</td>
<td>SecurityInvariant Dependability</td>
<td>75</td>
</tr>
<tr>
<td>6.8.1</td>
<td>SecurityInvariant Dependability List Implementation</td>
<td>76</td>
</tr>
<tr>
<td>6.8.2</td>
<td>Dependability packing</td>
<td>77</td>
</tr>
<tr>
<td>6.9</td>
<td>SecurityInvariant NonInterference</td>
<td>78</td>
</tr>
<tr>
<td>6.9.1</td>
<td>monotonic and preliminaries</td>
<td>79</td>
</tr>
<tr>
<td>6.9.2</td>
<td>SecurityInvariant NonInterference List Implementation</td>
<td>80</td>
</tr>
<tr>
<td>6.9.3</td>
<td>NonInterference packing</td>
<td>82</td>
</tr>
<tr>
<td>6.10</td>
<td>SecurityInvariant ACLcommunicateWith</td>
<td>82</td>
</tr>
<tr>
<td>6.11</td>
<td>SecurityInvariant ACLnotCommunicateWith</td>
<td>83</td>
</tr>
<tr>
<td>6.11.1</td>
<td>SecurityInvariant ACLnotCommunicateWith List Implementation</td>
<td>84</td>
</tr>
<tr>
<td>6.11.2</td>
<td>packing</td>
<td>85</td>
</tr>
<tr>
<td>6.11.3</td>
<td>List Implementation</td>
<td>86</td>
</tr>
<tr>
<td>6.11.4</td>
<td>packing</td>
<td>86</td>
</tr>
<tr>
<td>6.12</td>
<td>SecurityInvariant Dependability-norefl</td>
<td>87</td>
</tr>
<tr>
<td>6.12.1</td>
<td>SecurityInvariant Dependability norefl List Implementation</td>
<td>88</td>
</tr>
<tr>
<td>6.12.2</td>
<td>packing</td>
<td>89</td>
</tr>
</tbody>
</table>
# Table of Contents

7 Composition Theory 90  
7.1 Reusing Lemmata .............................................. 92  
7.2 Algorithms ....................................................... 93  
7.3 Lemmata .......................................................... 94  
7.4 generate valid topology ......................................... 94  
7.5 More Lemmata ..................................................... 96  

8 Stateful Policy 98  
8.1 Summarizing the important theorems .......................... 102  

9 Composition Theory – List Implementation 103  
9.1 Generating instantiated (configured) network security invariants ................................................. 103  
9.2 About security invariants ....................................... 104  
9.3 Calculating offending flows ..................................... 104  
9.4 Accessors ........................................................... 105  
9.5 All security requirements fulfilled .............................. 106  
9.6 generate valid topology ......................................... 106  
9.7 generate valid topology ......................................... 106  

10 Stateful Policy – Algorithm 107  
10.1 Some unimportant lemmata .................................... 107  
10.2 Sketch for generating a stateful policy from a simple directed policy ............................................ 108  

11 Stateful Policy – List Implementation 113  
11.1 Algorithms ......................................................... 114  
11.1.1 Meta SecurityInvariant: System Boundaries .................. 116  

12 ML Visualization Interface 118  
12.1 Utility Functions ................................................. 118  

13 Network Security Policy Verification 119  

14 A small Tutorial 119  
14.1 Policy ............................................................. 119  
14.2 Security Invariants ............................................. 120  
14.3 A stateful implementation ..................................... 122  

15 Example: Imaginary Factory Network 128  
15.1 Specification of Security Invariants ............................ 130  
15.2 Policy Verification .............................................. 135  
15.3 About NonInterference .......................................... 137  
15.4 Stateful Implementation ........................................ 139  
15.5 Iptables Implementation ....................................... 142
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.

--- We need at least some vertices available for a graph ...

```plaintext
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.

