Formal Verification of Secure Forwarding Protocols (Artifact for CSF'21)

Tobias Klenze (tobias.klenze@inf.ethz.ch), Christoph Sprenger (sprenger@inf.ethz.ch)

February 7, 2021

Contents

1 Verification Infrastructure 6

1.1 Event Systems . . . 7
1.1.1 Reachable states and invariants . . . 7
1.1.2 Traces . . . 8
1.1.3 Simulation . . . 11
1.1.4 Simulation up to simulation preorder . . . 14
1.2 Atomic messages . . . 15
1.2.1 Agents . . . 15
1.2.2 Nonces and keys . . . 15
1.3 Symmetric and Asymetric Keys . . . 16
1.3.1 Asymmetric Keys . . . 16
1.3.2 Basic properties of *pubK* and *priK* . . . 16
1.3.3 "Image" equations that hold for injective functions . . . 17
1.3.4 Symmetric Keys . . . 17
1.4 Theory of ASes and Messages for Security Protocols . . . 19
1.4.1 keysFor operator . . . 20
1.4.2 Inductive relation "parts" . . . 21
1.4.3 Inductive relation "analz" . . . 25
1.4.4 Inductive relation "synth" . . . 31
1.4.5 HPair: a combination of Hash and MPair . . . 35
1.5 Tools . . . 39
1.5.1 Prefixes, suffixes, and fragments . . . 39
1.5.2 Fragments . . . 39
1.5.3 Pair Fragments . . . 40
1.5.4 Head and Tails . . . 41
1.6 takeW, holds and extract: Applying context-sensitive checks on list elements 42
1.6.1 Definitions . . . 42
1.6.2 Lemmas . . . 43

2 Abstract, and Concrete Parametrized Models 47

2.1 Network model . . . 48
2.1.1 Interface check . . . 48
2.2 Abstract Model . . . 50
2.2.1 Events . . . 51
2.2.2 Transition system . . . 53
2.2.3 Path authorization property . . . 54

2.2.4 Detectability property . . . 55
2.3 Intermediate Model . . . 57
2.3.1 Events . . . 57
2.3.2 Transition system . . . 58
2.3.3 Auxilliary definitions . . . 60
2.4 Concrete Parametrized Model . . . 61
2.4.1 Hop validation check, authorized segments, and path extraction. . . . 61
2.4.2 Intruder Knowledge definition . . . 64
2.4.3 Events . . . 66
2.4.4 Transition system . . . 67
2.4.5 Assumptions of the parametrized model . . . 68
2.4.6 Mapping dp2 state to dp1 state . . . 68
2.4.7 Invariant: Derivable Intruder Knowledge is constant under *dp2-trans* . 69
2.4.8 Refinement proof . . . 71
2.4.9 Property preservation . . . 72
2.5 Network Assumptions used for authorized segments. . . . 75
2.6 Parametrized dataplane protocol for directed protocols . . . 77
2.6.1 Hop validation check, authorized segments, and path extraction. . . . 77
2.6.2 Assumptions of the parametrized model . . . 79
2.6