```plaintext
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
```


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.

— 'v is the type of the nodes in the graph (hosts in the network). 'a is the type of the host attributes.

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
— Offending Flows definitions:

**definition** is-offending-flows::('v × 'v) set ⇒ 'v graph ⇒ ('v ⇒ 'a) ⇒ bool where
is-offending-flows f G nP ≡ ¬ sinvar G nP ∧ sinvar (delete-edges G f) nP

— Above definition is not minimal:

**definition** is-offending-flows-min-set::('v × 'v) set ⇒ 'v graph ⇒ ('v ⇒ 'a) ⇒ bool where
is-offending-flows-min-set f G nP ≡ is-offending-flows f G nP ∧ (∀ (e1, e2) ∈ f. ¬ sinvar (add-edge e1 e2 (delete-edges G f)) nP)

— The set of all offending flows,

**definition** set-offending-flows::'v graph ⇒ ('v ⇒ 'a) ⇒ ('v × 'v) set set ⇒ bool where
set-offending-flows G nP = { F. F ⊆ (edges G) ∧ is-offending-flows-min-set F G nP }

Some of the set-offending-flows definition

**lemma** offending-not-empty: [ F ∈ set-offending-flows G nP ] ⇒ F ≠ {}
(proof)

**lemma** empty-offending-contra:
[ F ∈ set-offending-flows G nP; F = {} ] ⇒ False
(proof)

**lemma** offending-not-evalD: F ∈ set-offending-flows G nP ⇒ ¬ sinvar G nP
(proof)

**lemma** sinvar-no-offending: sinvar G nP ⇒ set-offending-flows G nP = {}
(proof)

**theorem** removing-offending-flows-makes-invariant-hold:
∀ F ∈ set-offending-flows G nP. sinvar (delete-edges G F) nP
(proof)

**corollary** valid-without-offending-flows:
[ F ∈ set-offending-flows G nP ] ⇒ sinvar (delete-edges G F) nP
(proof)

**lemma** set-offending-flows-simp:
[ wf-graph G ] ⇒
set-offending-flows G nP = { F. F ⊆ edges G ∧
¬ sinvar G nP ∧ sinvar (nodes = nodes G, edges = edges G − F) nP) ∧
(∀ (e1, e2) ∈ F. ¬ sinvar (nodes = nodes G, edges = {(e1, e2)} ∪ (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)
\[\{ \text{nodes} = \{''v1''\}, \text{edges} = \{(''v1''\', ''v2''\')} \} \uplus \{\text{id} = \{}\]
In general, there exists a sinvar such that the invariant does not hold and no offending flows exits.

\[ \exists \text{sinvar}. \neg \text{sinvar } G \ nP \land \text{SecurityInvariant-withOffendingFlows.set-offending-flows } \text{sinvar } G \ nP = \{\} \]

(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:
\[ [\text{wf-graph } G; \neg \text{sinvar } G \ nP ] \implies \text{set-offending-flows } G \ nP \neq \{\} \]

and

mono-sinvar:
\[ [\text{wf-graph } (\{ \text{nodes} = N, \text{edges} = E \}); \ E' \subseteq \ E; \text{sinvar } (\{ \text{nodes} = N, \text{edges} = E \}) \ nP ] \implies \text{sinvar } (\{ \text{nodes} = N, \text{edges} = E' \}) \ nP \]

and mono-offending:
\[ [\text{wf-graph } G; \text{is-offending-flows } \text{ff } G \ nP ] \implies \text{is-offending-flows } (\text{ff} \cup f') \ G \ nP \]

begin


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].
Some notes about the notation: \( \text{fst} \circ F \) means to apply the function \( \text{fst} \) to the set \( F \) element-wise.

Example: If \( F \) is a set of directed edges, \( F \subseteq \text{edges} G \), then \( \text{fst} \circ F \) is the set of senders and \( \text{snd} \circ f \) the set of receivers.

---

**locale** SecurityInvariant = SecurityInvariant-preliminaries sinvar

**for** sinvar::\((v::\text{vertex}) \text{ graph} \Rightarrow (v::\text{vertex} \Rightarrow \text{a}) \Rightarrow \text{bool} \)

**fixes** default-node-properties :: \( \text{a} (\bot) \)

**and** receiver-violation :: \( \text{bool} \)

**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 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**:

\[
\begin{align*}
\text{wf-graph } G; \neg \text{sinvar } G nP; F \in \text{set-offending-flows } G nP \implies \\
(\neg \text{receiver-violation} \implies i \in \text{fst} \circ F \implies \neg \text{sinvar } G (nP(i := \bot))) \land \\
(\text{receiver-violation} \implies i \in \text{snd} \circ F \implies \neg \text{sinvar } G (nP(i := \bot)))
\end{align*}
\]

and

**default-unique**:

\[
\begin{align*}
\text{otherbot} \neq \bot \implies \\
\exists (G::(v::\text{vertex}) \text{ graph}) nP i F. \text{wf-graph } G \land \neg \text{sinvar } G nP \land F \in \text{set-offending-flows } G nP \land \neg \text{sinvar } (\text{delete-edges } G F) nP \land \\
(\neg \text{receiver-violation} \implies i \in \text{fst} \circ F \land \text{sinvar } G (nP(i := \text{otherbot}))) \land \\
(\text{receiver-violation} \implies i \in \text{snd} \circ F \land \text{sinvar } G (nP(i := \text{otherbot})))
\end{align*}
\]

---

**fun** node-props :: \((v, \text{a}) \) TopoS-Params \Rightarrow (v \Rightarrow \text{a}) where

node-props \( P = (\lambda i. \text{case } (\text{node-properties } P) i \text{ of Some property } \Rightarrow \text{property } | \text{None } \Rightarrow \bot)\)

**definition** node-props-formaldef :: \((v, \text{a}) \) TopoS-Params \Rightarrow (v \Rightarrow \text{a}) where

node-props-formaldef \( P \equiv \\
(\lambda i. (\text{if } i \in \text{dom } (\text{node-properties } P) \text{ then the } (\text{node-properties } P i) \text{ else } \bot))\)

**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 \( \text{sinvar} \)

**definition** eval::v graph \Rightarrow (v, \text{a}) TopoS-Params \Rightarrow bool where

\( \text{eval } G P \equiv \text{wf-graph } G \land \text{sinvar } G (\text{node-props } P) \)
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))) ∧
  (receiver-violation → i ∈ snd ' F → ¬ sinvar G (nP(i := otherbot)))
shows otherbot = ⊥
⟨proof⟩
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 ⇒ (v::vertex ⇒ 'a) ⇒ bool
+ fixes default-node-properties :: 'a (⊥)
assumes default-secure-IFS:
  [ wf-graph G; f ∈ set-offending-flows G nP ] ⇒
  ∀ i ∈ snd ' f. ¬ sinvar G (nP(i := ⊥))
and
— If some otherbot fulfills default-secure, it must be ⊥ Hence, ⊥ is uniquely defined
default-unique-IFS:
(∀ G f nP i. wf-graph G ∧ f ∈ set-offending-flows G nP ∧ i ∈ snd ' f
  → ¬ sinvar G (nP(i := otherbot))) ⇒ otherbot = ⊥
begin
lemma default-unique-EX-notation: otherbot ≠ ⊥ ⇒
  ∃ G nP i f. wf-graph G ∧ ¬ sinvar G nP ∧ f ∈ set-offending-flows G nP ∧
  sinvar (delete-edges G f) nP ∧
  (i ∈ snd ' f ∧ sinvar G (nP(i := otherbot)))
 ⟨proof⟩
end

sublocale SecurityInvariant-IFS ⊆ SecurityInvariant where receiver-violation=True
⟨proof⟩

locale SecurityInvariant-IFS-otherDirection = SecurityInvariant where receiver-violation=True
sublocale SecurityInvariant-IFS-otherDirection ⊆ SecurityInvariant-IFS
⟨proof⟩

lemma default-uniqueness-by-counterexample-IFS:
  assumes (∀ G F nP i. wf-graph G ∧ F ∈ SecurityInvariant-withOffendingFlows.set-offending-flows
  sinvar G nP ∧ i ∈ snd' F
  →¬ sinvar G (nP(i := otherbot)))
  and otherbot ≠ default-value
  sinvar G nP) ∧
  sinvar (delete-edges G F) nP ∧
  i ∈ snd ' F ∧ sinvar G (nP(i := otherbot))
shows otherbot = default-value
⟨proof⟩

2.4 Access Control Strategy (ACS)

locale SecurityInvariant-ACS = SecurityInvariant-preliminaries sinvar
  for sinvar::"(v::vertex) graph ⇒ (v::vertex ⇒ 'a) ⇒ bool"
+
  fixes default-node-properties :: 'a (⊥)

assumes default-secure-ACS:
  [ wf-graph G; f ∈ set-offending-flows G nP ]
  ⇒ ∀ i ∈ fst' f. ¬ sinvar G (nP(i := ⊥))

and

default-unique-ACS:
  (∀ G f nP i. wf-graph G ∧ f ∈ set-offending-flows G nP ∧ i ∈ fst' f
  →¬ sinvar G (nP(i := otherbot)))
  ⇒ otherbot = ⊥

begin
  lemma default-unique-EX-notation: otherbot ≠ ⊥
  ⇒ ∃ G nP i f. wf-graph G ∧ ¬ sinvar G nP ∧ f ∈ set-offending-flows G nP ∧
  sinvar (delete-edges G f) nP ∧
  (i ∈ fst' f ∧ sinvar G (nP(i := otherbot)))
  ⟨proof⟩
end

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

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

lemma default-uniqueness-by-counterexample-ACS:
  assumes (∀ G F nP i. wf-graph G ∧ F ∈ SecurityInvariant-withOffendingFlows.set-offending-flows
  sinvar G nP ∧ i ∈ fst ' F
  →¬ sinvar G (nP(i := otherbot)))
  and otherbot ≠ default-value
  sinvar G nP) ∧
The sublocale relationships tell that the simplified SecurityInvariant-ACS and SecurityInvariant-IFS assumptions suffice to do the 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 monoticity,

context SecurityInvariant-withOffendingFlows
begin

We define the monotonicity of sinvar:

\[ \forall nP \, N \, E' \, E. \left[ \text{wf-graph}\left(\{\text{nodes} = N, \text{edges} = E\}\right); \, E' \subseteq E; \, \text{sinvar}\left(\{\text{nodes} = N, \text{edges} = E\}\right) \right] \rightarrow \text{sinvar}\left(\{\text{nodes} = N, \text{edges} = E'\}\right) \]

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

definition sinvar-mono :: bool where

\[ \text{sinvar-mono} \longleftrightarrow \left(\forall nP \, N \, E' \, E. \left[ \text{wf-graph}\left(\{\text{nodes} = N, \text{edges} = E\}\right) \land E' \subseteq E \land \text{sinvar}\left(\{\text{nodes} = N, \text{edges} = E\}\right) \rightarrow \text{sinvar}\left(\{\text{nodes} = N, \text{edges} = E'\}\right) \right] \right) \]

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

lemma sinvar-mono-I-proofrule-simple:

\[ \left[ \left(\forall G \, nP. \, \text{sinvar}\left(\{G\} \, nP = (\forall (e1, e2) \in \text{edges} G. \, P \, e1 \, e2 \, nP)\right) \right) \right] \rightarrow \text{sinvar-mono} \]

(proof)

lemma sinvar-mono-I-proofrule:

\[ \left[ \left(\forall nP \, (G::'e\, \text{graph}). \, \text{sinvar}\left(\{G\} \, nP = (\forall (e1, e2) \in \text{edges} G. \, P \, e1 \, e2 \, nP \, G)\right) \right); \right. \]

\[ \left. \left(\forall nP \, e1 \, e2 \, N \, E' \, E. \left[ \text{wf-graph}\left(\{\text{nodes} = N, \text{edges} = E\}\right) \land (e1,e2) \in E \land E' \subseteq E \land P \, e1 \, e2 \, nP \left(\{\text{nodes} = N, \text{edges} = E\}\right) \rightarrow P \, e1 \, e2 \, nP \left(\{\text{nodes} = N, \text{edges} = E'\}\right) \right] \right] \rightarrow \text{sinvar-mono} \]

(proof)

Invariant violations do not disappear if we add more flows.

lemma sinvar-mono-imp-negative-mono:

\[ \text{sinvar-mono} \rightarrow \text{wf-graph}\left(\{\text{nodes} = N, \text{edges} = E\}\right) \rightarrow E' \subseteq E \rightarrow \neg \text{sinvar}\left(\{\text{nodes} = N, \text{edges} = E'\}\right) \rightarrow \neg \text{sinvar}\left(\{\text{nodes} = N, \text{edges} = E\}\right) \rightarrow \neg \text{sinvar}\left(\{\text{nodes} = N, \text{edges} = E\}\right) \rightarrow \neg \text{sinvar}\left(\{\text{nodes} = N, \text{edges} = E'\}\right) \]

(proof)
(proof)

**corollary** sinvar-mono-imp-negative-delete-edge-mono:

\[ \text{sinvar-mono} \implies \text{wf-graph } G \implies X \subseteq Y \implies \neg \text{sinvar } (\text{delete-edges } G Y) \text{ nP} \implies \neg \text{sinvar } (\text{delete-edges } G X) \text{ nP} \]

(proof)

**lemma** sinvar-mono-imp-is-offending-flows-mono:

**assumes** mono: sinvar-mono

and \( \text{wfG}: \text{wf-graph } G \)

**shows** is-offending-flows \( FF G \text{ nP} \implies \text{is-offending-flows } (FF \cup F) G \text{ nP} \)

(proof)

**lemma** sinvar-mono-imp-sinvar-mono:

\[ \text{sinvar-mono} \implies \text{wf-graph } (\{ \text{nodes} = N, \text{edges} = E \}) \implies E' \subseteq E \implies \text{sinvar } (\{ \text{nodes} = N, \text{edges} = E' \}) \text{ nP} \]

(proof)

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** minimize-offending-overapprox :: ('v × 'v) list ⇒ ('v × 'v) list ⇒ 'v graph ⇒ ('v ⇒ 'a) ⇒ ('v × 'v) list where

minimize-offending-overapprox [] keep - = keep |

minimize-offending-overapprox (f#fs) keep G nP = (if sinvar (delete-edges-list G (fs@keep)) nP then

minimize-offending-overapprox fs keep G nP

else

minimize-offending-overapprox fs (f#keep) G nP

)

The graph we check in minimize-offending-overapprox, \( G (-) (fs \cup \text{keep}) \) is the graph from the offending-flows-min-set condition. We add \( f \) and remove it.

**lemma** minimize-offending-overapprox-subset:

\[ \text{set } (\text{minimize-offending-overapprox } ff \text{ keeps } G \text{ nP}) \subseteq \text{set } ff \cup \text{set } \text{keeps} \]

(proof)

**lemma** not-model-mono-imp-addedge-mono:

**assumes** mono: sinvar-mono
\textbf{and} \( vG \): \textit{wf-graph} \( G \) and \( \textit{ain}: (a1, a2) \in \text{edges} \ G \) and \( \textit{xy}: X \subseteq Y \) and \( \textit{ns}: \neg \sinvar\ (\text{add-edge} \ a1 \ a2 \ (\text{delete-edges} \ G \ (Y))) \) \( nP \)
\( \textit{shows} \neg \sinvar\ (\text{add-edge} \ a1 \ a2 \ (\text{delete-edges} \ G \ X)) \) \( nP \)
\( \langle \textit{proof} \rangle \)

\textbf{theorem} \textit{is-offending-flows-min-set-minimalize-offending-overapprox}:
\textbf{assumes} \textit{mono}: \sinvar-\textit{mono}
\textbf{and} \( vG \): \textit{wf-graph} \( G \) and \( \textit{iO}: \textit{is-offending-flows} \ (\text{set} \ \textit{ff})\ G \) \( nP \) and \( \textit{sF}: \text{set} \ \textit{ff} \subseteq \text{edges} \ G \) and \( \textit{dF}: \text{distinct} \ \textit{ff} \)
\textbf{shows} \textit{is-offending-flows-min-set} \ (\textit{minimalize-offending-overapprox} \ \textit{ff} \ [\ G \ nP]) \ G \) \( nP \)
\( \langle \textit{proof} \rangle \)

\textbf{corollary} \textit{mono-imp-set-offending-flows-not-empty}:
\textbf{assumes} \textit{mono-sinvar}: \sinvar-\textit{mono}
\textbf{and} \( vG \): \textit{wf-graph} \( G \) and \( \textit{iO}: \textit{is-offending-flows} \ (\text{set} \ \textit{ff})\ G \) \( nP \) and \( \textit{sS}: \text{set} \ \textit{ff} \subseteq \text{edges} \ G \) and \( \textit{dF}: \text{distinct} \ \textit{ff} \)
\textbf{shows} \textit{set-offending-flows} \( G \) \( nP \neq \{\} \)
\( \langle \textit{proof} \rangle \)

To show that \textit{set-offending-flows} is not empty, the previous corollary \([\sinvar-\textit{mono}; \ \textit{wf-graph} \ ?G; \ \textit{is-offending-flows} \ (\text{set} \ ?\textit{ff}) \ ?G \ ?nP; \ \textit{set} \ ?\textit{ff} \subseteq \text{edges} \ ?G; \ \textit{distinct} \ ?\textit{ff}] \implies \textit{set-offending-flows} \ ?G \ ?nP \neq \{\}\) is very useful. Just select \textit{set} \textit{ff} = \textit{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

\textbf{theorem} \textit{valid-empty-edges-iff-exists-offending-flows}:
\textbf{assumes} \textit{mono}: \sinvar-\textit{mono and} \textit{wfG}: \textit{wf-graph} \( G \) and \( \textit{noteval}: \neg \sinvar \ G \) \( nP \)
\textbf{shows} \( \sinvar\ (\{\} \ | \ \textit{nodes} = \text{nodes} \ G, \ \textit{edges} = \{\}) \) \( nP \) \iff \textit{set-offending-flows} \( G \) \( nP \neq \{\} \)
\( \langle \textit{proof} \rangle \)

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

\textbf{lemma} \textit{minimalize-offending-overapprox-keeps-keeps}: \( \textit{set} \ \textit{keeps} \subseteq \textit{set} \ (\textit{minimalize-offending-overapprox} \ \textit{ff} \ \textit{keeps} \ G \ nP) \)
\( \langle \textit{proof} \rangle \)

\textbf{lemma} \textit{minimalize-offending-overapprox-subseteq-input}: \( \textit{set} \ (\textit{minimalize-offending-overapprox} \ \textit{ff} \ \textit{keeps} \ G \ nP) \subseteq (\textit{set} \ \textit{ff}) \cup (\textit{set} \ \textit{keeps}) \)
\( \langle \textit{proof} \rangle \)

\textbf{end}

\textbf{context} \textit{SecurityInvariant-preliminaries}
\textbf{begin}
\textit{sinvar-\textit{mono}} naturally holds in \textit{SecurityInvariant-preliminaries}
\textbf{lemma} \textit{sinvar-\textit{monI}}: \textit{sinvar-\textit{mono}}
Note: due to monotonicity, the minimality also holds for arbitrary subsets

\textbf{Lemma assumes} \( \text{wf-graph } G \) \text{ and } \( \text{is-offending-flows-min-set } F G nP \) \text{ and } \( F \subseteq \text{edges } G \) \text{ and } \( E \subseteq F \) \text{ and } \( E \neq \{\} \)

\textbf{Shows} \( \neg \sinvar (| \text{nodes } = \text{nodes } G, \text{edges } = ((\text{edges } G) - F) \cup E |) nP \)

\textbf{Proof}

The algorithm \texttt{minimalize-offending-overapprox} is correct

\textbf{Lemma} \texttt{minimalize-offending-overapprox-sound}:
\[ \text{[ wf-graph } G; \text{is-offending-flows (set } ff \text{) } G nP; \text{set } ff \subseteq \text{edges } G; \text{distinct } ff \] \implies \text{is-offending-flows-min-set (set (minimalize-offending-overapprox } ff\| G nP)) G nP \]

\textbf{Proof}

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

\textbf{Theorem} \texttt{minimalize-offending-overapprox-gives-back-an-offending-flow}:
\[ \text{[ wf-graph } G; \text{is-offending-flows (set } ff \text{) } G nP; \text{set } ff \subseteq \text{edges } G; \text{distinct } ff \] \implies \text{ (set (minimalize-offending-overapprox } ff\| G nP)) \in \text{set-offending-flows } G nP \]

\textbf{Proof}

end

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

\textbf{Fun} \texttt{minimalize-offending-overapprox} :: \( ('v \text{ graph } \Rightarrow \text{bool}) \Rightarrow ('v \times 'v) \text{ list } \Rightarrow ('v \times 'v) \text{ list } \Rightarrow 'v \text{ graph } \Rightarrow ('v \times 'v) \text{ list } \) where
\begin{verbatim}
mminimalize-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)
\end{verbatim}

\textbf{Lemma} \texttt{minimalize-offending-overapprox-boundnP}:
\textbf{Shows} \texttt{minimalize-offending-overapprox (λG. m G nP) fs keeps G =}
\texttt{SecurityInvariant-withOffendingFlows.minimalize-offending-overapprox m fs keeps G nP}

\textbf{Proof}

\textbf{Context} \texttt{SecurityInvariant-withOffendingFlows}

\textbf{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. \( \sinvar-mono; \text{wf-graph } ?G; \neg \sinvar ?G ?nP \implies \sinvar (| \text{nodes } = \text{nodes } ?G, \text{edges } = \{\} |) ?nP = (\text{set-offending-flows } ?G ?nP \neq \{\}) \) already hints this.
lemma mono-imp-emptyoffending-eq-nevervalid:
\[ \begin{align*}
\neg (\exists F \subseteq \text{edges } G. \ \text{sinvar} (\text{delete-edges } G F) \ nP) \\
\Rightarrow \\
\{ e \in \text{set-offending-flows} \mid \text{nodes } = V, \ \text{edges } = \{ e \} \} \ nP
\end{align*} \]
\end{proof}

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 \( F_{\text{add}} \) and minimality in \( F \) is preserved

\[ \text{lemma minimality-offending-flows-mono-edges-graph-extend:} \]
\[ \begin{align*}
\neg (\exists F \subseteq \text{edges } G. \ \text{sinvar} (\text{delete-edges } G F) \ nP) \\
\Rightarrow \\
\forall (e1, e2) \in F. \ \neg \text{sinvar} (\text{add-edge } e1 e2 (\text{delete-edges } (\text{nodes } = V, \ \text{edges } = E \cup F)) nP)
\end{align*} \]
\end{proof}

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

\[ \text{corollary minimality-offending-flows-mono-edges-graph:} \]
\[ \begin{align*}
\forall (e1, e2) \in F. \ \neg \text{sinvar} (\text{add-edge } e1 e2 (\text{delete-edges } (\text{nodes } = V, \ \text{edges } = E \cup F)) nP)
\end{align*} \]
\end{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.

\[ \text{theorem mono-extend-set-offending-flows:} \]
\[ \begin{align*}
\exists F \in \text{set-offending-flows} \mid \text{nodes } = V, \ \text{edges } = E \ nP \quad \Rightarrow \\
\exists F' \subseteq F. \ \forall (e1, e2) \in F. \ \neg \text{sinvar} (\text{add-edge } e1 e2 (\text{delete-edges } (\text{nodes } = V, \ \text{edges } = E \cup F)) nP)
\end{align*} \]
\end{proof}

The offending flows are monotonic.

\[ \text{corollary offending-flows-union-mono:} \]
\[ \begin{align*}
\forall (e1, e2) \in F. \ \neg \text{sinvar} (\text{add-edge } e1 e2 (\text{delete-edges } (\text{nodes } = V, \ \text{edges } = E \cup F)) nP)
\end{align*} \]
\end{proof}

\[ \text{lemma set-offending-flows-insert-contains-new:} \]
\[ \begin{align*}
\forall (e1, e2) \in F. \ \neg \text{sinvar} (\text{add-edge } e1 e2 (\text{delete-edges } (\text{nodes } = V, \ \text{edges } = E \cup F)) nP)
\end{align*} \]
\end{proof}

value \( \text{Pow} \{1::\text{int}, 2, 3\} \cup \{\{8\}, \{9\}\} \)
\textbf{value} \bigcup x \in \text{Pow} \{1::\text{int}, 2, 3\}, \bigcup y \in \{\{8::\text{int}\}, \{9\}\}. \{x \cup y\}

— combines powerset of A with B

\textbf{definition} \textit{pow-combine} :: \textit{x set} \Rightarrow \textit{\textit{x set} set} \Rightarrow \textit{\textit{x set set} where}

\[ \text{pow-combine} \ A \ B \equiv (\bigcup \ X \in \text{Pow} \ A. \bigcup \ Y \in \ B. \{X \cup Y\}) \cup \text{Pow} \ A \]

\textbf{value} \ \text{pow-combine} \ \{1::\text{int}, 2\} \ \{\{5::\text{int}, 6\}, \{8\}\}

\textbf{value} \ \text{pow-combine} \ \{1::\text{int}, 2\} \ \{\}\n
\textbf{lemma} \ \textit{pow-combine-mono}:

\textit{fixes} S :: 'a set set

\textit{and} \ X :: 'a set

\textit{and} \ Y :: 'a set

\textit{assumes} \ a1: \forall \ F \in S. F \subseteq X

\textit{shows} \ \forall \ F \in \text{pow-combine} \ Y \ S. F \subseteq Y \cup X

\langle \textit{proof} \rangle

\textbf{lemma} \ S \subseteq \text{pow-combine} \ X \ S \langle \textit{proof} \rangle

\textbf{lemma} \ Pow \ X \subseteq \text{pow-combine} \ X \ S \langle \textit{proof} \rangle

\textbf{lemma} \ \textit{rule-pow-combine-fixfst}: \ B \subseteq C \Rightarrow \text{pow-combine} \ A \ B \subseteq \text{pow-combine} \ A \ C

\langle \textit{proof} \rangle

\textbf{value} \ \text{pow-combine} \ \{1::\text{int}, 2\} \ \{\{5::\text{int}, 6\}, \{1\}\} \subseteq \text{pow-combine} \ \{1::\text{int}, 2\} \ \{\{5::\text{int}, 6\}, \{8\}\}

\textbf{lemma} \ \textit{rule-pow-combine-fixfst-Union}: \bigcup \ B \subseteq \bigcup \ C \Rightarrow \bigcup \ \text{pow-combine} \ A \ B \subseteq \bigcup \ \text{pow-combine} \ A \ C

\langle \textit{proof} \rangle

\textbf{context} \ \textit{SecurityInvariant-preliminaries}

\textbf{begin}

\textbf{lemma} \ \textit{offending-partition-subset-empty}:

\textit{assumes} \ a1: \forall \ F \in \text{(set-offending-flows} (\text{nodes} = V, \text{edges} = E \cup X) \ |NP). F \subseteq X

\textit{and} \ \textit{wfGEX}: \text{wf-graph} (\text{nodes} = V, \text{edges} = E \cup X)

\textit{and} disj: \ E \cap X = \{\}

\textit{shows} \ \text{(set-offending-flows} (\text{nodes} = V, \text{edges} = E \ |NP) = \{\}

\langle \textit{proof} \rangle

\textbf{corollary} \ \textit{partitioned-offending-subseteq-pow-combine}:

\textit{assumes} \ \textit{wfGEX}: \text{wf-graph} (\text{nodes} = V, \text{edges} = E \cup X)

\textit{and} disj: \ E \cap X = \{\}

\textit{and} \ \textit{partitioned-offending}: \forall \ F \in \text{(set-offending-flows} (\text{nodes} = V, \text{edges} = E \cup X) \ |NP). F \subseteq X

\textit{shows} \ \text{(set-offending-flows} (\text{nodes} = V, \text{edges} = E \cup X) \ |NP) \subseteq \text{pow-combine} \ X \ \text{(set-offending-flows} (\text{nodes} = V, \text{edges} = E) \ |NP)

\langle \textit{proof} \rangle

\textbf{end}
context SecurityInvariant-preliminaries
begin

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

lemma Un-set-offending-flows-bound-minus:
assumes $\text{wfG}$: $\text{wf-graph}$ ($\{\text{nodes} = V, \text{edges} = E\}$
and $\text{offending}$: $\bigcup$ set-offending-flows ($\{\text{nodes} = V, \text{edges} = E\}$ $nP \subseteq X$
shows $\bigcup$ set-offending-flows ($\{\text{nodes} = V, \text{edges} = E - \{f\}\}$ $nP \subseteq X - \{f\}$

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 $\text{wfG}$: $\text{wf-graph}$ ($\{\text{nodes} = V, \text{edges} = E\}$
and $\text{offending}$: $\bigcup$ set-offending-flows ($\{\text{nodes} = V, \text{edges} = E\}$ $nP \subseteq X$
shows $\bigcup$ set-offending-flows ($\{\text{nodes} = V, \text{edges} = E - E'\}$ $nP \subseteq X - E'$

⟨proof⟩

corollary Un-set-offending-flows-bound-minus-subseteq':
\[
\begin{align*}
[\text{wf-graph} (\{\text{nodes} = V, \text{edges} = E\}); & \\
\bigcup \text{ set-offending-flows} (\{\text{nodes} = V, \text{edges} = E \} nP \subseteq X) & \implies \\
\bigcup \text{ set-offending-flows} (\{\text{nodes} = V, \text{edges} = E - E'\} nP \subseteq X - E') 
\end{align*}
\]
⟨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 $\text{sinvar}$ is predicate which starts with $\forall (v_1, v_2) \in \text{edges} G \ldots$, 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 $\text{sinvar-all-edges-normal-form}$ :: ($'a \Rightarrow 'a \Rightarrow \text{bool}$) $\Rightarrow \text{bool}$ where $\text{sinvar-all-edges-normal-form}$ $P \equiv \forall G nP. \text{sinvar} G nP = (\forall (e1, e2) \in \text{edges} G. P (nP e1) (nP e2))$

reflexivity is needed for convenience. If a security invariant is not reflexive, that means that all nodes with the default parameter $\bot$ are not allowed to communicate with each other. Non-reflexivity is possible, but requires more work.
\textbf{definition} ENF-refl :: ('a ⇒ 'a ⇒ bool) ⇒ bool \textbf{where}
\begin{align*}
\text{ENF-refl } P &\equiv \text{sinvar-all-edges-normal-form } P \land (\forall p1. P p1 p1)
\end{align*}

\textbf{lemma} monotonicity-sinvar-mono: sinvar-all-edges-normal-form \( P \implies \) sinvar-mono
\begin{proof}
\end{proof}

\textbf{end}

\subsection{Offending Flows}

\textbf{context} SecurityInvariant-withOffendingFlows
\textbf{begin}

The insight: for all edges in the members of the offending flows, \( \lnot P \) holds.

\textbf{lemma} ENF-offending-imp-not-P:
\begin{align*}
\text{assumes} \quad &\text{sinvar-all-edges-normal-form } P F \in \text{set-offending-flows } G nP (e1, e2) \in F \\
\text{shows} \quad &\lnot P (nP e1) (nP e2)
\end{align*}
\begin{proof}
\end{proof}

Hence, the members of \text{set-offending-flows} must look as follows.

\textbf{lemma} ENF-offending-set-P-representation:
\begin{align*}
\text{assumes} \quad &\text{sinvar-all-edges-normal-form } P F \in \text{set-offending-flows } G nP \\
\text{shows} \quad &F = \{ (e1,e2), (e1, e2) \in \text{edges } G \land \lnot P (nP e1) (nP e2) \} \begin{cases} \text{is } F = ?E \end{cases}
\end{align*}
\begin{proof}
\end{proof}

We can show left to right of the desired representation of \text{set-offending-flows}

\textbf{lemma} ENF-offending-subseteq-lhs:
\begin{align*}
\text{assumes} \quad &\text{sinvar-all-edges-normal-form } P \\
\text{shows} \quad &\text{set-offending-flows } G nP \subseteq \{ (e1,e2), (e1, e2) \in \text{edges } G \land \lnot P (nP e1) (nP e2) \} \\
\end{align*}
\begin{proof}
\end{proof}

if \text{set-offending-flows} is not empty, we have the other direction.

\textbf{lemma} ENF-offending-not-empty-imp-ENF-offending-subseteq-rhs:
\begin{align*}
\text{assumes} \quad &\text{sinvar-all-edges-normal-form } P \text{ set-offending-flows } G nP \neq \{ \}
\text{shows} \quad &\{ (e1,e2) \in \text{edges } G, \lnot P (nP e1) (nP e2) \} \subseteq \text{set-offending-flows } G nP
\end{align*}
\begin{proof}
\end{proof}

\textbf{lemma} ENF-notevalmodel-imp-offending-not-empty:
\begin{align*}
\text{sinvar-all-edges-normal-form } P \implies \lnot \text{sinvar } G nP \implies \text{set-offending-flows } G nP \neq \{ \}
\end{align*}
\begin{proof}
\end{proof}

\textbf{lemma} ENF-offending-case1:
\begin{align*}
[ \text{sinvar-all-edges-normal-form } P; \lnot \text{sinvar } G nP ] &\implies \\
\{ (e1,e2), (e1, e2) \in (\text{edges } G) \land \lnot P (nP e1) (nP e2) \} &\text{ = set-offending-flows } G nP
\end{align*}
\begin{proof}
\end{proof}

\textbf{lemma} ENF-offending-case2:
\begin{align*}
[ \text{sinvar-all-edges-normal-form } P; \text{sinvar } G nP ] &\implies \\
\{ \} &\text{ = set-offending-flows } G nP
\end{align*}
\begin{proof}
\end{proof}
\textbf{theorem} ENF-offending-set:
\[\sinvar\text{-all-edges-normal-form } P \implies\]
\[\text{set-offending-flows } G \ nP = (\text{if } \sinvar \ G \ nP \text{ then } \{\} \text{ else } \{((e_1,e_2), (e_1, e_2) \in \text{edges } G \land \neg P (nP e_1) (nP e_2))\})\]
\text{(proof)}

4.1.2 Lemmata

\textbf{lemma} (in SecurityInvariant-withOffendingFlows) ENF-offending-members:
\[\neg \sinvar \ G \ nP; \sinvar\text{-all-edges-normal-form } P; f \in \text{set-offending-flows } G \ nP \implies f \subseteq \text{edges } G \land (\forall (e_1, e_2) \in f. \neg P (nP e_1) (nP e_2))\]
\text{(proof)}

4.1.3 Instance Helper

\textbf{lemma} (in SecurityInvariant-withOffendingFlows) ENF-refl-not-offedning:
\[\neg \sinvar \ G \ nP; f \in \text{set-offending-flows } G \ nP; \text{ENF-refl } P \implies (\forall (e_1, e_2) \in f. e_1 \neq e_2)\]
\text{(proof)}

\textbf{lemma} (in SecurityInvariant-withOffendingFlows) ENF-default-update-fst:
\textbf{fixes} default-node-properties :: 'a ('\perp')
\textbf{assumes} modelInv: \neg \sinvar \ G \ nP
\textbf{and} ENFdef: sinvar\text{-all-edges-normal-form } P
\textbf{and} secdef: \forall (nP::'v \Rightarrow 'a) e1 e2. \neg (P (nP e_1) (nP e_2)) \implies \neg (P \ (nP e_2))
\textbf{shows} \neg (\forall (e_1, e_2) \in \text{edges } G. P ((nP(i := \perp)) e_1) (nP e_2))
\text{(proof)}

\textbf{lemma} (in SecurityInvariant-withOffendingFlows) ENF-fsts-refl-instance:
\textbf{fixes} default-node-properties :: 'a ('\perp')
\textbf{assumes} a-enf-refl: ENF-refl P
\textbf{and} a3: \forall (nP::'v \Rightarrow 'a) e1 e2. \neg (P (nP e_1) (nP e_2)) \implies \neg (P \ (nP e_2))
\textbf{and} a-offending: f \in \text{set-offending-flows } G \ nP
\textbf{and} a-i-fsts: i \in \text{fst ' } f
\textbf{shows} \neg \sinvar \ G \ (nP(i := \perp))
\text{(proof)}

20
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 ∈ 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)

⟨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

21
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: ∀ s r. P ⊥ s ⊥ r
- and a-botdefault: ∀ s r. (nP s) ⊥ r → ¬ P (nP s) r → ¬ P ⊥ (nP s) r
- 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) ENFnrSR-snds-weakrefl-instance:**

- **Fixes** default-node-properties :: 'a (⊥)
- **Assumes** a-enf: sinvar-all-edges-normal-form-not-refl-SR P
- and a-weakrefl: ∀ s r. P ⊥ s ⊥ r
- and a-botdefault: ∀ s r. (nP s) ⊥ r → ¬ P (nP s) r → ¬ P ⊥ (nP s) r
- and a-offending: f ∈ set-offending-flows G nP
- and a-i-snds: i ∈ snd' f
  - shows ¬ sinvar G (nP(i := ⊥))

**Proof**

4.4 edges normal form not refl ENFnR

**Definition (in SecurityInvariant-withOffendingFlows) sinvar-all-edges-normal-form-not-refl :: ('a ⇒ 'a ⇒ bool) ⇒ bool where**

sinvar-all-edges-normal-form-not-refl P ≡ ∀ G nP. sinvar G nP = (∀ (e1, e2) ∈ edges G. e1 ≠ e2 → P (nP e1) (nP e2))

we derive everything from the ENFnrSR form

**Lemma (in SecurityInvariant-withOffendingFlows) ENFnr-to-ENFnrSR:**


**Proof**

4.4.1 Offending Flows

**Theorem (in SecurityInvariant-withOffendingFlows) ENFnr-offending-set:**

\[
\begin{align*}
&\{ (e1,e2). (e1, e2) ∈ edges G \land e1 ≠ e2 \land \neg P (nP e1) e1 (nP e2) e2 \} \\
\end{align*}
\]

**Proof**
4.4.2 Instance helper

**lemma** *(in SecurityInvariant-withOffendingFlows) ENFnr-fsts-weakrefl-instance:*
  
  *fixes* default-node-properties :: ‘a (⊥)
  
  *assumes* a-enf: sinvar-all-edges-normal-form-not-refl P
  
  and a-botdefault: ∀ e1 e2. e2 ≠ ⊥ → ¬ P e1 e2 → ¬ P ⊥ e2
  
  and a-alltobot: ∀ e2. P ⊥ 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) 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 ⊥

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-bottosorted: ∀ e2. e1 ≠ ⊥ → ¬ P e1 e2 → ¬ P ⊥

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⟩**
begin

lemma (in SecurityInvariant-preliminaries) sinvar-valid-remove-flattened-offending-flows:
  assumes wf-graph \[(\text{nodes} = \text{nodes}_G, \text{edges} = \text{edges}_G)\]
  shows sinvar \[(\text{nodes} = \text{nodes}_G, \text{edges} = \text{edges}_G - \bigcup \text{set-offending-flows} \{\text{nodes} = \text{nodes}_G, \text{edges} = \text{edges}_G\} \text{nP} \} \} \text{nP}\]
⟨proof ⟩

lemma (in SecurityInvariant-preliminaries) sinvar-valid-remove-SOME-offending-flows:
  assumes set-offending-flows \{\text{nodes} = \text{nodes}_G, \text{edges} = \text{edges}_G \} \text{nP} \neq \{\}
  shows sinvar \{\text{nodes} = \text{nodes}_G, \text{edges} = \text{edges}_G - \{\text{SOME} F. F \in \text{set-offending-flows} \{\text{nodes} = \text{nodes}_G, \text{edges} = \text{edges}_G\} \text{nP} \} \} \text{nP}\]
⟨proof ⟩

lemma (in SecurityInvariant-preliminaries) sinvar-valid-remove-minimalize-offending-overapprox:
  assumes wf-graph \[(\text{nodes} = \text{nodes}_G, \text{edges} = \text{edges}_G)\]
  and \(\neg \) sinvar \{\text{nodes} = \text{nodes}_G, \text{edges} = \text{edges}_G\} \text{nP} 
  and set Es = \text{edges}_G and distinct Es
  shows sinvar \{(\text{nodes} = \text{nodes}_G, \text{edges} = \text{edges}_G - \text{set (minimalize-offending-overapprox Es} [] \{\text{nodes} = \text{nodes}_G, \text{edges} = \text{edges}_G\} \text{nP} \} \} \text{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 \equiv Unassigned

fun allowed-subnet-flow :: subnets \Rightarrow subnets \Rightarrow 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 s) \cdot = True |
  allowed-subnet-flow (BorderRouter s) (Subnet s) = False |
  allowed-subnet-flow (BorderRouter s) \cdot = True |
  allowed-subnet-flow (BorderRouter' s1) (Subnet s2) = (s1 = s2) |
  allowed-subnet-flow (BorderRouter' \cdot) = True |
  allowed-subnet-flow InboundRouter (Subnet s) = False |
  allowed-subnet-flow InboundRouter \cdot = True |
  allowed-subnet-flow Unassigned Unassigned = True |
  allowed-subnet-flow Unassigned InboundRouter = True |

24
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) "→" 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⟩
⟨proof⟩
⟨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:
⟨proof⟩

interpretation Subnets: SecurityInvariant-ACS
where default-node-properties = SINVAR-Subnets2.default-node-properties
and sinvar = SINVAR-Subnets2.sinvar
⟨proof⟩

lemma TopoS-Subnets2: SecurityInvariant sinvar default-node-properties receiver-violation
4.6 Stricter Bell LaPadula Security Invariant

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
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: ∨ (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.REFL-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) } })

⟨proof⟩

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

⟨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

⟨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) } })
Taints-offending-set
proof

(proof)

lemma TopoS-Tainting: SecurityInvariant sinvar default-node-properties receiver-violation
(proof)

end

end

theory SINVAR-BLBasic
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: \( \forall nP\ e1\ e2. \neg (\leq)\cdot\text{security-level} \Rightarrow \text{security-level} \Rightarrow \text{bool} \) \((nP\ e1)\ (nP\ e2) \Rightarrow \neg (\leq)\cdot\text{security-level} \)

lemma zero-default-candidate-rule: \( \forall (nP::(\forall v \Rightarrow \text{security-level}))\ e1\ e2. \neg (nP\ e1) \leq (nP\ e2) \Rightarrow \neg (nP\ e1) \leq 0 \)

lemma privacylevel-refl: \((\leq)\cdot\text{security-level} \Rightarrow \text{security-level} \Rightarrow \text{bool}\) \(e\ e\)

lemma BLP-ENF: SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar \((\leq)\)

lemma BLP-ENF-refl: SecurityInvariant-withOffendingFlows.ENF-refl sinvar \((\leq)\)

definition BLP-offending-set: \((\forall v \Rightarrow \text{security-level}) \Rightarrow (\forall v \times v)\cdot\text{set}\cdot\text{set}\) where

\[
\text{BLP-offending-set} G nP = (\text{if sinvar } G nP \text{ then } \\
\{\} \text{ else } \\
\{ e \in \text{ edges } G. \text{ case } e \Rightarrow (c1,e2) \Rightarrow (nP\ e1) > (nP\ e2) \})
\]


interpretation BLPbasic: SecurityInvariant-IFS sinvar default-node-properties

lemma TopoS-BLPBasic: SecurityInvariant sinvar default-node-properties receiver-violation

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

lemma sinvar-BLPbasic-tancl: \(\forall v \in \text{ nodes } G. \forall v' \in \text{ succ-tran } G v. (nP\ v) \leq (nP\ v')\)

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
```

```
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-mon 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. af-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

sinvar (λc1 c2. taints c1 − untaints c1 ⊆ taints c2)
⟨proof⟩ lemma Taints-ENF-refl: SecurityInvariant-withOffendingFlows.ENF-refl
sinvar (λc1 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 c1) ⊆ taints (nP c2)} })
Taints-offending-set
⟨proof⟩

interpretation Taints: SecurityInvariant-IFS sinvar default-node-properties
⟨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 ≡ (λn1 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)
  ⟨proof⟩

**lemma** privacylevel-refl: BLP-P e e
  ⟨proof⟩

**lemma** BLP-ENF: SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form sinvar BLP-P
lemma BLP-ENF-refl: SecurityInvariant-withOffendingFlows.\textit{ENF-refl} sinvar BLP-P

\begin{proof}
\end{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)\}\})


\begin{proof}
\end{proof}

interpretation BLPtrusted: SecurityInvariant-IFS
where default-node-properties = default-node-properties
and sinvar = sinvar

\begin{proof}
\end{proof}

lemma TopoS-BLPtrusted: SecurityInvariant sinvar default-node-properties receiver-violation

\begin{proof}
\end{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: \(\forall ts v. nP v = ts \rightarrow\) finite ts \(\Rightarrow\)
SINVARTainting.sinvar G nP \(\Rightarrow\) SINVAR-BLPbasic.sinvar G ((\(\lambda ts.\) card (ts \(\cap\) X)) \(\circ\) nP)

\begin{proof}
\end{proof}

lemma tainting-imp-blp-cutcard2: finite X \(\Rightarrow\)
SINVARTainting.sinvar G nP \(\Rightarrow\) SINVAR-BLPbasic.sinvar G ((\(\lambda ts.\) card (ts \(\cap\) X)) \(\circ\) nP)

\begin{proof}
\end{proof}

lemma \(\forall ts v. nP v = ts \rightarrow\) finite ts \(\Rightarrow\)
SINVARTainting.sinvar G nP \(\Rightarrow\) SINVAR-BLPbasic.sinvar G (card \(\circ\) nP)

\begin{proof}
\end{proof}
lemma \( \forall b \in \text{snd } \cdot \text{edges } G. \text{finite} \ (nP \ b) \implies \)
\[
\text{SINVAR-Tainting}. \text{sinvar } G \ nP \implies \text{SINVAR-BLPbasic}. \text{sinvar } G \ (\text{card } \circ \ nP)
\]
(proof)

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. \text{if } a \in \ ts \ \text{then } \text{1::security-level} \ \text{else } \text{0::security-level}
  shows SINVAR-Tainting.sinvar \ G \ nP \longleftrightarrow (\forall \ a. \text{SINVAR-BLPbasic}.sinvar \ G \ (\text{extract } a \circ nP))
  (proof)

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. \text{card} \ (A \cap \ ts)
  assumes finite: (\forall \ ts \ v. \ nP \ v = ts \implies \text{finite} \ ts)
  shows SINVAR-Tainting.sinvar \ G \ nP \longleftrightarrow (\forall A. \text{SINVAR-BLPbasic}.sinvar \ G \ (\text{extract } A \circ nP))
  (proof)

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 \( \forall \ ts \ v. \ nP \ v = ts \implies \text{finite} \ (\text{taints } ts) \implies \)
\[
\text{SINVAR-TaintingTrusted}. \text{sinvar } G \ nP \implies \text{SINVAR-BLPtrusted}. \text{sinvar } G \ ((\lambda \ ts. \{\text{security-level } = \text{card} \ (\text{taints } ts - \text{untaints } ts), \text{trusted } = (\text{untaints } ts \neq \{\})\}) \circ nP)
\]
(proof)

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

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-level = \text{card} (A \cap (\text{taints } ts - \text{untaints } ts))
    , \text{trusted }= (A \cap \text{untaints } ts) \neq \{\})
  assumes finite: (\forall \ ts \ v. \ nP \ v = ts \implies \text{finite} \ (\text{taints } ts)
  shows SINVAR-TaintingTrusted.sinvar \ G \ nP \longleftrightarrow (\forall A. \text{SINVAR-BLPtrusted}.sinvar \ G \ (\text{project } A \circ nP))
  (proof)
proof
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-graphs. We will always require the stricter
wf-list-graph, which implies wf-graph.

lemma wf-list-graph G \implies 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 \implies
  (sinvar-spec (list-graph-to-graph G) nP) = (sinvar-impl G nP)
  and
  offending-flows-spec-impl: wf-list-graph G \implies
  (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: \( \forall nP. \) 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) ∧
  (\( \forall (e1, e2) \in set f. \) ¬ 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: \( \forall G nP. \) wf-list-graph G = sinvar-spec (list-graph-to-graph G) nP = sinvar-impl G nP

37
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 × '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 (∆G. sinvar-impl G nP) fs keeps G =
    TopoS-withOffendingFlows.minimalize-offending-overapprox (∆G. sinvar-spec G nP) fs keeps
    (list-graph-to-graph G)
⟨proof⟩

With TopoS-Interface-impl.minimize-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 (∆G. 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⟩

39
6.0.3 Example

definition fabNet :: string list-graph where
edgesL = [("PresenceSensor", "SensorSink"), ("Webcam", "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
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 ≡ ("PresenceSensor" ⇒ 2,
"Webcam" ⇒ 3,
"SensorSink" ⇒ 3,
"Statistics" ⇒ 3)

value BLP-eval fabNet sensorProps-NMParams-try3

export-code SINVAR-LIB-BLPbasic in 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 Unassigned - = False
fun sinvar :: 'v graph ⇒ ('v ⇒ subnets) ⇒ bool where
  sinvar GnP = (∀ (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⟩
proof


(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:

(proof)

interpretation Subnets: SecurityInvariant-ACS
where default-node-properties = SINVAR-Subnets.default-node-properties
and sinvar = SINVAR-Subnets.sinvar

(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 \equiv
\{ 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,2,3,4,5,6,7,8,9,11,12,42],
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 ≡ (λe. 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 sinvar

(λe. SINVAR-Subnets.default-node-properties)

value sinvar

(λe. SINVAR-Subnets.default-node-properties)

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

\[
\text{datatype domainNameDept} = \text{Dept string domainNameDept (infixr \(\mathtt{−−}\) 65)} | \text{Leaf} \quad \text{leaf of the tree, end of all domainNames}
\]

Example: the CoffeeMachine of I8

\[
\text{value } \"i8\" \mathtt{−−} \text{"CoffeeMachine"} \mathtt{−−} \text{Leaf}
\]

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

\[
\text{datatype domainTree} = \text{Department string — division} \\
\text{domainTree list — sub divisions}
\]

one step in tree to find matching department

\[
\text{fun hierarchy-next :: domainTree list }\Rightarrow \text{domainNameDept }\Rightarrow \text{domainTree option where}
\]

\[
\text{hierarchy-next }\mathbf{[]}\mathbf{=}\text{None} | \\
\text{hierarchy-next }\mathbf{(s\#ss)}\text{Leaf} = \text{None} | \\
\text{hierarchy-next }\mathbf{(\text{Department }d\text{ ds)#ss} }\text{Dept }n\text{ ns} = (\text{if }d=n\text{ then Some }\text{Department }d\text{ ds} \text{else hierarchy-next ss (Dept }n\text{ ns)})
\]

Examples:

\[
\text{lemma hierarchy-next }\mathbf{[\text{Department }\"i20\" ]}, \text{Department }\"i8\" [\text{Department }\"CoffeeMachine\" ], \text{Department }\"TeaMachine\" ][] \quad (\"i8\" \mathtt{−−} \text{Leaf}) \\
= \text{Some }\text{Department }\"i8\" [\text{Department }\"CoffeeMachine\" ], \text{Department }\"TeaMachine\" ][]
\]

\[
\langle \text{proof} \rangle
\]

\[
\text{lemma hierarchy-next }\mathbf{[\text{Department }\"i20\" ]}, \text{Department }\"i8\" [\text{Department }\"CoffeeMachine\" ], \text{Department }\"TeaMachine\" ][] \quad (\"i8\" \mathtt{−−} \text{"whatsoever"} \mathtt{−−} \text{Leaf}) \\
= \text{Some }\text{Department }\"i8\" [\text{Department }\"CoffeeMachine\" ], \text{Department }\"TeaMachine\" ][]
\]

\[
\langle \text{proof} \rangle
\]

\[
\text{lemma hierarchy-next }\mathbf{[\text{Department }\"i20\" ]}, \text{Department }\"i8\" [\text{Department }\"CoffeeMachine\" ], \text{Department }\"TeaMachine\" ][] \quad \text{Leaf} \\
= \text{None }\langle \text{proof} \rangle
\]

\[
\text{lemma hierarchy-next }\mathbf{[\text{Department }\"i20\" ]}, \text{Department }\"i8\" [\text{Department }\"CoffeeMachine\" ], \text{Department }\"TeaMachine\" ][] \quad (\"i0\" \mathtt{−−} \text{Leaf}) \\
= \text{None }\langle \text{proof} \rangle
\]

Does a given domainNameDept match the specified tree structure?

\[
\text{fun valid-hierarchy-pos :: domainTree }\Rightarrow \text{domainNameDept }\Rightarrow \text{bool where}
\]

\[
\text{valid-hierarchy-pos (Department }d\text{ ds) Leaf = True }| \\
\text{valid-hierarchy-pos (Department }d\text{ ds) (Dept }n\text{ Leaf) = (d=n) }| \\
\text{valid-hierarchy-pos (Department }d\text{ ds) (Dept }n\text{ ns) = (n=d \wedge (case hierarchy-next }ds\text{ ns of} \\
\text{ None }\Rightarrow \text{False }| \\
\text{ Some }t \Rightarrow \text{valid-hierarchy-pos }t\text{ ns))}
\]

45
Examples:

```plaintext
lemma valid-hierarchy-pos (Department "TUM" []) Leaf (proof)
```

```plaintext
lemma valid-hierarchy-pos (Department "TUM" []) Leaf (proof)
```

```plaintext
lemma valid-hierarchy-pos (Department "TUM" []) ("TUM"--Leaf) (proof)
```

```plaintext
lemma valid-hierarchy-pos (Department "TUM" []) ("TUM"--"facilityManagement"--Leaf) = False (proof)
```

```plaintext
lemma valid-hierarchy-pos (Department "TUM" []) ("LMU"--Leaf) = False (proof)
```

```plaintext
lemma valid-hierarchy-pos (Department "TUM" [Department "i8" [], Department "i20" []]) ("TUM"--"i8"--Leaf) (proof)
```

```plaintext
lemma valid-hierarchy-pos
(Department "TUM" []
  Department "i8" []
  Department "TeaMachine" []
), Department "i20" []
) ("TUM"--"i8"--"CoffeeMachine"--Leaf) (proof)
```

```plaintext
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 incomparable-sup-is-Top: n1 ≠ n2 ⇒ n1 -- x ≤ z ⇒ n2 -- y ≤ z ⇒ z = Leaf (proof)
```

46
lemma common-inf-imp-comparable: (z::domainNameDept) ≤ a ⇒ z ≤ b ⇒ a ≤ b ∨ b ≤ a 
(proof)

lemma prepend-domain: a ≤ b ⇒ x -- a ≤ x -- b
(proof)

lemma unfold-domain-leq: y ≤ zn -- zns ⇒ ∃ yns. y = zn -- yns ∧ yns ≤ zns
(proof)

lemma less-eq-refl:
  fixes x :: domainNameDept
  shows x ≤ y ⇒ y ≤ z ⇒ x ≤ 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" -- "CoffeeMachine" -- Leaf) (proof)
lemma ("TUM" -- "i8" -- "CoffeeMachine" -- Leaf) ≤ ("TUM" -- 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 ≤ domainNameDeptChopOne 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"−−"z"−−"d"−−Leaf) = "a"−−Leaf ⟨proof⟩
lemma chop-sup ("a"−−"b"−−"c"−−Leaf) ("x"−−"z"−−"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 Making 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 \textit{DN-inf-commute}: \(\text{DN-inf } x \ y = \text{DN-inf } y \ x\)
(proof)

lemma \textit{DN-inf-is-inf}: \(\text{is-inf } x \ y \ (\text{DN-inf } x \ y)\)
(proof)

\begin{verbatim}
fun \textit{DN-sup} :: \textit{domainName} \Rightarrow \textit{domainName} \Rightarrow \textit{domainName} where
  \textit{DN-sup} \textit{Unassigned} \ a = \ a |
  \textit{DN-sup} \ a \ \textit{Unassigned} = \ a |
  \textit{DN-sup} (\textit{DN} \ a) (\textit{DN} \ b) = \textit{DN} (\textit{chop-sup} \ a \ b)
\end{verbatim}

lemma \textit{DN-sup-commute}: \(\text{DN-sup } x \ y = \text{DN-sup } y \ x\)
(proof)

lemma \textit{DN-sup-is-sup}: \(\text{is-sup } x \ y \ (\text{DN-sup } x \ y)\)
(proof)

\textit{domainName} is a Lattice:

\begin{verbatim}
instantiation \textit{domainName} :: \textit{lattice}
begin
  instance (proof)
end
\end{verbatim}

\begin{verbatim}
datatype \textit{domainNameTrust} = \textit{DN} (\textit{domainNameDept} \times \textit{nat}) | \textit{Unassigned}
\end{verbatim}

fun \textit{leq-domainNameTrust} :: \textit{domainNameTrust} \Rightarrow \textit{domainNameTrust} \Rightarrow \textit{domainNameTrust} \Rightarrow \textit{bool} (infixr \textit{\subseteq_{trust}} 65)
where
  \textit{leq-domainNameTrust} \textit{Unassigned} \ - = \ True |
  \textit{leq-domainNameTrust} \ - \ \textit{Unassigned} = \ False |
  \textit{leq-domainNameTrust} (\textit{DN} (\textit{dnA}, \textit{trustA})) (\textit{DN} (\textit{dnB}, \textit{trustB})) = (\textit{dnA} \leq (\textit{domainNameChop} \textit{dnB} \textit{trustB}))

lemma \textit{leq-domainNameTrust-refl}: \(x \subseteq_{trust} x\)
(proof)

lemma \textit{leq-domainNameTrust-NOT-trans}: \(\exists x \ y \ z. \ x \subseteq_{trust} y \land y \subseteq_{trust} z \land \neg x \subseteq_{trust} z\)
(proof)

lemma \textit{leq-domainNameTrust-NOT-antisym}: \(\exists x \ y. \ x \subseteq_{trust} y \land y \subseteq_{trust} x \land x \neq y\)
(proof)

6.2.4 The network security invariant

definition \textit{default-node-properties} :: \textit{domainNameTrust}
where \textit{default-node-properties} = \textit{Unassigned}
The sender is, noticing its trust level, on the same or higher hierarchy level as the receiver.

```plaintext
fun sinvar :: 'v graph ⇒ ('v ⇒ domainNameTrust) ⇒ bool where
sinvar G nP = (∀ (s, r) ∈ edges G. (nP r) ⊑trust (nP s))
```

A domain name must be in the supplied tree

```plaintext
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 )
```

```plaintext
lemma verify-globals (| nodes=set [1,2,3], edges=set [] |) (λn. default-node-properties) (Department "TUM"")
⟨proof⟩
```

```plaintext
definition receiver-violation :: bool where receiver-violation = False
```

```plaintext
thm SecurityInvariant-withOffendingFlows.sinvar-mono-def
```

```plaintext
lemma sinvar-mono: SecurityInvariant-withOffendingFlows.sinvar-mono sinvar
⟨proof⟩
```

```plaintext
interpretation SecurityInvariant-preliminaries
where sinvar = sinvar
⟨proof⟩
```

### 6.2.5 ENF

```plaintext
sinvar (λ s r. r ⊑trust s)
⟨proof⟩
```

```plaintext
lemma DomainHierarchyNG-ENF-refl: SecurityInvariant-withOffendingFlows.ENF-refl sinvar (λ s r. r ⊑trust s)
⟨proof⟩
```

```plaintext
lemma unassigned-default-candidate: ∀ nP s r. ¬ (nP r) ⊑trust (nP s) → ¬ (nP r) ⊑trust default-node-properties
⟨proof⟩
```

```plaintext
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) ⊑trust (nP e1) } })
```

```plaintext
sinvar = DomainHierarchyNG-offending-set
⟨proof⟩
```

```plaintext
lemma Unassigned-unique-default: otherbot ≠ default-node-properties
```
∃ G nP gP i f.
  \text{wf-graph } G \land
  \neg \text{sinvar } G nP \land
  f \in \text{SecurityInvariant-withOffendingFlows.set-offending-flows sinvar } G nP \land
  \text{sinvar (delete-edges } G f) nP \land
  (i \in \text{fst } f \land \text{sinvar } G (nP(i := \text{otherbot})))

\langle proof \rangle

\text{interpretation DomainHierarchyNg: SecurityInvariant-ACS}
\text{where} \text{default-node-properties } = \text{default-node-properties}
\text{and} \text{sinvar } = \text{sinvar}
\text{rewrites} \text{SecurityInvariant-withOffendingFlows.set-offending-flows sinvar } = \text{DomainHierarchyNg-offending-set}
\langle proof \rangle

\text{lemma TopoS-DomainHierarchyNg: SecurityInvariant sinvar default-node-properties receiver-violation}
\langle proof \rangle

\text{hide-const (open) sinvar receiver-violation}

\text{end theory SINVAR-DomainHierarchyNg-impl imports SINVAR-DomainHierarchyNg ../TopoS-Interface-impl begin}

6.2.6 SecurityInvariant DomainHierarchy List Implementation

\text{code-identifier code-module} \text{SINVAR-DomainHierarchyNg-impl } \Rightarrow \text{(Scala) } \text{SINVAR-DomainHierarchyNg}

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

\text{definition DomainHierarchyNg-sanity-check-config } :: \text{domainNameTrust list } \Rightarrow \text{domainTree } \Rightarrow \text{bool where}
  \text{DomainHierarchyNg-sanity-check-config host-attributes tree } = (\forall c \in \text{set host-attributes.})
  \text{case } c \text{ of Unassigned } \Rightarrow \text{True}
  \mid \text{DN (level, trust) } \Rightarrow \text{valid-hierarchy-pos tree level}

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

\text{lemma DomainHierarchyNg-sanity-check-config c tree } \Rightarrow
  \{ x. \exists v. nP v = x \text{ } \Rightarrow \text{set } c \Rightarrow
  \text{verify-globals } G nP \text{ 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) ⊑ 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-DomainHierarchyNG
SINVAR-DomainHierarchyNG.sinvar
Examples:

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

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

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

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

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

**hide-const (open) sinvar**

end

theory SINVAR-BLPtrusted-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 GnP = (∀ (e1,e2) ∈ set (edgesL G). (if trusted (nP e2) then True else security-level (nP e1) ≤ security-level (nP e2) ))
definition \textit{BLP-offending-list}:: $'v \text{ list-graph} \Rightarrow (v \Rightarrow \text{SINVAR-BLPtrusted.node-config}) \Rightarrow (v \times 'v) \text{ list list}$ where

\begin{align*}
\text{BLP-offending-list } G \ nP &= (\text{if } \text{sinvar } G \ nP \text{ then } \\
&\quad [] \\
&\quad \text{else } \\
&\quad [e \leftarrow \text{edgesL } G. \ \text{case } e \text{ of } (e1, e2) \Rightarrow \lnot \ \text{SINVAR-BLPtrusted.BLP-P} (nP e1) (nP e2)])
\end{align*}

\textbf{definition} \textit{NetModel-node-props} \ P = (\lambda i. \text{(case (node-properties } P) i \text{ of Some property } \Rightarrow \text{property } \\
&| \text{None } \Rightarrow \text{SINVAR-BLPtrusted.default-node-properties})

\textbf{lemma}[\text{code-unfold}]: \text{SecurityInvariant.node-props SINVAR-BLPtrusted.default-node-properties } P = \text{NetModel-node-props } P

\textbf{definition} \textit{BLP-eval} \ G \ P = (\text{wf-list-graph } G \ \land
\text{sinvar } G (\text{SecurityInvariant.node-props SINVAR-BLPtrusted.default-node-properties } P))

\textbf{interpretation} \textit{BLPtrusted-impl: TopoS-List-Impl}

\textbf{definition} \textit{SINVAR-LIB-BLPtrusted} :: $'v:: \text{vertex}, \text{SINVAR-BLPtrusted.node-config} \text{ TopoS-packed}$ where

\begin{align*}
\text{SINVAR-LIB-BLPtrusted} &\equiv \\
&\{ \text{nm-name = "BLPtrusted"}, \\
&\text{nm-receiver-violation = SINVAR-BLPtrusted.receiver-violation}, \\
&\text{nm-default = SINVAR-BLPtrusted.default-node-properties}, \\
&\text{nm-sinvar = sinvar}, \\
&\text{nm-offending-flows = BLP-offending-list}, \\
&\text{nm-node-props = NetModel-node-props}, \\
&\text{nm-eval = BLP-eval} \}
\end{align*}

\textbf{interpretation} \textit{SINVAR-LIB-BLPtrusted-interpretation: TopoS-modelLibrary SINVAR-LIB-BLPtrusted}

\begin{align*}
\text{SINVAR-BLPtrusted.sinvar} \\
\langle \text{proof}\rangle
\end{align*}

\textbf{6.2.9 BLPtrusted packing}

\textbf{definition} \textit{SINVAR-LIB-BLPtrusted} :: ($'v:: \text{vertex}, \text{SINVAR-BLPtrusted.node-config} \text{ TopoS-packed}$ where

\begin{align*}
\text{SINVAR-LIB-BLPtrusted} &\equiv \\
&\{ \text{nm-name = "BLPtrusted"}, \\
&\text{nm-receiver-violation = SINVAR-BLPtrusted.receiver-violation}, \\
&\text{nm-default = SINVAR-BLPtrusted.default-node-properties}, \\
&\text{nm-sinvar = sinvar}, \\
&\text{nm-offending-flows = BLP-offending-list}, \\
&\text{nm-node-props = NetModel-node-props}, \\
&\text{nm-eval = BLP-eval} \}
\end{align*}

\textbf{interpretation} \textit{SINVAR-LIB-BLPtrusted-interpretation: TopoS-modelLibrary SINVAR-LIB-BLPtrusted}

\begin{align*}
\text{SINVAR-BLPtrusted.sinvar} \\
\langle \text{proof}\rangle
\end{align*}

\textbf{6.2.10 Example}

\textbf{export-code} \textit{SINVAR-LIB-BLPtrusted in Scala}

\textbf{hide-const} \textbf{(open)} \textit{NetModel-node-props BLP-offending-list BLP-eval}
6.3 SecurityInvariant PolEnforcePointExtended

A PolEnforcePoint is an application-level central policy enforcement point. Legacy note: The old versions 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, a PolEnforcePointIN 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.

```plaintext
datatype seecw-member = PolEnforcePoint | PolEnforcePointIN | DomainMember | AccessibleMember | Unassigned
```

```plaintext
definition default-node-properties :: seecw-member
  where default-node-properties ≡ Unassigned
```

```plaintext
fun allowed-secw-flow :: seecw-member ⇒ seecw-member ⇒ bool
  where allowed-secw-flow PolEnforcePoint - = True |
    allowed-secw-flow PolEnforcePointIN - = True |
    allowed-secw-flow DomainMember DomainMember = False |
    allowed-secw-flow DomainMember - = True |
    allowed-secw-flow AccessibleMember DomainMember = False |
    allowed-secw-flow AccessibleMember - = True |
    allowed-secw-flow Unassigned Unassigned = True |
    allowed-secw-flow Unassigned PolEnforcePointIN = True |
    allowed-secw-flow Unassigned AccessibleMember = True |
    allowed-secw-flow Unassigned - = False
```

```plaintext
fun sinvar :: 'v graph ⇒ ('v ⇒ seecw-member) ⇒ bool
  where sinvar G nP = (∀ (e1,e2) ∈ edges G. e1 ≠ e2 → allowed-secw-flow (nP e1) (nP e2))
```

```plaintext
definition receiver-violation :: bool
  where receiver-violation = False
```

6.3.1 Preliminaries

```plaintext
lemma sinvar-mono: SecurityInvariant-withOffendingFlows.sinvar-mono sinvar
  ⟨proof⟩
```

```plaintext
interpretation SecurityInvariant-preliminaries
  where sinvar = sinvar
  ⟨proof⟩
```
6.3.2 ENF

  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) } })
  sinvar = PolEnforcePointExtended-offending-set
  ⟨proof⟩

interpretation PolEnforcePointExtended: SecurityInvariant-ACS
where default-node-properties = default-node-properties
and sinvar = sinvar
  ⟨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) ] ])

57
definition NetModel-node-props P = (\lambda i. \text{(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 \land
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 :: (\text{\text{'e::vertex, seegw-member}) TopoS-packed where}
SINVAR-LIB-PolEnforcePointExtended ≡
(\text{\text{| nm-name = "PolEnforcePointExtended",}}
\text{\text{nm-receiver-violation = SINVAR-SecGwExt.receiver-violation,}}
\text{\text{nm-default = SINVAR-SecGwExt.default-node-properties,}}
\text{\text{nm-sinvar = sinvar,}}
\text{\text{nm-offending-flows = PolEnforcePointExtended-offending-list,}}
\text{\text{nm-node-props = NetModel-node-props,}}
\text{\text{nm-eval = PolEnforcePoint-eval}}
)

interpretation SINVAR-LIB-PolEnforcePointExtended-interpretation: TopoS-modelLibrary SINVAR-LIB-PolEnforcePoint
(proof)

Examples

definition example-net-seeqw :: nat list-graph where
example-net-seeqw ≡ (\text{| 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-seeqw

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

export-code sinvar in SML

definition test = sinvar (\text{| nodesL=[1::nat], edgesL=[]}) (\lambda -. SINVAR-SecGwExt.default-node-properties)

58
export-code test in SML
value sinvar (\nodesL=1::nat, edgesL=[] ) (\.-. SINVARS-Ext.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 SINVARS-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

  ⟨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:
  ⟨proof⟩

interpretation Sink: SecurityInvariant-IFS
where default-node-properties = default-node-properties
and sinvar = sinvar
  ⟨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 \( \text{NetModel-node-props} \ P = (\lambda \ i. \ \text{(case \ (node-properties \ P) \ i \ of \ Some \ property \Rightarrow \ property} \\
| \ None \Rightarrow \ \text{SINVAR-Sink.default-node-properties})) \)

lemma [code-unfold]: \( \text{SecurityInvariant.node-props} \ \text{SINVAR-Sink.default-node-properties} \ P = \text{NetModel-node-props} \ P \)

(proof)

definition \( \text{Sink-eval} \ G \ P = (\text{wf-list-graph} \ G \ \land \ \text{sinvar} \ G (\text{SecurityInvariant.node-props} \ \text{SINVAR-Sink.default-node-properties} \ P)) \)

interpretation \( \text{Sink-impl:TopoS-List-Impl} \)

where default-node-properties=\( \text{SINVAR-Sink.default-node-properties} \)

and sinvar-spec=\( \text{SINVAR-Sink.sinvar} \)

and sinvar-impl=\( \text{sinvar} \)

and receiver-violation=\( \text{SINVAR-Sink.receiver-violation} \)

and offending-flows-impl=\( \text{Sink-offending-list} \)

and node-props-impl=\( \text{NetModel-node-props} \)

and eval-impl=\( \text{Sink-eval} \)

(proof)

6.4.4 Sink packing

definition \( \text{SINVAR-LIB-Sink} :: (v::vertex, \text{node-config}) \text{TopoS-packed} \)

where \( \text{SINVAR-LIB-Sink} \equiv (\lambda \ nm-name= "Sink", \\
\text{nm-receiver-violation} = \text{SINVAR-Sink.receiver-violation}, \\
\text{nm-default} = \text{SINVAR-Sink.default-node-properties}, \\
\text{nm-sinvar} = \text{sinvar}, \\
\text{nm-offending-flows} = \text{Sink-offending-list}, \\
\text{nm-node-props} = \text{NetModel-node-props}, \\
\text{nm-eval} = \text{Sink-eval}) \)

interpretation \( \text{SINVAR-LIB-Sink-interpretation: TopoS-modelLibrary} \ \text{SINVAR-LIB-Sink} \)

\( \text{SINVAR-Sink.sinvar} \)

(proof)

Examples

definition \( \text{example-net-sink} :: \text{nat list-graph} \)

where \( \text{example-net-sink} \equiv ([ \ \text{nodesL} = [1::nat,2,3,8,11,12], \\
\text{edgesL} = [(1,8),(1,2),(2,8),(3,8),(4,8),(2,3),(3,2),(11,8),(12,8),(11,12),(1,12)]]) \)

value \( \text{wf-list-graph} \ \text{example-net-sink} \)

definition \( \text{example-conf-sink} \)

where \( \text{example-conf-sink} \equiv (\lambda e. \ \text{SINVAR-Sink.default-node-properties})(8:= \text{Sink}, 2:= \text{SinkPool}, 3:= \text{SinkPool}, \\
4:= \text{SinkPool}) \)

value \( \text{sinvar} \ \text{example-net-sink} \ \text{example-conf-sink} \)

value \( \text{Sink-offending-list} \ \text{example-net-sink} \ \text{example-conf-sink} \)

definition \( \text{example-net-sink-invalid} \)

where \( \text{example-net-sink-invalid} \equiv \text{example-net-sink}(\text{edgesL} := (2,1)#(8,11)#(8,2)#(\text{edgesL} \ \text{example-net-sink})) \)
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) } )
⟨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))

63
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 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 ≡ (∀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}]
6.6 SecurityInvariant CommunicationPartners

Idea of this security invariant: 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 ⇒ 'v ⇒ 'v ⇒ 'v ⇒ 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 ∈ set M) |
  allowed-flow (Master -) s (Master M) r = (s ∈ set M) |
  allowed-flow (Master -) - Care - = True |
  allowed-flow (Master -) - DontCare - = True

fun sinvar :: 'v graph ⇒ ('v ⇒ 'v ⇒ 'v ⇒ 'v ⇒ bool) ⇒ bool where
  sinvar G nP = (∀ (s,r) ∈ edges G. s ≠ r → allowed-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 \( \text{sinvar} = \text{sinvar} \)

\( \langle \text{proof} \rangle \)

### 6.6.2 ENRnr

**Lemma** CommunicationPartners-ENRnrSR: \( \text{SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-not-refl-SR} \)

\( \langle \text{proof} \rangle \)

**Lemma** Unassigned-weakrefl: \( \forall \ s \ r. \ \text{allowed-flow} \ \text{DontCare} \ s \ \text{DontCare} \ r \)

\( \langle \text{proof} \rangle \)

**Lemma** Unassigned-botdefault: \( \forall \ s \ r. \ (nP \ r) \neq \text{DontCare} \rightarrow \neg \ \text{allowed-flow} \ (nP \ s) \ (nP \ r) \rightarrow \neg \ \text{allowed-flow} \ \text{DontCare} \ (nP \ r) \)

\( \langle \text{proof} \rangle \)

**Lemma** ~ allowed-flow DontCare s (Master M) r \( \langle \text{proof} \rangle \)

**Lemma** ~ allowed-flow any s (Master []) r \( \langle \text{proof} \rangle \)

**Lemma** All-to-Unassigned: \( \forall \ s \ r. \ \text{allowed-flow} \ (nP \ s) \ s \ (nP \ r) \ r \rightarrow \neg \ \text{allowed-flow} \)

\( \langle \text{proof} \rangle \)

**Definition** CommunicationPartners-offending-set:: \( 'v \ \text{graph} \Rightarrow ('v \Rightarrow 'v \ \text{node-config}) \Rightarrow ('v \times 'v) \set \) where

\[ \text{CommunicationPartners-offending-set} \ G \ nP = \begin{cases} 
\{ \} 
& \text{if} \ \text{sinvar} \ G \ nP \ \text{then} \\
\{ \{ e \in \text{edges} \ G. \ \text{case} \ e \ \text{of} \ (e1,e2) \Rightarrow e1 \neq e2 \land \neg \ \text{allowed-flow} \ (nP \ e1) \ e1 \ (nP \ e2) \ e2 \} \} 
& \text{else} 
\end{cases} \)

**Lemma** CommunicationPartners-offending-set: \( \text{SecurityInvariant-withOffendingFlows.set-offending-flows sinvar} = \text{CommunicationPartners-offending-set} \)

\( \langle \text{proof} \rangle \)

**Interpretation** CommunicationPartners: \( \text{SecurityInvariant-ACS} \)

where \( \text{default-node-properties} = \text{default-node-properties} \)

and \( \text{sinvar} = \text{sinvar} \)

rewrites \( \text{SecurityInvariant-withOffendingFlows.set-offending-flows sinvar} = \text{CommunicationPartners-offending-set} \)

\( \langle \text{proof} \rangle \)

**Lemma** TopoS-SubnetsInGW: \( \text{SecurityInvariant sinvar default-node-properties receiver-violation} \)

\( \langle \text{proof} \rangle \)

Example:

**Lemma** \( \text{sinvar} \ (\| \text{nodes} = \{ "db1", "db2", "h1", "h2", "foo", "bar" \},\)\)

\( \text{edges} = \{ ("h1", "db1"), ("h2", "db1"), ("h1", "h2"),\)\)

\( ("db1", "h1"), ("db1", "foo"), ("db1", "db2"), ("db1", "db1"),\)\)

\( ("h1", "foo"), ("foo", "h1"), ("foo", "bar") \} \))

\( (((\lambda \ h. \ \text{default-node-properties})("h1" := \text{Care})("h2" := \text{Care}))\)

\( ("db1" := \text{Master } ["h1", "h2"])("db2" := \text{Master } ["db1"])) \) \( \langle \text{proof} \rangle \)

hide-fact (open) \( \text{sinvar-mono} \)

hide-const (open) \( \text{sinvar receiver-violation default-node-properties} \)
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-properties)
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,
        |)

67
\[
\begin{align*}
nm\text{-offending-flows} &= \text{CommunicationPartners-offending-list}, \\
nm\text{-node-props} &= \text{NetModel-node-props}, \\
nm\text{-eval} &= \text{CommunicationPartners-eval}
\end{align*}
\]

\textbf{interpretation} \textit{SINV\textsc{AR-LIB-CommunicationPartners-interpretation}}: TopoS\textsc{-modelLibrary} SINV\textsc{AR-LIB-CommunicationPartners} \text{sinvar}

\textit{proof}

Examples

\begin{itemize}
\item hide-const \textbf{(open)} \textit{NetModel}\text{-node-props}
\item hide-const \textbf{(open)} \textit{sinvar}
\end{itemize}

end

\textbf{theory} \textit{SINV\textsc{AR-NoRefl}}

\textbf{imports} ../\textit{TopoS-Helper}

\begin{itemize}
\item begin
\end{itemize}

\section{6.7 SecurityInvariant \textit{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.

\begin{itemize}
\item \textbf{datatype} \textit{node-config} = \textit{NoRefl} | \textit{Refl}
\item \textbf{definition} \textit{default-node-properties} :: \textit{node-config}
\hspace{1em} where \textit{default-node-properties} = \textit{NoRefl}
\end{itemize}

\begin{itemize}
\item \textbf{fun} \textit{sinvar} :: \textit{\textquote{\textquote{\textquote{v graph} \Rightarrow (v \Rightarrow node-config)} \Rightarrow bool}}
\hspace{1em} \textit{where} \textit{sinvar G nP} = (\forall (s, r) \in \textit{edges G}. s = r \rightarrow nP s = \textit{Refl})
\end{itemize}

\begin{itemize}
\item \textbf{definition} \textit{receiver-violation} :: \textit{bool}
\hspace{1em} \textit{where} \textit{receiver-violation} = \textit{False}
\end{itemize}

\subsection{6.7.1 Preliminaries}

\begin{itemize}
\item \textbf{lemma} \textit{sinvar-mono}: \textit{SecurityInvariant-withOffendingFlows.sinvar-mono sinvar}
\hspace{1em} \textit{proof}
\end{itemize}

\textbf{interpretation} \textit{SecurityInvariant-preliminaries}

\begin{itemize}
\item \textit{where} \textit{sinvar} = \textit{sinvar}
\hspace{1em} \textit{proof}
\end{itemize}

\begin{itemize}
\item \textbf{lemma} \textit{NoRfl-ENRsr}: \textit{SecurityInvariant-withOffendingFlows.sinvar-all-edges-normal-form-sr sinvar}
\hspace{1em} (\lambda nP_s s nP_r r. s = r \rightarrow nP_s = \textit{Refl})
\hspace{1em} \textit{proof}
\end{itemize}

\begin{itemize}
\item \textbf{definition} \textit{NoRfl-offending-set} :: \textit{\textquote{\textquote{\textquote{v graph} \Rightarrow (v \Rightarrow node-config)} \Rightarrow (v \times v) set set}}
\hspace{1em} \textit{where} \textit{NoRfl-offending-set} G nP = (if sinvar G nP then \\
\hspace{2em} \{\} \\
\hspace{2em} else}
\end{itemize}
\{ \{ e \in \text{edges } G. \text{ case } e \text{ of } (e_1,e_2) \Rightarrow e_1 = e_2 \land nP e_1 = \text{NoRefl} \} \} \}

\textbf{thm} \text{SecurityInvariant-withOffendingFlows.\text{ENFsr-offending-set}[OF NoRefl-\text{ENRsr}]} \]

\textbf{lemma} \text{NoRefl-offending-set}: \text{SecurityInvariant-withOffendingFlows.\text{set-offending-flows \text{sinvar} = NoRefl-offending-set}}

\textbf{lemma} \text{NoRefl-unique-default}: \forall G f nP i. \text{wf-graph } G \land f \in \text{set-offending-flows } G nP \land i \in \text{fst } \cdot f \longrightarrow \neg \text{sinvar } G (nP(i := \text{otherbot})) \Rightarrow \text{otherbot} = \text{NoRefl}

\textbf{interpretation} \text{NoRefl}: \text{SecurityInvariant-ACS}

\textbf{where} \text{default-node-properties = default-node-properties}

\textbf{and} \text{sinvar = sinvar}

\textbf{rewrites} \text{SecurityInvariant-withOffendingFlows.\text{set-offending-flows \text{sinvar} = NoRefl-offending-set}}

\textbf{It can also be interpreted as IFS}

\textbf{lemma} \text{NoRefl-SecurityInvariant-IFS}: \text{SecurityInvariant-IFS \text{sinvar default-node-properties}}

\textbf{lemma} \text{TopoS-NoRefl}: \text{SecurityInvariant sinvar default-node-properties receiver-violation)}

\textbf{hide-fact} (open) \text{sinvar-mono}

\textbf{hide-const} (open) \text{sinvar receiver-violation default-node-properties}

\textbf{end}

\textbf{theory} \text{SINVAR-NoRefl-impl}

\textbf{imports} \text{SINVAR-NoRefl ../TopoS-Interface-impl}

\textbf{begin}

\textbf{code-identifier code-module} \text{SINVAR-NoRefl-impl => (Scala) SINVAR-NoRefl}

\textbf{6.7.2 SecurityInvariant NoRefl List Implementation}

\textbf{fun} \text{sinvar :: }'v \text{ list-graph } \Rightarrow (\forall \text{ node-config } ) \Rightarrow \text{bool where}

\text{sinvar } G nP = (\forall (s,r) \in \text{set } (\text{edgesL } G). s = r \longrightarrow nP s = \text{Refl})

\textbf{definition} \text{NoRefl-offending-list:: }'v \text{ list-graph } \Rightarrow (\forall \text{ node-config } ) \Rightarrow (\forall 'v \times 'v) \text{ list where}

\text{NoRefl-offending-list } G nP = (\text{if } \text{sinvar } G nP \text{ then}

\text{[ ]}

\text{else}

\text{[ [e \leftarrow \text{edgesL } G. \text{ case } e \text{ of } (e_1,e_2) \Rightarrow e_1 = e_2 \land nP e_1 = \text{NoRefl} ] ]})

\textbf{definition} \text{NetModel-node-props } P = (\lambda i. \text{ (case } \text{node-properties } P \text{ i of Some } \text{property } \Rightarrow \text{property}

\text{None } \Rightarrow \text{SINVAR-NoRefl.default-node-properties)})

69
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 ≡
(f
tm-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,

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 ≡ (f
odesL = [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 ≡ ((λe. SINVAR-NoRefl.default-node-properties)(2:= Refl))

lemma sinvar example-net example-conf (proof)

lemma NoRefl-offending-list example-net (λe. 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
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))


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
6.7.6 Example

context
begin
private definition tainting-example :: string list-graph where

tainting-example ≡ |
    "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 1 2"),
    ("read 3", "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 in Scala

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

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 ⊆ C ←→ (∀ a∈A. a ∈ C ∨ a ∈ B) (proof)

lemma ¬(A - B ⊆ C) ←→ (∃ a ∈ A. a ∉ C ∧ a ∉ B) (proof)

fun sinvar :: 'v list-graph ⇒ ('v ⇒ SINVAR-TaintingTrusted.taints) ⇒ bool where
\[
\text{sinvar } G \; nP = (\forall (v_1,v_2) \in \text{set } \text{edges}_L G). \; \text{taints } (nP \; v_1) - \text{untaints } (nP \; v_1) \subseteq \text{taints } (nP \; v_2)
\]

### export-code sinvar in SML

**value**[code] sinvar (\[] nodesL = [], edgesL = [] \) (\lambda. SINVAR-TaintingTrusted.default-node-properties)

**lemma** sinvar (\[] nodesL = [], edgesL = [] \) (\lambda. SINVAR-TaintingTrusted.default-node-properties)

\[\langle \text{proof} \rangle\]

### definition TaintingTrusted-offending-list

\[
\text{TaintingTrusted-offending-list } G \; nP = \begin{cases} 
\text{if sinvar } G \; nP \text{ then } & \text{[]} \\
\text{else } & \text{[ } e \leftarrow \text{edges}_L G. \text{ case } e \text{ of } (v_1,v_2) \Rightarrow \neg(\text{taints } (nP \; v_1) - \text{untaints } (nP \; v_1) \subseteq \text{taints } (nP \; v_2)) \text{ [ ] } \text{ ]} 
\end{cases}
\]

### export-code TaintingTrusted-offending-list in SML

### definition NetModel-node-props P

\[
(\lambda i. \begin{case} \text{node-properties } P \text{ i of } \begin{cases} \text{Some property } & \Rightarrow \text{property} \\
\text{None } & \Rightarrow \text{SINVAR-TaintingTrusted.default-node-properties} \end{cases} \end{case})
\]

**lemma**[code-unfold]: SecurityInvariant.node-props SINVAR-TaintingTrusted.default-node-properties P = NetModel-node-props P

\[\langle \text{proof} \rangle\]

### definition TaintingTrusted-eval G P

\[
\text{sinvar } G \; (\text{SecurityInvariant.node-props } \text{SINVAR-TaintingTrusted.default-node-properties } P)
\]

### interpretation TaintingTrusted-impl: TopoS-List-Impl

\[\text{where default-node-properties } = \text{SINVAR-TaintingTrusted.default-node-properties} \]

\[\text{and sinvar-spec } = \text{SINVAR-TaintingTrusted.sinvar} \]

\[\text{and sinvar-impl } = \text{sinvar} \]

\[\text{and receiver-violation } = \text{SINVAR-TaintingTrusted.receiver-violation} \]

\[\text{and offending-flows-impl } = \text{TaintingTrusted-offending-list} \]

\[\text{and node-props-impl } = \text{NetModel-node-props} \]

\[\text{and eval-impl } = \text{TaintingTrusted-eval} \]

\[\langle \text{proof} \rangle\]

### 6.7.8 TaintingTrusted packing

**definition** SINVAR-LIB-TaintingTrusted :: (\text{v::vertex, SINVAR-TaintingTrusted.taints}) TopoS-packed

\[\text{where} \]

\[\text{SINVAR-LIB-TaintingTrusted } \equiv \]

\[\text{\{} \text{nm-name} = "\text{TaintingTrusted}".} \]

\[\text{nm-receiver-violation }= \text{SINVAR-TaintingTrusted.receiver-violation}.\]

\[\text{nm-default }= \text{SINVAR-TaintingTrusted.default-node-properties}.\]

\[\text{nm-sinvar }= \text{sinvar}.\]

\[\text{nm-offending-flows }= \text{TaintingTrusted-offending-list}.\]
\[
\text{nm-node-props} = \text{NetModel-node-props}, \\
\text{nm-eval} = \text{TaintingTrusted-eval}
\]

\begin{verbatim}
interpretation SINVAR-LIB-BLPbasic-interpretation: TopoS-modelLibrary SINVAR-LIB-TaintingTrusted
  SINVAR-TaintingTrusted.sinvar
\end{verbatim}

6.7.9 Example

context
begin
private definition tainting-example :: string list-graph where
  tainting-example ≡ \{
    "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.
  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-exampl...tainting-example-props (proof)
end

export-code SINVAR-LIB-TaintingTrusted in Scala
export-code SINVAR-LIB-TaintingTrusted in 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
  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 = (λv. 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)] |)
(λe. 3)
value sinvar
(| nodesL = [1::nat,2,3,4,8,9,10], edgesL = [(1,2), (2,3), (3,4), (8,9),(9,8)] |)
(λe. 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 = (λv. 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.default-node-properties.
(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 (λe. 0) in map (λv. nP v) (nodesL G))
value let G = ([\(\|\) nodesL = [1::nat, 2, 3, 4], edgesL = [[1, 1]]) in (let nP = dependability-fix-nP G \((\lambda e. 0)\) in map (\(\lambda v. nP v\) \(\) nodesL G))

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

definition NetModel-node-props P = (\(\lambda i.\) (case (node-properties P) i of Some property \(\Rightarrow\) property \(\mid\) None \(\Rightarrow\) 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 \(\land\)
sinvar G (SecurityInvariant.node-props SINVAR-Dependability.default-node-properties P))

lemma sinvar-correct: wf-list-graph G \(\Rightarrow\) 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 \(\equiv\)
\(\langle\) 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 \(\rangle\)

interpretation SINVAR-LIB-Dependability-interpretation: TopoS-modelLibrary SINVAR-LIB-Dependability
Example:

```plaintext
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:= 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 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

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 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 n' (undirected-reachable G n)) ⊆ {Unrelated})

lemma sinvar G nP ✴→
  (∀ n ∈ {v' ∈ (nodes G). (nP v') = Interfering}. {nP 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 = vertex-1 \land b = vertex-2\} = \{(vertex-2, vertex-1)\} \langle proof \rangle
**Lemma** tmp6:: \{(vertex-1, vertex-2), (vertex-2, vertex-1)\}^+ = \{(vertex-1, vertex-1), (vertex-2, vertex-2), (vertex-1, vertex-2), (vertex-2, vertex-1)\} \langle proof \rangle
**Lemma** tmp2:: (insert (vertex-1, vertex-2) \{(b, a). a = vertex-1 \land b = vertex-2\})^+ = \{(vertex-1, vertex-1), (vertex-2, vertex-2), (vertex-1, vertex-2), (vertex-2, vertex-1)\} \langle proof \rangle
**Lemma** tmp4:: \{(e1, e2). e1 = vertex-1 \land e2 = vertex-2 \land (e1 = vertex-1 \rightarrow e2 \neq vertex-2)\} = {} \langle proof \rangle
**Lemma** tmp5:: \{(b, a). a = vertex-1 \land b = vertex-2 \lor a = vertex-1 \land b = vertex-2 \land (a = vertex-1 \rightarrow b \neq vertex-2)\} = \{(vertex-2, vertex-1)\} \langle proof \rangle
**Lemma** unique-default-example:: undirected-reachable \{(nodes = \{vertex-1, vertex-2\}, edges = \{(vertex-1, vertex-2)\}\} vertex-1 = \{vertex-2\} \langle proof \rangle
**Lemma** unique-default-example-hlp1:: delete-edges \{(nodes = \{vertex-1, vertex-2\}, edges = \{(vertex-1, vertex-2)\}\} vertex-1 = \{\}

**Proof**

**Lemma** unique-default-example-2:: undirected-reachable \{(nodes = \{vertex-1, vertex-2\}, edges = \{(vertex-1, vertex-2)\}\} vertex-1 = \{\}

**Proof**

**Lemma** unique-default-example-3:: undirected-reachable \{(nodes = \{vertex-1, vertex-2\}, edges = \{(vertex-1, vertex-2)\}\} vertex-2 = \{\}

**Proof**

**Lemma** unique-default-example-4:: (undirected-reachable \{add-edge vertex-1 vertex-2\} \{(nodes = \{vertex-1, vertex-2\}, edges = \{(vertex-1, vertex-2)\}\} vertex-1 = \{vertex-2\}

**Proof**

**Lemma** unique-default-example-5:: (undirected-reachable \{add-edge vertex-1 vertex-2\} \{(nodes = \{vertex-1, vertex-2\}, edges = \{(vertex-1, vertex-2)\}\} vertex-2 = \{vertex-1\}

**Proof**

**Lemma** empty-undirected-reachable-false:: \(xb \in \text{undirected-reachable (delete-edges G (edges G)) na}

\langle proof \rangle

**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)

(proof)
lemma sinvar-eq-help2: set l = \{Unrelated\} \implies \text{remdups} l = \{Unrelated\}

(proof)

lemma sinvar-eq-help3: (let result = \text{remdups} (map nP (undirected-reachable G n)) in result = [] \lor result = \{Unrelated\}) = (set (map nP (undirected-reachable G n)) \subseteq \{Unrelated\})

(proof)

lemma sinvar-list-eq-set: sinvar = sinvar-set

(proof)

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))


(proof)

definition NonInterference-eval G P = (wf-list-graph G \land

sinvar G (SecurityInvariant.node-props SINVAR-NonInterference.default-node-properties P))

lemma sinvar-correct: wf-list-graph G \implies 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), (4, 5), (8, 9), (9, 8)]⟩
private definition example-conf = ⟨
   (λe. 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))

**interpretation** SecurityInvariant-preliminaries
  where sinvar = sinvar

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

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

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

**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-ACLnotCommunicateWith

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 ∈ (succ-tran G v). a /∈ (nP v))
definition NetModel-node-props \((P::\langle v::\text{vertex}, \ 'v \text { set} \rangle \ TopoS-\text{Params}) = \\
(\lambda \ i \ . \ (\text{case } \text{(node-properties } P) \ i \ \text{of Some property } \Rightarrow \text{property } | \text{None } \Rightarrow \text{SINVAR-ACLnotCommunicateWith.default-node-properties}) \\
P = \text{NetModel-node-props } P \langle \text{proof} \rangle \)

definition ACLnotCommunicateWith-offending-list = Generic-offending-list \text{sinvar}

definition ACLnotCommunicateWith-eval \(G \ P = (\text{wf-list-graph } G \ \land \\
\text{sinvar } G \ (\text{SecurityInvariant.node-props } \text{SINVAR-ACLnotCommunicateWith.default-node-properties} \ P)) \langle \text{proof} \rangle \\

lemma \text{sinvar-correct}: \text{wf-list-graph } G \implies \text{SINVAR-ACLnotCommunicateWith.sinvar } (\text{list-graph-to-graph} \ G) \ nP = \text{sinvar } G \ nP \langle \text{proof} \rangle \\

interpretation ACLnotCommunicateWith-impl: TopoS-List-Impl \\
where default-node-properties=\text{SINVAR-ACLnotCommunicateWith.default-node-properties} \\
and sinvar-spec=\text{SINVAR-ACLnotCommunicateWith.sinvar} \\
and sinvar-impl=\text{sinvar} \\
and receiver-violation=\text{SINVAR-ACLnotCommunicateWith.receiver-violation} \\
and offending-flows-impl=ACLnotCommunicateWith-offending-list \\
and node-props-impl=NetModel-node-props \\
and eval-impl=ACLnotCommunicateWith-eval \langle \text{proof} \rangle \\

6.11.2 packing 

definition SINVAR-LIB-ACLnotCommunicateWith:: \((v::\text{vertex}, \ 'v \text { set} \rangle \ TopoS-packed \ \text{where} \\
\text{SINVAR-LIB-ACLnotCommunicateWith } \equiv \\
\{ \ \text{nm-name } = "\text{ACLnotCommunicateWith}" , \\
\text{nm-receiver-violation } = \text{SINVAR-ACLnotCommunicateWith.receiver-violation} , \\
\text{nm-default } = \text{SINVAR-ACLnotCommunicateWith.default-node-properties} , \\
\text{nm-sinvar } = \text{sinvar} , \\
\text{nm-offending-flows } = \text{ACLnotCommunicateWith-offending-list} , \\
\text{nm-node-props } = \text{NetModel-node-props} , \\
\text{nm-eval } = \text{ACLnotCommunicateWith-eval} \} \\
\} \\
interpretation SINVAR-LIB-ACLnotCommunicateWith-interpretation: TopoS-modelLibrary SINVAR-LIB-ACLnotCommunicateWith.sinvar \\
\langle \text{proof} \rangle \\

Examples 

hide-const (open) \text{NetModel-node-props} \\
hide-const (open) \text{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-properties)
lemma[case 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-ACLcommunicateWith
SINVAR-ACLcommunicateWith.sinvar
⟨proof⟩

Examples

context begin

1 can access 2 and 3
2 can access 3

private lemma sinvar
\[
\begin{align*}
\text{\textit{netL} = \{ & 1::\text{nat}, 2, 3\},} \\
\text{\textit{edgesL} = \{(1,2), (2,3), (3,4)\}} \\
(((\lambda v. \text{SINVAR-ACLcommunicateWith}.\text{default-node-properties}) \\
(1 := [2,3])) \\
(2 := [3])) \langle \text{proof} \rangle
\end{align*}
\]

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** \textit{ACLcommunicateWith-offending-list} 
\[
\begin{align*}
\text{\textit{netL} = \{ & 1::\text{nat}, 2, 3, 4\},} \\
\text{\textit{edgesL} = \{(1,2), (1,3), (2,3), (3,4)\}} \\
(((\lambda v. \text{SINVAR-ACLcommunicateWith}.\text{default-node-properties}) \\
(1 := [1,2,3])) \\
(2 := [1,2,3,4])) \\
(3 := [1,2,3,4])) \\
(4 := [1,2,3,4])) = \\
[[\langle 1, 2 \rangle], \langle 1, 3 \rangle, \langle 2, 3 \rangle, \langle 3, 4 \rangle]] \langle \text{proof} \rangle
\end{align*}
\]

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** \textit{ACLcommunicateWith-offending-list} 
\[
\begin{align*}
\text{\textit{netL} = \{ & 1::\text{nat}, 2, 3, 4\},} \\
\text{\textit{edgesL} = \{(1,2), (1,3), (2,3), (3,4)\}} \\
(((\lambda v. \text{SINVAR-ACLcommunicateWith}.\text{default-node-properties}) \\
(1 := [1,2,3])) \\
(2 := [1,2,3,4])) \\
(3 := [1,2,3,4])) \\
(4 := [1,2,3,4])) = \\
[[\langle 1, 2 \rangle], \langle 1, 3 \rangle, \langle 2, 3 \rangle, \langle 3, 4 \rangle]] \langle \text{proof} \rangle
\end{align*}
\]

end

\textit{hide-const (open) NetModel-node-props}
\textit{hide-const (open) sinvar}

end

\textit{theory SINVAR-Dependability-norefl}
\textit{imports ../TopoS-Helper}
begin

6.12 SecurityInvariant \textit{Dependability-norefl}

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

\textit{type-synonym dependability-level = nat}
\textit{definition default-node-properties :: dependability-level}
\textit{where default-node-properties \equiv 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) ∈ set (edgesL 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)] |⟩
  ⟨λe. 3⟩
value sinvar
  ⟨| nodesL = [1::nat,2,3,4,8,9,10], edgesL = [(1,2), (2,3), (3,4), (8,9),(9,8)] |⟩
  ⟨λe. 2⟩
**definition** Dependability-norefl-offending-list :: 'v list-graph ⇒ ('v ⇒ dependability-level) ⇒ ('v × '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 ⇒ property | None ⇒ 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 ⇒ 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-norefl

SINVAR-Dependability-norefl.sinvar

(proof)

hide-fact (open) sinvar-correct

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

89
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-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
Lib/Efficient-Distinct
HOL-Library.Code-Target-Nat

begin

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 \((\mathcal{V}\text{ graph} \Rightarrow \text{bool})\) list, i.e. a list of predicates over the policy. We need in instantiated security invariant, i.e. get rid of \(\mathcal{V}\) and \(\mathcal{A}\)

— An instance (configured) a security invariant I.e. a concrete security requirement, in different terminology.

record \((\mathcal{V})\) SecurityInvariant-configured =
  c-sinvar::\((\mathcal{V})\) graph \Rightarrow \text{bool}
  c-offending-flows::\((\mathcal{V})\) graph \Rightarrow (\mathcal{V} \times \mathcal{V}) \text{ set set}
  c-isIFS::\text{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 DomainHierarchy.

fun new-configured-SecurityInvariant ::
  \((((\mathcal{V}::\text{vertex})\ graph \Rightarrow (\mathcal{V} \Rightarrow \mathcal{A}) \Rightarrow \text{bool}) \times \mathcal{A} \times \text{bool} \times ((\mathcal{V} \Rightarrow \mathcal{A})) \Rightarrow (\mathcal{V}\text{SecurityInvariant-configured}))\) option where
  new-configured-SecurityInvariant (sinvar, defbot, receiver-violation, \(nP\)) =
(if SecurityInvariant sinvar defbot receiver-violation then
  Some (λG. sinvar G nP),
  c-offending-flows = (λG. 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 ⇒
c-sinvar (the (new-configured-SecurityInvariant (sinvar, defbot, receiver-violation, nP))) = (λG. sinvar G nP)
(proof)

lemma new-configured-TopoS-offending-flows-correct:
SecurityInvariant sinvar defbot receiver-violation ⇒
c-offending-flows (the (new-configured-SecurityInvariant (sinvar, defbot, receiver-violation, 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 = {} ] ⇒ 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 ] ⇒
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 fulfils its security requirement, the offending flows are always defined.

lemma defined-offending':
\[
\begin{align*}
\text{lemma } \text{subst-offending-flows: } & \forall nP. \text{SecurityInvariant-withOffendingFlows.set-offending-flows} (\lambda G nP. \text{c-sinvar } m G) \implies \{ nP = \text{c-offending-flows } m G \} \\
(\text{proof})
\end{align*}
\]

all the \text{SecurityInvariant-preliminaries} stuff must hold, for an arbitrary \( nP \)

\text{lemma } \text{SecurityInvariant-preliminaries}: \lambda (G::('v::vertex) graph) \ (nP::'v \Rightarrow 'a). \text{c-sinvar } m G

(\text{proof})

\text{lemma negative-mono:}
\[
\begin{align*}
\forall N E'. \text{wf-graph } (\{ \text{nodes } = N, \text{edges } = E' \}) \implies \\
E' \subseteq E \implies \neg \text{c-sinvar } m (\{ \text{nodes } = N, \text{edges } = E' \}) \implies \\
\neg \text{c-sinvar } m (\{ \text{nodes } = N, \text{edges } = E \})
\end{align*}
\]

(\text{proof})

7.1 Reusing Lemmata

\text{lemmas mono-extend-set-offending-flows =}

\[
\begin{align*}
[\text{wf-graph } (\{ \text{nodes } = V, \text{edges } = E \}); \ E' \subseteq E; \ F' \in \text{c-offending-flows } m (\{ \text{nodes } = V, \text{edges } = E' \})] \implies \\
\exists F \in \text{c-offending-flows } m (\{ \text{nodes } = V, \text{edges } = E \}). \ F' \subseteq F
\end{align*}
\]

\text{lemmas offending-flows-union-mono =}

\[
[\text{wf-graph } (\{ \text{nodes } = V, \text{edges } = E \}); \ E' \subseteq E] \implies \\
\bigcup \text{c-offending-flows } m (\{ \text{nodes } = V, \text{edges } = E \})
\]

\text{lemmas sinvar-valid-remove-flattened-offending-flows =}

\[
\text{wf-graph } (\{ \text{nodes } = \text{nodesG}, \text{edges } = \text{edgesG} \}) \implies \text{c-sinvar } m (\{ \text{nodes } = \text{nodesG}, \text{edges } = \text{edgesG} \} \\
\bigcup \text{c-offending-flows } m (\{ \text{nodes } = \text{nodesG}, \text{edges } = \text{edgesG} \}))
\]

\text{lemmas sinvar-valid-remove-SOME-offending-flows =}

\[
\text{c-offending-flows } m (\{ \text{nodes } = \text{nodesG}, \text{edges } = \text{edgesG} \} \neq \{ \}) \implies \text{c-sinvar } m (\{ \text{nodes } = \text{nodesG}, \text{edges } = \text{edgesG} \} \\
\text{ SOME } F. \ F \in \text{c-offending-flows } m (\{ \text{nodes } = \text{nodesG}, \text{edges } = \text{edgesG} \}))
\]

\text{lemmas sinvar-valid-remove-minimalize-offending-overapprox =}

\[
[\text{wf-graph } (\{ \text{nodes } = \text{nodesG}, \text{edges } = \text{edgesG} \}); \ \neg \text{c-sinvar } m (\{ \text{nodes } = \text{nodesG}, \text{edges } = \text{edgesG} \}); \ \text{set } Es = \text{edgesG}; \ \text{distinct } Es] \implies \text{c-sinvar } m (\{ \text{nodes } = \text{nodesG}, \text{edges } = \text{edgesG} \\} \\
\text{distinct E's }]
\]
set (SecurityInvariant-with-OffendingFlows.minimize-offending-overapprox (λG nP. c-sinvar m G) Es [] (nodes = nodesG, edges = edgesG) nP))

lemmas empty-offending-contra =
SecurityInvariant-with-OffendingFlows.empty-offending-contra where
sinvar = (λG 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 =

[wf-graph (nodes = V, edges = E); \bigcup c-offending-flows m (nodes = V, edges = E) ⊆ X]

⇒ \bigcup c-offending-flows m (nodes = V, edges = E ′) ⊆ X − E ′

lemmas Un-set-offending-flows-bound-minus-subseteq ′ =

[wf-graph (nodes = V, edges = E); \bigcup c-offending-flows m (nodes = V, edges = E) ⊆ X]

⇒ \bigcup c-offending-flows m (nodes = V, edges = E ′) ⊆ X − E ′

end

thm configured-SecurityInvariant-def

configured-SecurityInvariant m ≡ (∀ G. c-offending-flows m G = \{F. F ⊆ edges G \land \neg c-sinvar m G \land c-sinvar m (delete-edges G F) \land (\forall (e1, e2)\in F. \neg c-sinvar m (add-edge e1 e2 (delete-edges G F)))\}) \land (\forall N. wf-graph (nodes = N, edges = {})) ➝ c-sinvar m (nodes = N, edges = {}) ➝ c-sinvar m (nodes = N, edges = E ′) ➝ E ′ ⊆ 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
equipped for $G$.

Get all possible offending flows from all security requirements

\begin{definition}
\textit{get-offending-flows} :: \texttt{v SecurityInvariant-configured list} \Rightarrow \texttt{v graph} \Rightarrow (\texttt{v x v} \text{ set set})
\end{definition}

\begin{align*}
\text{get-offending-flows} M G &= (\bigcup m \in \text{set } M. \ c\text{-offending-flows } m G)
\end{align*}

\begin{definition}
\textit{all-security-requirements-fulfilled} :: \texttt{(v::vertex) SecurityInvariant-configured list} \Rightarrow \texttt{v graph} \Rightarrow \texttt{bool}
\end{definition}

\begin{align*}
\text{all-security-requirements-fulfilled} M G &\equiv \forall m \in \text{set } M. (c\text{-sinvar } m) G
\end{align*}

Generate a valid topology from the security requirements

\begin{definition}
\textit{generate-valid-topology} :: \texttt{v SecurityInvariant-configured list} \Rightarrow \texttt{v graph} \Rightarrow \texttt{v graph}
\end{definition}

\begin{align*}
\text{generate-valid-topology} &\quad \circ \quad G = G \\
\text{generate-valid-topology} &\quad (m \# M s G) = \text{delete-edges } (\text{generate-valid-topology } M s G) (\bigcup (c\text{-offending-flows } m G))
\end{align*}

\begin{lemma}
\textit{get-ACS-union-get-IFS}: set (get-ACS M) \cup set (get-IFS M) = set M
\end{lemma}
lemma generate-valid-topology-def-alt:
generate-valid-topology M G = delete-edges G (⋃ (get-offending-flows M G))
(proof)

lemma wf-graph-generate-valid-topology:
wf-graph G ⇒ wf-graph (generate-valid-topology M G)
(proof)

lemma generate-valid-topology-mono-models:
edges (generate-valid-topology (m#M) (⋃ get-offending-flows m G)) ⊆ edges (generate-valid-topology M (⋃ nodes = V, edges = E))
(proof)

generate-valid-topology generates a valid topology (Policy)!

theorem generate-valid-topology-sound:
[ [ valid-reqs M ; wf-graph (⋃ nodes = V, edges = E) ] ] ⇒ 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 (⋃ m ∈ set M. (⋃ (c-offending-flows m G)))
(proof)

lemma c-offending-flows-subseteq-edges:
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: \( \forall m \in \text{set } M. \exists P. \forall G. \ c\text{-sinvar } m \ G = (\forall e \in \text{edges } G. \ P e) \)
shows \( \forall x \in (\bigcup m \in \text{set } M. \bigcup c\text{-offending-flows } m \ (\text{fully-connected } G)) \).
\( \neg \) all-security-requirements-fulfilled \( M \) \( \bigl( \bigl| \text{nodes} = V, \text{edges} = \text{insert } x E \bigr) \)

(proof)

theorem generate-valid-topology-max-topo: \[
\text{valid-reqs } M; \text{wf-graph } G; \\
\forall m \in \text{set } M. \exists P. \forall G. \ c\text{-sinvar } m \ G = (\forall e \in \text{edges } G. \ P e) \implies \\
\text{max-topo } M \ (\text{generate-valid-topology } M \ (\text{fully-connected } G))
\]
(proof)

lemma enf-all-valid-policy-subset-of-max:
assumes validRs: valid-reqs M
and wfG: wf-graph G
and enf: \( \forall m \in \text{set } M. \exists P. \forall G. \ c\text{-sinvar } m \ G = (\forall e \in \text{edges } G. \ P e) \)
and nodesG': nodes G = nodes G'
shows \( \text{wf-graph } G'; \\
\) all-security-requirements-fulfilled \( M \ G' \implies \\
\) edges \( G' \subseteq \text{edges } (\text{generate-valid-topology } M \ (\text{fully-connected } G)) \)
(proof)

7.5 More Lemmata

lemma (in configured-SecurityInvariant) c-sinvar-valid-imp-no-offending-flows:
\( c\text{-sinvar } m \ G \implies c\text{-offending-flows } m \ G = \{\} \)
(proof)

lemma all-security-requirements-fulfilled-imp-no-offending-flows:
\( \text{valid-reqs } M \implies \text{all-security-requirements-fulfilled } M \ G \implies (\bigcup m \in \text{set } M. \bigcup c\text{-offending-flows } m \ G) = \{\} \)
(proof)

corollary all-security-requirements-fulfilled-imp-get-offending-empty:
\( \text{valid-reqs } M \implies \text{all-security-requirements-fulfilled } M \ G \implies \text{get-offending-flows } M \ G = \{\} \)
(proof)

corollary generate-valid-topology-does-nothing-if-valid:
\( \left[ \text{valid-reqs } M; \text{all-security-requirements-fulfilled } M \ G \right] \implies \text{generate-valid-topology } M \ G = G \)
(proof)

lemma mono-extend-get-offending-flows: \[
\left[ \text{valid-reqs } M; \text{wf-graph } \{\text{nodes} = V, \text{edges} = E'\}; \text{E}' \subseteq E; \text{F}' \in \text{get-offending-flows } M \ \{\text{nodes} = V, \text{edges} = E'\} \right] \implies \\
\exists F \in \text{get-offending-flows } M \ \{\text{nodes} = V, \text{edges} = E'\}. \text{F}' \subseteq F
\]
(proof)

lemma get-offending-flows-subseteq-edges: valid-reqs M \implies F \in \text{get-offending-flows } M \ \{\text{nodes} = V, \text{edges} = E\} \implies F \subseteq E
(proof)

**thm** configured-SecurityInvariant. offending-flows-union-mono

**lemma** get-offending-flows-union-mono: \[ \begin{array}{c}
\text{valid-reqs } M; \\
\text{wf-graph } \{ \text{nodes } = V, \text{edges } = E \}; E' \subseteq E \\
\bigcup \text{get-offending-flows } M \{ \text{nodes } = V, \text{edges } = E' \} \subseteq \bigcup \text{get-offending-flows } M \{ \text{nodes } = V, \text{edges } = E \}
\end{array} \] (proof)

**thm** configured-SecurityInvariant. Un-set-offending-flows-bound-minus-subseteq

**lemma** Un-set-offending-flows-bound-minus-subseteq': \[ \begin{array}{c}
\text{valid-reqs } M; \\
\text{wf-graph } \{ \text{nodes } = V, \text{edges } = E \}; E' \subseteq E; \\
\bigcup \text{get-offending-flows } M \{ \text{nodes } = V, \text{edges } = E' \} \subseteq X \\
\bigcup \text{get-offending-flows } M \{ \text{nodes } = V, \text{edges } = E - E' \} \subseteq X - E'
\end{array} \] (proof)

**lemma** ENF-uniquely-defined-offending: valid-reqs M \implies \text{wf-graph } G \implies \\
\forall m \in \text{set } M. \exists P. \forall G. \ c\text{-sinvar } m \ G = (\forall e \in \text{edges } G. \ P \ e) \implies \\
\forall m \in \text{set } M. \forall G. \neg c\text{-sinvar } m \ G \implies (\exists OFF. \ c\text{-offending-flows } m \ G = \{OFF\})
(\text{proof})

**lemma** assumes configured-SecurityInvariant m \\
and \forall G. \neg c\text{-sinvar } m \ G \implies (\exists OFF. \ c\text{-offending-flows } m \ G = \{OFF\}) \\
shows \exists OFF-P. \forall G. \ c\text{-offending-flows } m \ G = (\text{if } c\text{-sinvar } m \ G \text{ then } \{\} \text{ else } \{OFF-P \ G\})
(\text{proof})

Hilber's eps operator example

**lemma** (SOME x. x : \{1::nat, 2, 3\}) = x \implies x = 1 \lor x = 2 \lor x = 3 
(\text{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 
\begin{align*}
generate-valid-topology-SOME [] G & = G \\
generate-valid-topology-SOME (m#Ms) G & = (\text{if } c\text{-sinvar } m \ G \text{ then } \text{generate-valid-topology-SOME } Ms G \\
& \text{ else delete-edges } (\text{generate-valid-topology-SOME } Ms G) \text{ (SOME } F. \ F \in \text{c-offending-flows } m \ G) \}
\end{align*}

**lemma** generate-valid-topology-SOME-nodes: nodes (generate-valid-topology-SOME M (\{nodes = V, \text{edges } = E\})) = V 
(\text{proof})

**theorem** generate-valid-topology-SOME-sound: 
\[ \begin{array}{c}
\text{valid-reqs } M; \text{wf-graph } \{ \text{nodes } = V, \text{edges } = E \} \end{array} \implies \\
\text{all-security-requirements-fulfilled } M \{ \text{generate-valid-topology-SOME } M (\{nodes = V, \text{edges } = E\}) \}
(\text{proof})

**lemma** generate-valid-topology-SOME-def-alt:
generate-valid-topology-SOME $M, G = \text{delete-edges } G (\bigcup m \in M. \text{if } c\text{-sinvar } m \text{ then } \{\} \text{ else } (\text{SOME } F. \ F \in c\text{-offending-flows } m \ G))$

(proof)

lemma generate-valid-topology-SOME-superset:

\[ \text{valid-reqs } M; \text{wf-graph } G \implies \text{edges } (\text{generate-valid-topology } M \ G) \subseteq \text{edges } (\text{generate-valid-topology-SOME } M \ G) \]

(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 \Rightarrow ('v \times 'v) list \Rightarrow 'v graph \Rightarrow '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:

\[ \text{valid-reqs } M; \text{wf-graph } ([\text{nodes } = V, \text{edges } = E]; \text{set } E = E; \text{distinct } E) \implies \text{all-security-requirements-fulfilled } M (\text{generate-valid-topology-some } M \ Es ([\text{nodes } = V, \text{edges } = E])) \]

(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 \rightarrow 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

\[ G = (V, E_{fix}, E_{state}) \]
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.

\[
\text{definition all-flows :: } v \text{ stateful-policy } \Rightarrow (v \times v) \set \text{ where } all-flows \ T \equiv \text{flows-fix} \ T \cup \text{flows-state} \ T \cup \text{backflows (flows-state} \ T)\]

\[
\text{definition stateful-policy-to-network-graph :: } v \text{ stateful-policy } \Rightarrow v \text{ graph } \text{ where } \text{stateful-policy-to-network-graph} \ T = (| \text{nodes} = \text{hosts} \ T, \text{edges} = \text{all-flows} \ T |)\]

\( v \) stateful-policy syntactically well-formed

locale wf-stateful-policy =

\[\begin{align*}
\text{fixes } T &:: v \text{ stateful-policy} \\
\text{assumes } E-wf: \text{fst } (\text{flows-fix} \ T) \subseteq (\text{hosts} \ T) \\
\text{snd } (\text{flows-fix} \ T) \subseteq (\text{hosts} \ T) \\
\text{and } E-state-fix: \text{flows-state} \ T \subseteq \text{flows-fix} \ T \\
\text{and } E-state-valid: \text{finite-Hosts: finite (hosts} \ T) \\
\end{align*}\]

begin

\[
\text{lemma E-wfD: assumes } (v,v') \in \text{flows-fix} \ T \\
\text{shows } v \in \text{hosts} \ T \ v' \in \text{hosts} \ T \\
\langle \text{proof} \rangle
\]

\[
\text{lemma E-state-valid: fst } (\text{flows-state} \ T) \subseteq (\text{hosts} \ T) \\
\text{snd } (\text{flows-state} \ T) \subseteq (\text{hosts} \ T) \\
\langle \text{proof} \rangle
\]

\[
\text{lemma E-state-validD: assumes } (v,v') \in \text{flows-state} \ T \\
\text{shows } v \in \text{hosts} \ T \ v' \in \text{hosts} \ T \\
\langle \text{proof} \rangle
\]

\[
\text{lemma finite-fix: finite (flows-fix} \ T) \\
\langle \text{proof} \rangle
\]

\[
\text{lemma finite-state: finite (flows-state} \ T) \\
\langle \text{proof} \rangle
\]

\[
\text{lemma finite-backflows-state: finite (backflows (flows-state} \ T)) \\
\langle \text{proof} \rangle
\]

\[
\text{lemma E-state-backflows-wf: fst } \text{backflows (flows-state} \ T) \subseteq (\text{hosts} \ T) \\
\text{snd } \text{backflows (flows-state} \ T) \subseteq (\text{hosts} \ T) \\
\langle \text{proof} \rangle
\]

end

Minimizing stateful flows such that only newly added backflows remain

\[
\text{definition filternew-flows-state :: } v \text{ stateful-policy } \Rightarrow (v \times v) \set \text{ where }
\]
filternew-flows-state $\mathcal{T} \equiv \{(s, r) \in \text{flows-state } \mathcal{T}. \ (r, s) \notin \text{flows-fix } \mathcal{T}\}$

**lemma** filternew-subseteq-flows-state: filternew-flows-state $\mathcal{T} \subseteq \text{flows-state } \mathcal{T}$

**lemma** filternew-flows-state-alt: filternew-flows-state $\mathcal{T} = \text{flows-state } \mathcal{T} - \text{(backflows (flows-fix } \mathcal{T}))$

**lemma** filternew-flows-state-alt2: filternew-flows-state $\mathcal{T} = \{e \in \text{flows-state } \mathcal{T}. \ e \notin \text{backflows (flows-fix } \mathcal{T)}\}$

**lemma** backflows-filternew-flows-state: backflows (filternew-flows-state $\mathcal{T}) = \text{(backflows (flows-state } \mathcal{T})) - \text{(flows-fix } \mathcal{T})$

**lemma** backflows-filternew-disjunct-flows-fix:
  $\forall b \in \text{(backflows (filternew-flows-state } \mathcal{T)}). b \notin \text{flows-fix } \mathcal{T}$

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 $G \implies \text{wf-stateful-policy } \mathcal{T} \implies \text{stateful-policy-to-network-graph } \mathcal{T} = \text{(stateful-policy-to-network-graph (hosts = hosts } \mathcal{T}, \text{flows-fix = flows-fix } \mathcal{T}, \text{flows-state = filternew-flows-state } \mathcal{T})\}$

**lemma** backflows-filternew-disjunct-flows-fix:
  $\forall b \in \text{(backflows (filternew-flows-state } \mathcal{T}). b \notin \text{flows-fix } \mathcal{T}$

When is a stateful policy $\mathcal{T}$ compliant with a high-level policy $G$ and the security requirements $M$?

**locale** stateful-policy-compliance =
fi$xes \mathcal{T} :: (\'v::vertex) \text{stateful-policy}$

fi$xes G :: \'v graph$

fi$xes M :: (\'v) \text{SecurityInvariant-configured list}$

assumes
  — the graph must be syntactically valid
  $\text{wfG: wf-graph } G$

and
  — security requirements must be valid
validReqs: $\text{valid-reqs } M$

and

— the high-level policy must be valid

$\text{high-level-policy-valid: all-security-requirements-fulfilled } M \ G$

and

— the stateful policy must be syntactically valid

$\text{stateful-policy-wf: wf-stateful-policy } T$

and

— the stateful policy must talk about the same nodes as the high-level policy

$\text{hosts-nodes:}$

$\text{hosts } T = \text{nodes } G$

and

— only flows that are allowed in the high-level policy are allowed in the stateful policy

$\text{flows-edges:}$

$\text{flows-fix } T \subseteq \text{edges } G$

and

— the low level policy must comply with the high-level policy

— all information flow strategy requirements must be fulfilled, i.e. no leaks!

$\text{compliant-stateful-IFS:}$

$\text{all-security-requirements-fulfilled } (\text{get-IFS } M) (\text{stateful-policy-to-network-graph } T)$

and

— No Access Control side effects must occur

$\text{compliant-stateful-ACS:}$

$\forall F \in \text{get-offending-flows } (\text{get-ACS } M) (\text{stateful-policy-to-network-graph } T). F \subseteq \text{backflows } (\text{filternew-flows-state } T)$

\begin{verbatim}
begin

lemma \text{compliant-stateful-ACS-no-side-effects-filternew-helper:}

$\forall E \subseteq \text{backflows } (\text{filternew-flows-state } T). \forall F \in \text{get-offending-flows } (\text{get-ACS } M) \ (\parallel \text{nodes } = \text{hosts } T, \text{edges } = \text{flows-fix } T \cup \text{E} \parallel). F \subseteq E$

\end{verbatim}

\begin{verbatim}
theorem \text{compliant-stateful-ACS-no-side-effects:}

$\forall E \subseteq \text{backflows } (\text{flows-state } T). \forall F \in \text{get-offending-flows } (\text{get-ACS } M) \ (\parallel \text{nodes } = \text{hosts } T, \text{edges } = \text{flows-fix } T \cup \text{flows-state } T \cup \text{E} \parallel). F \subseteq E$

\end{verbatim}

\begin{verbatim}
corollary \text{compliant-stateful-ACS-no-side-effects':}$\forall E \subseteq \text{backflows } (\text{flows-state } T). \forall F \in \text{get-offending-flows } (\text{get-ACS } M) \ (\parallel \text{nodes } = \text{hosts } T, \text{edges } = \text{flows-fix } T \cup \text{flows-state } T \cup \text{E} \parallel). F \subseteq E$

\end{verbatim}

The high level graph generated from the low level policy is a valid graph

\begin{verbatim}
lemma \text{valid-stateful-policy: } wf-graph \ (\parallel \text{nodes } = \text{hosts } T, \text{edges } = \text{all-flows } T)\parallel$

\end{verbatim}

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

\begin{verbatim}
lemma \text{compliant-stateful-ACS-static-valid: all-security-requirements-fulfilled } (\text{get-ACS } M) \ (\parallel \text{nodes } = \text{hosts } T, \text{edges } = \text{flows-fix } T \parallel)$

\end{verbatim}

101
**theorem** compliant-stateful-ACS-static-valid:
all-security-requirements-fulfilled \( M \mid \text{nodes} = \text{hosts} \ T, \text{edges} = \text{flows-fix} \ T \cup \text{flows-state} \ T \).  
(proof)

The flows with state are a subset of the flows allowed by the policy

**theorem** flows-state-edges:
flows-state \( T \subseteq \text{edges} \ G \)  
(proof)

All offending flows are subsets of the reverse stateful flows

**lemma** compliant-stateful-ACS-only-state-violations:
\( \forall F \in \text{get-offending-flows} (\text{get-ACS} M) (\text{stateful-policy-to-network-graph} \ T). \ F \subseteq \text{backflows} (\text{flows-state} \ T) \)  
(proof)

**theorem** compliant-stateful-ACS-only-state-violations:
\( \forall F \in \text{get-offending-flows} M (\text{stateful-policy-to-network-graph} \ T). \ F \subseteq \text{backflows} (\text{flows-state} \ T) \)  
(proof)

All violations are backflows of valid flows

**corollary** compliant-stateful-ACS-only-state-violations-union:
\( \bigcup \text{get-offending-flows} (\text{get-ACS} M) (\text{stateful-policy-to-network-graph} \ T) \subseteq \text{backflows} (\text{flows-state} \ T) \)  
(proof)

**corollary** compliant-stateful-ACS-only-state-violations-union:
\( \bigcup \text{get-offending-flows} M (\text{stateful-policy-to-network-graph} \ T) \subseteq \text{backflows} (\text{flows-state} \ T) \)  
(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:
\( \forall (v_1, v_2) \in \text{backflows} (\text{flows-state} \ T). \ (\bigcup \text{get-offending-flows} (\text{get-ACS} M) \mid \text{nodes} = \text{hosts} \ T, \text{edges} = \text{flows-fix} \ T \cup \text{flows-state} \ T \cup \{(v_1, v_2)\}) \subseteq \{(v_1, v_2)\} \)  
(proof)

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

\textbf{thm} stateful-policy-compliance.compliant-stateful-ACS-only-state-violations-union'"
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-affending-flows = (λ G. SecurityInvariant-withOffendingFlows.set-affending-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-

### 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
lemmas get-offending-flows :: \langle\text{implc-get-offending-flows} M G\rangle = fold (\lambda m. accu \cdot \text{implc-offending-flows} m G) M ∅

lemma implc-get-offending-flows-fold:
\text{implc-get-offending-flows} M G = fold (\lambda m. \text{accu} \cdot \text{implc-offending-flows} m G) M ∅

lemma implc-get-offending-flows-Un: \text{set} (\text{implc-get-offending-flows} M G) = (\bigcup m \in \text{set} M. \text{set} (\text{implc-offending-flows} m G))

lemma implc-get-offending-flows-map-concat: (\text{implc-get-offending-flows} M G) = \text{concat} [\text{implc-offending-flows} m G. m \leftarrow M]

theorem implc-get-offending-flows-complies:
assumes a1: \forall (m-impl, m-spec) \in \text{set} M. \text{SecurityInvariant-complies-formal-def} m-impl m-spec
and a2: \text{wf-list-graph} G
shows set (\text{implc-get-offending-flows} (\text{get-impl} M) G) = (\text{get-offending-flows} (\text{get-spec} M) (\text{list-graph-to-graph} G))

\section{Accessors}

\textbf{definition get-IFS :: }'v \text{ SecurityInvariant list} \Rightarrow 'v \text{ SecurityInvariant list where}
\text{get-IFS} M \equiv [m \leftarrow M. \text{implc-isIFS} m]

\textbf{definition get-ACS :: }'v \text{ SecurityInvariant list} \Rightarrow 'v \text{ SecurityInvariant list where}
\text{get-ACS} M \equiv [m \leftarrow M. \neg \text{implc-isIFS} m]

\textbf{lemma get-IFS-get-ACS-complies:}
assumes a: \forall (m-impl, m-spec) \in \text{set} M. \text{SecurityInvariant-complies-formal-def} m-impl m-spec
shows \forall \text{ set} (\text{zip} (\text{get-IFS} (\text{get-impl} M) G) (\text{TopoS-Composition-Theory}.\text{get-IFS} (\text{get-spec} M))))
\text{SecurityInvariant-complies-formal-def} m-impl m-spec
and \forall \text{ set} (\text{zip} (\text{get-ACS} (\text{get-impl} M) G) (\text{TopoS-Composition-Theory}.\text{get-ACS} (\text{get-spec} M))))
\text{SecurityInvariant-complies-formal-def} m-impl m-spec
\text{lemma get-IFS-get-ACS-select-simps:}
assumes a1: \forall (m-impl, m-spec) \in \text{set} M. \text{SecurityInvariant-complies-formal-def} m-impl m-spec
shows \forall \text{ set} (\text{zip} (\text{get-IFS} (\text{get-impl} M) G) (\text{TopoS-Composition-Theory}.\text{get-IFS} (\text{get-spec} M))))
\text{SecurityInvariant-complies-formal-def} m-impl m-spec
and \( \forall (m\text{-impl}, m\text{-spec}) \in \text{set \{zip (get-ACS (get-impl M)) \ (TopoS-Composition-Theory.get-ACS (get-spec M))\}} \).

\[ \text{SecurityInvariant-complies-formal-def m\text{-impl} m\text{-spec} (is} \ \forall (m\text{-impl}, m\text{-spec}) \in \text{set \?zippedACS. SecurityInvariant-complies-formal-def m\text{-impl} m\text{-spec}}). \]

\[ \text{and} (\text{get-impl} (\text{zip (TopoS-Composition-Theory-impl.get-ACS (get-impl M)) (TopoS-Composition-Theory.get-ACS (get-spec M))})) \) = \text{TopoS-Composition-Theory-impl.get-ACS (get-impl M)} \]

\[ \text{and} (\text{get-spec} (\text{zip (TopoS-Composition-Theory-impl.get-ACS (get-impl M)) (TopoS-Composition-Theory.get-ACS (get-spec M))})) \) = \text{TopoS-Composition-Theory.get-ACS (get-spec M)} \]

\[ \langle \text{proof} \rangle \]

\textbf{thm get-IFS-get-ACS-select-simps} \]

\section{9.5 All security requirements fulfilled}

\textbf{definition} all-security-requirements-fulfilled :: \( \forall \) SecurityInvariant list \( \Rightarrow \) \( \forall \) list-graph \( \Rightarrow \) bool \ where all-security-requirements-fulfilled M G \( \equiv \forall m \in \text{set } M . \ (\text{implc-sinvar } m) \ G \)

\[ \text{lemma} \ \text{all-security-requirements-fulfilled-complies:} \]

\[ \forall (m\text{-impl}, m\text{-spec}) \in \text{set } M . \ \text{SecurityInvariant-complies-formal-def m\text{-impl} m\text{-spec;}} \]

\[ \text{wf-list-graph } (G:(\text{'v list-graph}) \ list-graph) \] \( \Rightarrow \) all-security-requirements-fulfilled \ (get-impl M) G \leftarrow \text{TopoS-Composition-Theory.all-security-requirements-fulfilled} \ (get-spec M) \ (\text{list-graph-to-graph } G) \]

\[ \langle \text{proof} \rangle \]

\section{9.6 generate valid topology}

\textbf{value} \( \text{concat} \ [\{1:int,2,3\}, \{4,6,5\}] \)

\textbf{fun} \( \text{generate-valid-topology} :: \forall \) SecurityInvariant list \( \Rightarrow \) \( \forall \) list-graph \( \Rightarrow \) \( (\forall \) list-graph) \ where \( \text{generate-valid-topology} M G \) = \( \text{delete-edges } G \ (\text{concat (imple-get-offending-flows } M G)) \)

\[ \text{lemma} \ \text{generate-valid-topology-complies:} \]

\[ \forall (m\text{-impl}, m\text{-spec}) \in \text{set } M . \ \text{SecurityInvariant-complies-formal-def m\text{-impl} m\text{-spec;}} \]

\[ \text{wf-list-graph } (G:(\text{'v list-graph})) \] \( \Rightarrow \) list-graph-to-graph \ (generate-valid-topology \ (get-impl M) G) = \text{TopoS-Composition-Theory.generate-valid-topology \ (get-spec M) \ (list-graph-to-graph } G) \]

\[ \langle \text{proof} \rangle \]

\section{9.7 generate valid topology}

tuned for invariants where we don’t want to calculate all offending flows

Theoretic foundations: The algorithm \( \text{generate-valid-topology-SOME} \) picks ONE offending flow non-deterministically. This is sound: \[ \forall \text{valid-reqs } ?M ; \ \text{wf-graph } (\{\text{nodes} = ?V , \ \text{edges} = ?E\}) \]

\( \Rightarrow \text{TopoS-Composition-Theory.all-security-requirements-fulfilled } ?M \ (\text{generate-valid-topology-SOME } ?M \ (\{\text{nodes} = ?V , \ \text{edges} = ?E\}) \). However, this non-deterministic choice is hard to implement.

To pick one offending flow deterministically, we have implemented \( \text{TopoS-Interface-impl.minimalize-offending-flows} \). It gives back one offending flow: \[ \text{SecurityInvariant-preliminaries } ?\text{sinvar; } \text{wf-graph } ?G; \ \text{SecurityInvariant-with-offending-flows.minimalize-offending-flows } \ ?\text{sinvar (set } ?ff) \ G \ ?nP; \ \text{set } ?ff \subseteq \text{edges } ?G; \ \text{distinct } ?ff \] \( \Rightarrow \) set \ (\text{SecurityInvariant-withOffendingFlows.minimalize-offending-flows } \ ?\text{sinvar } ?ff \ G \ ?nP) \) \in \text{SecurityInvariant-withOffendingFlows.set-offending-flows } ?\text{sinvar } ?G \ ?nP \)

The good thing about this function is, that it does not need to construct the complete \text{SecurityInvariant-withOffendingFlows.set-offending-flows}. Therefore, it can be used for security invariants which may have an exponential number of offending flows. The corre-
sponding algorithm that uses this function is \textit{generate-valid-topology-some}. It is also sound:

\[
\text{valid-reqs} \; ?M \Rightarrow \text{wf-graph} \; (\text{nodes} = ?V, \text{edges} = ?E); \text{set} \; ?Es = ?E; \text{distinct} \; ?Es \Rightarrow \text{TopoS-Composition-Theory.all-security-requirements-fulfilled} \; ?M \; \text{generate-valid-topology-some} \; ?M \; ?Es \; (\text{nodes} = ?V, \text{edges} = ?E)).
\]

\begin{verbatim}
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-sound (get-impl M) G) (minimalize-offending-overapprox (implc-sinvar m) (edgesL G) [] G)
)

end
\end{verbatim}

\textbf{thm} TopoS-Composition-Theory.generate-valid-topology-some-sound

\textbf{lemma} generate-valid-topology-some-sound:

\[
\forall \; (m\text{-impl}, m\text{-spec}) ∈ \text{set} \; M. \; \text{SecurityInvariant-complies-formal-def} \; m\text{-impl} \; m\text{-spec}; \text{wf-list-graph} \; (G:(\text{'v::vertex list-graph}))) \Rightarrow \\
\text{list-graph-to-graph} \; (\text{generate-valid-topology-some} \; (\text{get-impl} \; M) \; G) = \\
\text{TopoS-Composition-Theory.generate-valid-topology-some} \; (\text{get-spec} \; M) \; (\text{edgesL} \; G) \; (\text{list-graph-to-graph} \; G)
\]

\textit{proof}

end

theory TopoS-Stateful-Policy-Algorithm

imports TopoS-Stateful-Policy TopoS-Composition-Theory

begin

\section{Stateful Policy – Algorithm}

\subsection{Some unimportant lemmata}

\textbf{lemma} False-set: \{(r, s), False\} = {}\; \langle\text{proof}\rangle

\textbf{lemma} valid-reqs-ACS-D: valid-reqs M ⇒ valid-reqs (get-ACS M)\; \langle\text{proof}\rangle

\textbf{lemma} valid-reqs-IFS-D: valid-reqs M ⇒ valid-reqs (get-IFS M)\; \langle\text{proof}\rangle

\textbf{lemma} all-security-requirements-fulfilled-ACS-D: all-security-requirements-fulfilled M G ⇒ all-security-requirements-fulfilled (get-ACS M) G\; \langle\text{proof}\rangle

\textbf{lemma} all-security-requirements-fulfilled-IFS-D: all-security-requirements-fulfilled M G ⇒ all-security-requirements-fulfilled (get-IFS M) G\; \langle\text{proof}\rangle

\textbf{lemma} all-security-requirements-fulfilled-mono-stateful-policy-to-network-graph:

\[
\text{valid-reqs} \; M; \; E' ⊆ E; \text{wf-graph} \; (\text{nodes} = V, \text{edges} = Efix \cup E \} \] \Rightarrow \text{all-security-requirements-fulfilled} \; M (\text{stateful-policy-to-network-graph} \; (\text{hosts} = V, \text{flows-fix} = Efix, \text{flows-state} = E \}) \Rightarrow \text{all-security-requirements-fulfilled} \; M (\text{stateful-policy-to-network-graph} \; (\text{hosts} = V, \text{flows-fix} = Efix, \text{flows-state} = E' \})
\]

\textit{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:**

\[
\begin{align*}
\text{wf-graph} (\emptyset \models \text{nodes} = V, \text{edges} = E) &; \text{valid-reqs } M; \text{all-security-requirements-fulfilled } M (\emptyset \models \text{nodes} = V, \text{edges} = E) \iff \\
\text{stateful-policy-compliance} (\emptyset \models \text{hosts} = V, \text{flows-fix} = E, \text{flows-state} = \emptyset) &; (\emptyset \models \text{nodes} = V, \text{edges} = E) \models M
\end{align*}
\]

(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**

where

\[
\begin{align*}
\text{filter-IFS-no-violations-accu } G M \text{ accu } []) = \text{accu } | \\
\text{filter-IFS-no-violations-accu } G M \text{ accu } (e\#Es) = (if \\
\text{all-security-requirements-fulfilled } (\text{get-IFS } M) (\text{stateful-policy-to-network-graph} (\emptyset \models \text{nodes} = G, \text{flows-fix} = \text{edges } G, \text{flows-state} = \text{set } (e\#\text{accu }))) \\
\text{then filter-IFS-no-violations-accu } G M (e\#\text{accu } \text{Es}) \\
\text{else filter-IFS-no-violations-accu } G M \text{ accu } \text{Es})
\end{align*}
\]

**definition filter-IFS-no-violations :: 'v::vertex graph ⇒ 'v SecurityInvariant-configured list ⇒ ('v × 'v) list**

where

\[
\begin{align*}
\text{filter-IFS-no-violations } G M \text{ Es} = \text{filter-IFS-no-violations-accu } G M [] \text{Es }
\end{align*}
\]

**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) ⇒ wf-graph (\emptyset \models \text{nodes} = V, \text{edges} = E) ⇒

\[
\begin{align*}
\text{all-security-requirements-fulfilled } (\text{get-IFS } M) (\text{stateful-policy-to-network-graph} (\emptyset \models \text{nodes} = V, \text{flows-fix} = E, \text{flows-state} = \text{set } (\text{accu }))) \Rightarrow \\
(\text{set } \text{accu} \cup (\text{set } \text{edgesList} \subseteq E \Rightarrow \\
\text{all-security-requirements-fulfilled } (\text{get-IFS } M) (\text{stateful-policy-to-network-graph} (\emptyset \models \text{nodes} = V, \text{flows-fix} = \text{edges } G, \text{flows-state} = \text{set } (\text{filter-IFS-no-violations-accu } (\emptyset \models \text{nodes} = V, \text{edges} = E) \models M \text{accu } \text{edgesList}))) \})
\end{align*}
\]

(proof)

**lemma filter-IFS-no-violations-correct:** valid-reqs (get-IFS M); \text{wf-graph } G;

\[
\begin{align*}
(\text{set } \text{edgesList} \subseteq \text{edges } G) \Rightarrow \\
\text{all-security-requirements-fulfilled } (\text{get-IFS } M) (\text{stateful-policy-to-network-graph} (\emptyset \models \text{nodes } G, \text{flows-fix} = \text{edges } G, \text{flows-state} = \text{set } (\text{filter-IFS-no-violations } G M \text{edgesList}))) \})
\end{align*}
\]

(proof)

**lemma filter-IFS-no-violations-accu-no-IFS:** valid-reqs (get-IFS M) ⇒ \text{wf-graph } G ⇒ \text{get-IFS } M = [] \Rightarrow

\[
\begin{align*}
(\text{set } \text{accu} \cup (\text{set } \text{edgesList} \subseteq \text{edges } G \Rightarrow \\
\text{filter-IFS-no-violations-accu } G M \text{accu } \text{edgesList} = \text{rev } (\text{edgesList}) \text{@accu })
\end{align*}
\]

(proof)

**lemma filter-IFS-no-violations-accu-maximal-induction:** valid-reqs (get-IFS M) ⇒ \text{wf-graph } (\emptyset \models \text{nodes} = V, \text{edges} = E) \Rightarrow

\[
\begin{align*}
\text{set } \text{accu} \subseteq E \Rightarrow \text{set } \text{edgesList} \subseteq E \Rightarrow \\
\forall e \in E - (\text{set } \text{accu} \cup \text{set } \text{edgesList}).
\end{align*}
\]

108
\[ \neg \text{all-security-requirements-fulfilled} (\text{get-IFS } M) (\text{stateful-policy-to-network-graph} \not\bowtie \text{hosts} = V,\ \text{flows-fix} = E,\ \text{flows-state} = \{e\} \cup (\text{set accu} [])) \]

\[ \implies \]

let stateful = set (\text{filter-IFS-no-violations-accu} [] \text{nodes} = V,\ \text{edges} = E [] M \text{accu edgesList})

in

(\forall e \in E - \text{stateful}.
\neg \text{all-security-requirements-fulfilled} (\text{get-IFS } M) (\text{stateful-policy-to-network-graph} \not\bowtie \text{hosts} = V,\ \text{flows-fix} = E,\ \text{flows-state} = \{e\} \cup \text{stateful} []))

\hfill \langle \text{proof} \rangle

\text{lemma filter-IFS-no-violations-maximal:} \valid-reqs (\text{get-IFS } M); \\text{wf-graph } G;

(\text{set edgesList}) = \text{edges } G \implies \text{let stateful} = \text{set (filter-IFS-no-violations } G M \text{edgesList}) \text{in}

\forall e \in \text{edges } G - \text{stateful}.
\neg \text{all-security-requirements-fulfilled} (\text{get-IFS } M) (\text{stateful-policy-to-network-graph} \not\bowtie \text{hosts} = \text{nodes } G,\ \text{flows-fix} = \text{edges } G,\ \text{flows-state} = \{e\} \cup \text{stateful} [])

\hfill \langle \text{proof} \rangle

\text{corollary filter-IFS-no-violations-maximal-allsubsets:}

\text{assumes a1: valid-reqs (get-IFS } M)

\text{and a2: wf-graph } G

\text{and a4: (set edgesList}) = \text{edges } G

\text{shows let stateful} = \text{set (filter-IFS-no-violations } G M \text{edgesList}) \text{in}

\forall E \subseteq \text{edges } G - \text{stateful}.\ E \not\in \{\} \implies
\neg \text{all-security-requirements-fulfilled} (\text{get-IFS } M) (\text{stateful-policy-to-network-graph} \not\bowtie \text{hosts} = \text{nodes } G,\ \text{flows-fix} = \text{edges } G,\ \text{flows-state} = E \cup \text{stateful} [])

\hfill \langle \text{proof} \rangle

\text{thm filter-IFS-no-violations-correct filter-IFS-no-violations-maximal}

\text{Next}

\text{fun filter-compliant-stateful-ACS-accu} :: \text{'v::vertex graph} \Rightarrow \text{'v SecurityInvariant-configured list} \Rightarrow

(\text{'v} \times \text{'v}) \text{list} \Rightarrow (\text{'v} \times \text{'v}) \text{list} \Rightarrow (\text{'v} \times \text{'v}) \text{list where}

\text{filter-compliant-stateful-ACS-accu } G M \text{ accu } [] = \text{accu } |

\text{filter-compliant-stateful-ACS-accu } G M \text{ accu } (e\#Es) = (if

\text{e} \notin \text{backflows } (\text{edges } G) \land (\forall F \in \text{get-offending-flows } (\text{get-ACS } M) (\text{stateful-policy-to-network-graph} \not\bowtie \text{hosts} = \text{nodes } G,\ \text{flows-fix} = \text{edges } G,\ \text{flows-state} = \text{set } (e\#\text{accu} [])).\ F \subseteq \text{backflows } (\text{set } (e\#\text{accu})))

\text{then filter-compliant-stateful-ACS-accu } G M (e\#\text{accu} )\ Es

\text{else filter-compliant-stateful-ACS-accu } G M \text{ accu } Es)

\text{definition filter-compliant-stateful-ACS} :: \text{'v::vertex graph} \Rightarrow \text{'v SecurityInvariant-configured list}

\Rightarrow (\text{'v} \times \text{'v}) \text{list} \Rightarrow (\text{'v} \times \text{'v}) \text{list} \text{ where}

\text{filter-compliant-stateful-ACS } G M \text{ Es} = \text{filter-compliant-stateful-ACS-accu } G M [] \text{Es}

\text{lemma filter-compliant-stateful-ACS-subseteq-input:} \text{set } (\text{filter-compliant-stateful-ACS } G M \text{ Es}) \subseteq \text{set } Es

\hfill \langle \text{proof} \rangle

\text{lemma filter-compliant-stateful-ACS-accu-correct-induction:} \valid-reqs (\text{get-ACS } M) \Rightarrow \text{wf-graph}

(\text{nodes} = V,\ \text{edges} = E [] \Rightarrow

(\text{set accu} ) \cup (\text{set edgesList}) \subseteq E \Rightarrow

\forall F \in \text{get-offending-flows } (\text{get-ACS } M) (\text{stateful-policy-to-network-graph} \not\bowtie \text{hosts} = V,\ \text{flows-fix} = E,\ \text{flows-state} = \text{set } (\text{accu} [])).\ F \subseteq \text{backflows } (\text{set accu} ) \Rightarrow

(\forall a \in \text{set accu}.\ a \notin (\text{backflows } E)) \Rightarrow

T = (\not\bowtie \text{hosts} = V,\ \text{flows-fix} = E,\ \text{flows-state} = \text{set } (\text{filter-compliant-stateful-ACS-accu} \not\bowtie \text{nodes} = V,\ \text{edges} = E [] M \text{accu edgesList} []).\ F \subseteq \text{backflows } (\text{filternew-flows-state } T)
lemma filter-compliant-stateful-ACS-accu-no-side-effects: valid-reqs (get-ACS M) \implies\ wf-graph G \\
\quad \quad \quad \implies \quad \forall F \in get-offending-flows (get-ACS M) \forall (\text{nodes} = \text{nodes } G, \text{edges} = \text{edges } G \cup \text{backflows } (\text{edges } G)). F \subseteq (\text{backflows } (\text{edges } G)) - (\text{edges } G) \implies \\
\quad \quad \quad \quad (\forall a \in \text{set accu}, a \notin (\text{backflows } (\text{edges } G))) \implies \quad \\
\quad \quad \quad \quad \text{filter-compliant-stateful-ACS-accu } G M \text{ accu } \text{edgesList } = \text{rev}([ e \leftarrow \text{edgesList}. e \notin \text{backflows } (\text{edges } G)])^@\text{accu} \\
(proof)
lemma filter-compliant-stateful-ACS-maximal: \[\text{valid-reqs (get-ACS M); wf-graph (} \subseteq \text{ nodes } = V, \text{ edges } = E \}]\;
(set edgesList) = E;
stateful = set (filter-compliant-stateful-ACS (} \subseteq \text{ nodes } = V, \text{ edges } = E \}] \ M \text{ edgesList})
\Rightarrow 
\forall e \in E \in (\text{stateful } \cup \{ e \in E. e \in \text{ backflows } E ) \}). 
\\equiv \neg (\bigcup \text{ get-offending-flows (get-ACS M) (stateful-policy-to-network-graph (} \subseteq \text{ hosts } = V, \text{ flows-fix } = E, \text{ flows-state } = \text{ stateful } \cup \{ e \} ))
\subseteq \text{ backflows (filternew-flows-state (} \subseteq \text{ hosts } = V, \text{ flows-fix } = E, \text{ flows-state } = \text{ stateful } \cup \{ e \} )))
\langle \text{proof} \rangle

lemma filter-compliant-stateful-ACS-maximal-allsubsets:
\begin{align*}
\text{assumes } & a1: \text{valid-reqs (get-ACS M)} \text{ and } a2: \text{wf-graph (} \subseteq \text{ nodes } = V, \text{ edges } = E \}\]
\text{and } a3: (set edgesList) = E
\text{and } a4: \text{stateful } = \text{ set (filter-compliant-stateful-ACS (} \subseteq \text{ nodes } = V, \text{ edges } = E \}] \ M \text{ edgesList})
\text{and } a5: X \subseteq E \in (\text{stateful } \cup \text{ backflows } E ) \text{ and } a6: X \neq \}
\text{shows } 
\neg (\bigcup \text{ get-offending-flows (get-ACS M) (stateful-policy-to-network-graph (} \subseteq \text{ hosts } = V, \text{ flows-fix } = E, \text{ flows-state } = \text{ stateful } \cup X \})
\subseteq \text{ backflows (filternew-flows-state (} \subseteq \text{ hosts } = V, \text{ flows-fix } = E, \text{ flows-state } = \text{ stateful } \cup X \})
\langle \text{proof} \rangle

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 \Rightarrow 'v SecurityInvariant-configured list \Rightarrow ('v \times 'v) list \Rightarrow '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 (} \subseteq \text{ hosts } = \text{ nodes } G, \text{ flows-fix } = \text{ edges } G, \text{ flows-state } = \text{ set filterACS } [)))

lemma generate-valid-stateful-policy-IFSACS-wf-stateful-policy: \text{assumes } wfG: \text{wf-graph } G
\text{and } edgesList: (set edgesList) = \text{edges } G
\text{shows } \text{wf-stateful-policy (generate-valid-stateful-policy-IFSACS G M edgesList)}
\langle \text{proof} \rangle

lemma generate-valid-stateful-policy-IFSACS-select-simps:
\text{shows } \text{hosts (generate-valid-stateful-policy-IFSACS G M edgesList) = nodes } G
\text{and } \text{flows-fix (generate-valid-stateful-policy-IFSACS G M edgesList) = edges } G
\text{and } \text{flows-state (generate-valid-stateful-policy-IFSACS G M edgesList) \subseteq set edgesList}
\langle \text{proof} \rangle

lemma generate-valid-stateful-policy-IFSACS-all-security-requirements-fulfilled-IFS: \text{assumes } \text{validReqs: valid-reqs } M
\text{and } wfG: \text{wf-graph } G
\text{and } \text{high-level-policy-valid: all-security-requirements-fulfilled } M G
\text{and } \text{edgesList: (set edgesList) \subseteq edges } G

111
shows all-security-requirements-fulfilled (get-IFS M) (stateful-policy-to-network-graph (generate-valid-stateful-policy-IFSACS 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: T = generate-valid-stateful-policy-IFSACS G M edgesList
shows stateful-policy-compliance 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-policy-IFSACS-2 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: T = generate-valid-stateful-policy-IFSACS-2 G M edgesList
shows ∀ F∈get-offending-flows (get-ACS M) (stateful-policy-to-network-graph T), F ⊆ backflows (filternew-flows-state T)
⟨proof⟩

112
**Theorem** 

\texttt{generate-valid-stateful-policy-IFSACS-2-stateful-policy-compliance}:

\texttt{assumes} \texttt{validReqs: valid-reqs M}
and \texttt{wfG: wf-graph G}
and \texttt{high-level-policy-valid: all-security-requirements-fulfilled M G}
and \texttt{edgesList: (set edgesList) = edges G}
and \texttt{Tau: \tau = generate-valid-stateful-policy-IFSACS-2 G M edgesList}

\texttt{shows} \texttt{stateful-policy-compliance \tau G M}

\texttt{(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** \texttt{generate-valid-stateful-policy-IFSACS-2-noIFS-noACSsideeffects-imp-fullgraph}:

\texttt{assumes} \texttt{validReqs: valid-reqs M}
and \texttt{wfG: wf-graph G}
and \texttt{high-level-policy-valid: all-security-requirements-fulfilled M G}
and \texttt{edgesList: (set edgesList) = edges G}
and \texttt{no-ACS-sideeffects: \forall F \in \text{get-offending-flows (get-ACS M)} (|nodes = nodes G, edges = edges G \cup \text{backflows (edges G)}), F \subseteq (\text{backflows (edges G)}) - (edges G)}
and \texttt{no-IFS: get-IFS M = []}

\texttt{shows} \texttt{stateful-policy-to-network-graph (generate-valid-stateful-policy-IFSACS-2 G M edgesList) = undirected G}

\texttt{(proof)}

**Lemma** \texttt{generate-valid-stateful-policy-IFSACS-noIFS-noACSsideeffects-imp-fullgraph}:

\texttt{assumes} \texttt{validReqs: valid-reqs M}
and \texttt{wfG: wf-graph G}
and \texttt{high-level-policy-valid: all-security-requirements-fulfilled M G}
and \texttt{edgesList: (set edgesList) = edges G}
and \texttt{no-ACS-sideeffects: \forall F \in \text{get-offending-flows (get-ACS M)} (|nodes = nodes G, edges = edges G \cup \text{backflows (edges G)}), F \subseteq (\text{backflows (edges G)}) - (edges G)}
and \texttt{no-IFS: get-IFS M = []}

\texttt{shows} \texttt{stateful-policy-to-network-graph (generate-valid-stateful-policy-IFSACS G M edgesList) = undirected G}

\texttt{(proof)}

end

**Theory** \texttt{TopoS-Stateful-Policy-impl}

**Imports** \texttt{TopoS-Composition-Theory-impl TopoS-Stateful-Policy-Algorithm}

begin

11 Stateful Policy – List Implementaion

**Record** \texttt{'v stateful-list-policy =}

\texttt{hostsL :: 'v list}

\texttt{flows-fixL :: ('v x 'v) list}

\texttt{flows-stateL :: ('v x 'v) list}

113
**definition** stateful-list-policy-to-list-graph :: 'v stateful-list-policy ⇒ 'v list-graph

stateful-list-policy-to-list-graph T = (nodesL = hostsL T, edgesL = (flows-fixL T) @ [e ← flows-stateL T. e ∉ set (flows-fixL T)] @ [e ← backlinks (flows-stateL T). e ∉ set (flows-fixL T)])

**lemma** stateful-list-policy-to-list-graph-complies:

list-graph-to-graph (stateful-list-policy-to-list-graph (nodesL = 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 (nodesL = 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

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 (nodesL = nodesL G, flows-fixL = edgesL G, flows-stateL = E)) 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

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)) \( \land \) (\( \forall \)) F ∈ set (simple-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:
\[ \forall (m-impl, m-spec) ∈ set M. SecurityInvariant-complies-formal-def m-impl m-spec; \]
wf-list-graph G; set Es ⊆ set (edgesL G); set accu ⊆ set (edgesL G); distinct (Es@accu) \[ \implies \]
filter-compliant-stateful-ACS-accu G (get-impl M) accu Es = TopoS-Stateful-Policy-Algorithm.filter-compliant-stateful-ACS
(list-graph-to-graph G) (get-spec M) accu Es
⟨proof⟩

lemma filter-compliant-stateful-ACS-cont-complies:
\[ \forall (m-impl, m-spec) ∈ set M. SecurityInvariant-complies-formal-def m-impl m-spec; \]
wf-list-graph G; set Es ⊆ set (edgesL G); distinct Es \[ \implies \]
filter-compliant-stateful-ACS-accu G (get-impl M) [] Es = TopoS-Stateful-Policy-Algorithm.filter-compliant-stateful-ACS
(list-graph-to-graph G) (get-spec M) Es
⟨proof⟩

lemma filter-compliant-stateful-ACS-complies:
\[ \forall (m-impl, m-spec) ∈ set M. SecurityInvariant-complies-formal-def m-impl m-spec; \]
wf-list-graph G \[ \implies \]
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 \( \Rightarrow \) \( v \) SecurityInvariant list \( \Rightarrow \) \( v \) stateful-list-policy
where

\[
\text{generate-valid-stateful-policy-IFSACS-2 } G \ M = \\
\left( \text{hostsL} = \text{nodesL } G, \text{flows-fixL} = \text{edgesL } G, \text{flows-stateL} = \text{inefficient-list-intersect} \left( \text{filter-IFS-no-violations} \ G \ M \right) \right)
\]

lemma generate-valid-stateful-policy-IFSACS-2-complies: \( \forall (m-impl, m-spec) \in \text{set } M. \text{SecurityInvariant-complies-formal-def} \ m-impl \ m-spec; \)

\[
wf-list-graph \ G; \\
\text{valid-reqs} \ (\text{get-spec} \ M); \\
\text{TopoS-Composition-Theory.all-security-requirements-fulfilled} \ (\text{get-spec} \ M) \ (\text{list-graph-to-graph} \ G); \\
\mathcal{T} = (\text{generate-valid-stateful-policy-IFSACS-2} \ G \ (\text{get-impl} \ M)) \implies \\
\text{stateful-policy-compliance} \ (\text{hosts} = \text{set} \ (\text{hostsL} \ \mathcal{T}), \text{flows-fix} = \text{set} \ (\text{flows-fixL} \ \mathcal{T}), \text{flows-state} = \text{set} \ (\text{flows-stateL} \ \mathcal{T}) \ \mathcal{T}) \ \text{(list-graph-to-graph} \ G) \ \text{(get-spec} \ M) \langle \text{proof} \rangle
\]

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 \( \Rightarrow \) subnets
where

\[
\text{system-components-to-subnets} \ \text{SystemComponent} = \text{Member} \ |\\
\text{system-components-to-subnets} \ \text{SystemBoundaryInput} = \text{InboundGateway} \ |\\
\text{system-components-to-subnets} \ \text{SystemBoundaryOutput} = \text{Member} \ |\\
\text{system-components-to-subnets} \ \text{SystemBoundaryInputOutput} = \text{InboundGateway}
\]

fun system-components-to-blp :: system-components \( \Rightarrow \) SINVAR-BLPtrusted.node-config
where

\[
\text{system-components-to-blp} \ \text{SystemComponent} = \left( \text{security-level} = 1, \text{trusted} = \text{False} \right) \ |\\
\text{system-components-to-blp} \ \text{SystemBoundaryInput} = \left( \text{security-level} = 1, \text{trusted} = \text{False} \right) \ |\\
\text{system-components-to-blp} \ \text{SystemBoundaryOutput} = \left( \text{security-level} = 0, \text{trusted} = \text{True} \right) \ |\\
\text{system-components-to-blp} \ \text{SystemBoundaryInputOutput} = \left( \text{security-level} = 0, \text{trusted} = \text{True} \right)
\]

definition new-meta-system-boundary :: \( (v::\text{vertex} \times \text{system-components}) \ \text{list} \ \Rightarrow \ \text{string} \ \Rightarrow \ (\text{v SecurityInvariant}) \ \text{list} \)
where

\[
\text{new-meta-system-boundary} \ C \ \text{description} = \left[ \\
\text{new-configured-list-SecurityInvariant} \ \text{SINVAR-LIB-SubnetsInGW} \\
\left( \text{node-properties} = \text{map-of} \ (\lambda (v,c). \ (v, \text{system-components-to-subnets} \ c)) \ C \right) \\
\right] \ (\text{description} \ @ \ "" \ (\text{ACS})"")
\]

116
new-configured-list-SecurityInvariant SINVAR-LIB-BLPtrusted []
  node-properties = map-of (map (\(v,c\). (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) \iff
  c = SystemBoundaryInput \lor c = SystemBoundaryInputOutput
(proof)

lemma system-components-to-blp:
  (\neg trusted SINVAR-BLPtrusted. default-node-properties \implies
  security-level (system-components-to-blp c) \leq security-level SINVAR-BLPtrusted. default-node-properties)
  \iff
  c = SystemBoundaryOutput \lor c = SystemBoundaryInputOutput
(proof)

lemma all-security-requirements-fulfilled (new-meta-system-boundary C description) G \iff
  (\forall (v_1, v_2) \in set (edgesL G). case ((map-of C) v_1, (map-of C) v_2)
  of
    — No restrictions outside of the component
      (None, None) \Rightarrow True
    — no restrictions inside the component
      | (Some c1, Some c2) \Rightarrow True
    — System Boundaries Input
      | (None, Some SystemBoundaryInputOutput) \Rightarrow True
      | (None, Some SystemBoundaryInput) \Rightarrow True
    — System Boundaries Output
      | (Some SystemBoundaryOutput, None) \Rightarrow True
      | (Some SystemBoundaryInputOutput, None) \Rightarrow True
    — everything else is prohibited
      | - \Rightarrow 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 |)
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-an-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))

end

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 security policy is a directed graph.

definition policy :: nat list-graph where
  policy ≡ ( | nodesL = [1,2,3],
It is syntactically well-formed

**lemma** wf-list-graph-policy: wf-list-graph policy \(\langle\) proof \(\rangle\)

In contrast, this is not a syntactically well-formed graph.

**lemma** \(\neg\) wf-list-graph \(\langle\) nodesL = \([1,2]::\text{nat list}\), edgesL = \([(1,2), (2,2), (2,3)]\) \(\rangle\) \(\langle\) proof \(\rangle\)

Our policy has three rules.

**lemma** length (edgesL policy) = 3 \(\langle\) proof \(\rangle\)

### 14.2 Security Invariants

We construct a security invariant. Node 2 has confidential data

**definition** BLP-security-levels :: \(\text{nat} \rightarrow \text{SINVAR-BLPtrusted.node-config}\)

\[
\text{BLP-security-levels} \equiv [2 \mapsto (\text{security-level} = 1, \text{trusted} = \text{False})]
\]

**definition** BLP-m :: (nat SecurityInvariant) where

\[
\begin{align*}
\text{BLP-m} & \equiv \text{new-configured-list-SecurityInvariant SINVAR-LIB-BLPtrusted} () \\
\text{node-properties} & = \text{BLP-security-levels} \\
& \langle "\text{Two has confidential information}" \rangle
\end{align*}
\]

Interlude: BLP-m is a valid implementation of a SecurityInvariant

**definition** BLP-m-spec :: nat SecurityInvariant-configured option where

\[
\begin{align*}
\text{BLP-m-spec} & \equiv \text{new-configured-SecurityInvariant} \ (\text{SINVAR-BLPtrusted.sinvar}, \\
\text{SINVAR-BLPtrusted.default-node-properties,} \\
\text{SINVAR-BLPtrusted.receiver-violation,} \\
\text{SecurityInvariant.node-props SINVAR-BLPtrusted.default-node-properties} () \\
\text{node-properties} & = \text{BLP-security-levels} \\
& \rangle
\end{align*}
\]

Fist, we need to show that the formal definition obeys all requirements, \textit{new-configured-SecurityInvariant} verifies this. To double check, we manually give the configuration.

**lemma** BLP-m-spec: assumes \(nP = (\lambda v. (\text{case BLP-security-levels} v \text{ of Some } c \Rightarrow c | \text{None} \Rightarrow \text{SINVAR-BLPtrusted.default-node-properties}))\)

\[\text{shows} \quad \text{BLP-m-spec} = \text{Some} () \quad \begin{align*}
\text{c-sinvar} & = (\lambda G. \text{SINVAR-BLPtrusted.sinvar} G nP), \\
\text{c-offending-flows} & = (\lambda G. \text{SecurityInvariant-withOffendingFlows.set-offending-flows}
\text{SINVAR-BLPtrusted.sinvar} G nP), \\
\text{c-isIFS} & = \text{SINVAR-BLPtrusted.receiver-violation}
\end{align*} \ (\text{is BLP-m-spec} = \text{Some} ?\text{Spec}) \langle\text{proof}\rangle\]

**lemma** valid-reqs-BLP: valid-reqs \[\text{the BLP-m-spec}\] \langle\text{proof}\rangle

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 \[\text{the BLP-m-spec}\] \langle\text{proof}\rangle
We define the list of all security invariants of type \(\text{nat SecurityInvariant list}\). The type \(\text{nat}\) is because the policy’s nodes are of type \(\text{nat}\).

\[
\text{definition security-invariants} = [\text{BLP-m}]
\]

We can see that the policy does not fulfill the security invariants.

\[
\text{lemma} \; \neg \; \text{all-security-requirements-fulfilled} \; \text{security-invariants} \; \text{policy} \quad \langle \text{proof} \rangle
\]

We ask why. Obviously, node 2 leaks confidential data to node 3.

\[
\text{value} \; \text{implc-get-offending-flows} \; \text{security-invariants} \; \text{policy} = [[[2, 3]]] \quad \langle \text{proof} \rangle
\]

Interlude: the implementation \text{implc-get-offending-flows} corresponds to the formal definition \text{get-offending-flows}

\[
\text{lemma set ' set (implc-get-offending-flows (get-impl [[BLP-m, the BLP-m-spec]]) policy) = get-offending-flows (get-spec [[BLP-m, the BLP-m-spec]]) (list-graph-to-graph policy) \langle \text{proof} \rangle}
\]

Visualization of the violation (only in interactive mode)

\(\langle \text{ML} \rangle\)

Experimental: the config (only one) can be added to the end.

\(\langle \text{ML} \rangle\)

The policy has a flaw. We throw it away and generate a new one which fulfills the invariants.

\[
\text{definition max-policy} = \text{generate-valid-topology} \; \text{security-invariants} \; \langle \text{nodesL = nodesL policy, edgesL = List.product (nodesL policy) (nodesL policy)} \rangle
\]

Interlude: the implementation \text{implc-get-offending-flows} corresponds to the formal definition \text{get-offending-flows}

\[
\text{thm generate-valid-topology-complies}\]

Interlude: the formal definition is sound

\[
\text{thm generate-valid-topology-sound}
\]

Here, it is also complete

\[
\text{lemma wf-graph G \implies max-topo [the BLP-m-spec] (TopoS-Composition-Theory.generate-valid-topology [the BLP-m-spec] (fully-connected G)) \langle \text{proof} \rangle}
\]

Calculating the maximum policy

\[
\text{value max-policy} = \langle \text{nodesL} = [1, 2, 3], \text{edgesL} = [(1, 1), (1, 2), (1, 3), (2, 2), (3, 1), (3, 2), (3, 3)] \rangle \quad \langle \text{proof} \rangle
\]

Visualizing the maximum policy (only in interactive mode)

\(\langle \text{ML} \rangle\)

Of course, all security invariants hold for the maximum policy.

\[
\text{lemma all-security-requirements-fulfilled} \; \text{security-invariants} \; \text{max-policy} \quad \langle \text{proof} \rangle
\]
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::'b) 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
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 information sink"
⟨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]
Connecting (3::'a, 4::'b). This causes only one offending flow at (3::'a, 4::'b).

We try to increase the dependability level at 3::'a. Suddenly, offending flows everywhere.

If we recompute the dependability levels for the changed graph, we see that suddenly, The level at 1::'a 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::'a, she may be able to peek much deeper into the network.
context begin

private definition G-noninter :: nat list-graph where
G-noninter ≡ \(\{\text{nodesL} = [1::\text{nat}, 2, 3, 4], \text{edgesL} = [(1, 2), (1, 3), (2, 3), (3, 4)]\}\)

private lemma wf-list-graph G-noninter (proof)

definition NonI-m ≡ new-configured-list-SecurityInvariant SINVAR-LIB-NonInterference (proof)

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)]\]

(ML)

lemma implc-offending-flows NonI-m \(\{\text{nodesL} = [1::\text{nat}, 2, 3, 4], \text{edgesL} = [(1, 2), (1, 3), (2, 3), (4, 3)]\}\) =
\([(1, 2), (1, 3)], [(1, 3), (2, 3)], [(4, 3)]\)

(ML)

In comparison, SINVAR-LIB-ACLcommunicateWith is less strict. Changing the direction of the edge \(3::'a, 4::'b\) removes the access from \(1::'a\) to \(4::'a\) and the invariant holds.

lemma implc-offending-flows (new-configured-list-SecurityInvariant SINVAR-LIB-ACLcommunicateWith (ML)

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") \(\{\text{nodesL} = [1::\text{nat}, 2, 3, 4], \text{edgesL} = [(1, 2), (1, 3), (2, 3), (4, 3)]\}\) = []

(ML)

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 = subnet-host-attributes
  ] "Collaborating hosts"

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 = ["v1" ↦ Member, "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" ↦ Member ["v1b", "v2b"],
  "v1b" ↦ Care,
  "v2b" ↦ Care
  ]

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 ∈ set (edgesL (make-policy Subnets-m subnet-hosts))] = [("v1b", "v11"), ("v1b", "v12"), ("v1b", "v13"), ("v2b", "v21"), ("v2b", "v22"), ("v2b", "v23")]

127
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 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.

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.


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"),
```
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 to identify an individual employee, hence produces less critical data, hence has a lower level.

Secret corporate knowledge and intellectual property: The production process is a corporate trade secret. The mission control devices have the trade secrets in their program. The important and secret step is done by MissionControl2.
private lemma dom (BLP-tradesecrets-host-attributes) ⊆ set (nodesL policy)

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)

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.

case 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-attributes) ⊆ set (nodesL policy)

definition ACL-bot2-m ≡ new-configured-list-SecurityInvariant SINVAR-LIB-CommunicationPartners
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.

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" ⊆ "ControlDevices" ⊆ "ControlTerminal". Because the watchdog is in a different domain, it needs a trust level of 1 to access the robots it is monitoring.
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 ≡
\[
\begin{align*}
\text{"MissionControl1"} & \mapsto \text{SinkPool}, \\
\text{"MissionControl2"} & \mapsto \text{SinkPool}, \\
\text{"Robot1"} & \mapsto \text{Sink}, \\
\text{"Robot2"} & \mapsto \text{Sink}
\end{align*}
\]
private lemma dom SinkRobots-host-attributes ⊆ set (nodesL policy)
\langle proof \rangle
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 ≡
\[
\begin{align*}
\text{"Statistics"} & \mapsto \text{Subnet 1}, \\
\text{"SensorSink"} & \mapsto \text{Subnet 1}, \\
\text{"PresenceSensor"} & \mapsto \text{Subnet 1}, \\
\text{"Webcam"} & \mapsto \text{Subnet 1}, \\
\text{"TempSensor"} & \mapsto \text{Subnet 1}, \\
\text{"FireSensor"} & \mapsto \text{Subnet 1}, \\
\text{"AdminPc"} & \mapsto \text{Subnet 4}
\end{align*}
\]
private lemma dom Subnets-host-attributes ⊆ set (nodesL policy)
\langle proof \rangle
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 ≡
\[
\{ "Statistics" \mapsto \text{Member}, \\
\text{"SensorSink"} \mapsto \text{InboundGateway} \}
\]
private lemma dom SubnetsInGW-host-attributes ⊆ set (nodesL policy)
\langle proof \rangle
definition SubnetsInGW-m ≡ new-configured-list-SecurityInvariant 
SINVAR-LIB-SubnetsInGW (\{ node-properties = SubnetsInGW-host-attributes \})
"Protecting 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 academic 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 \textit{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 automatically generated one. If we exclude the NonInterference invariant from the policy set, 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.

\begin{verbatim}
value code make-policy invariants (nodesL policy)
lemma make-policy invariants (nodesL policy) =
  \{ nodesL =
      "Robot2", "AdminPc", "INET"],
    edgesL =
        ("SensorSink", "SensorSink"), ("SensorSink", "Webcam"),
        ("Webcam", "SensorSink"), ("Webcam", "Webcam"),
        ("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 \leftarrow edgesL (make-policy invariants (nodesL policy)). e \notin \text{set} (edgesL policy) \} =
  \{ set [v, e) v \leftarrow (nodesL policy)] \cup
    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.

\end{verbatim}
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?

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 than 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)))]
[\[e \leftarrow \text{edgesL}(\text{make-policy invariants (nodesL policy))}. \text{fst } e = "\text{"AdminPc"}" \wedge \text{snd } e \neq "\text{"AdminPc"}"\]

(\text{proof})

(\text{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 NonInterferance 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 \(\Phi\)-structured invariants and afterwards fix the policy manually for the remaining non-\(\Phi\)-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")]] (\text{proof})

(\text{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

\[
[BLP-privacy-m, BLP-employee-export-m, ACL-bot2-m, Control-hierarchy-m, PolEnforcePoint-m, Subnets-m, SubnetsInGW-m] = \]

\[
\{\text{hosts} = \text{nodes} \oplus \text{flows-fix} \oplus \text{flows-state} = \]

\[
[\text{"Webcam"}, \text{"SensorSink"}], \]

\[
[\text{"SensorSink"}, \text{"Statistics"}], \]

\[
[\text{"MissionControl1"}, \text{"Robot1"}], \]

\[
[\text{"MissionControl1"}, \text{"Robot2"}], \]

\[
[\text{"MissionControl2"}, \text{"Robot2"}], \]

\[
[\text{"AdminPc"}, \text{"MissionControl2"}], \]

\[
[\text{"AdminPc"}, \text{"MissionControl1"}], \]

\[
[\text{"Watchdog"}, \text{"Robot1"}], \]

\[
[\text{"Watchdog"}, \text{"Robot2"}]\} \langle \text{proof} \rangle
\]

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 $e \leftarrow \text{edges} \oplus \text{make-policy} [BLP-privacy-m, BLP-employee-export-m, ACL-bot2-m, Control-hierarchy-m, PolEnforcePoint-m, Subnets-m, SubnetsInGW-m]$ (nodes $\oplus$ policy).

\[
e \notin \text{edges} \oplus \text{nodes} \oplus \text{flows-fix} \oplus \text{flows-state} = \]

\[
[\text{"TempSensor"}, \text{"INET"}], \]

\[
[\text{"FireSensor"}, \text{"INET"}], \]

\[
[\text{"MissionControl1"}, \text{"MissionControl2"}], \]

\[
[\text{"Watchdog"}, \text{"MissionControl1"}], \]

\[
[\text{"Watchdog"}, \text{"MissionControl2"}], \]

\[
[\text{"Watchdog"}, \text{"INET"}], \]

\[
[\text{"AdminPc"}, \text{"Watchdog"}], \]

\[
[\text{"AdminPc"}, \text{"Robot1"}], \]

\[
[\text{"AdminPc"}, \text{"INET"}] \oplus \]

\[
[\text{"MissionControl1"}, \text{"INET"}], \]

\[
[\text{"MissionControl2"}, \text{"MissionControl1"}], \]

\[
[\text{"MissionControl2"}, \text{"Robot1"}], \]

\[
[\text{"MissionControl2"}, \text{"INET"}], \]

\[
[\text{"Robot1"}, \text{"INET"}], \]

\[
[\text{"Robot2"}, \text{"Robot1"}], \]

\[
[\text{"Robot2"}, \text{"INET"}]\} \langle \text{proof} \rangle
\]
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 \texttt{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 \texttt{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 \texttt{SinkPool}.

\textbf{definition} \texttt{invariants-tuned} \equiv \texttt{BLP-privacy-m, BLP-employee-export-m, ACL-bot2-m, Control-hierarchy-m, PolEnforcePoint-m, Subnets-m, SubnetsInGW-m, new-configured-list-SecurityInvariant \texttt{SINVAR-LIB-Sink}} \begin{align*} &\left\langle\begin{array}{l} \text{node-properties} = [\text{``MissionControl1''}\mapsto\text{SinkPool}, \\
\text{``MissionControl2''}\mapsto\text{SinkPool}, \\
\text{``Robot1''}\mapsto\text{SinkPool}, \\
\text{``Robot2''}\mapsto\text{SinkPool}, \\
\text{``Watchdog''}\mapsto\text{SinkPool}, \\
\text{``AdminPc''}\mapsto\text{SinkPool} \end{array}\right] \\
\text{``non-leaking production units''}, \\
\text{new-configured-list-SecurityInvariant \texttt{SINVAR-LIB-BLPtrusted}} \left\langle\begin{array}{l} \text{node-properties} = [\text{``MissionControl1''}\mapsto\{\text{security-level = 1, trusted = False }\}, \\
\text{``MissionControl2''}\mapsto\{\text{security-level = 2, trusted = False }\}, \\
\text{``Robot1''}\mapsto\{\text{security-level = 1, trusted = False }\}, \\
\text{``Robot2''}\mapsto\{\text{security-level = 2, trusted = False }\}, \\
\text{``Watchdog''}\mapsto\{\text{security-level = 1, trusted = True }\}, \\
\end{array}\right] \\
\text{``trade secrets''} \end{align*}

\textbf{lemma} \texttt{all-security-requirements-fulfilled invariants-tuned policy \langle proof \rangle}

\textbf{definition} \texttt{stateful-policy-tuned} = \texttt{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))) ⊂ set (edgesL (make-policy invariants (nodesL policy))) ⟨proof⟩

lemma set [e ← edgesL (make-policy invariants-tuned (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"],
["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
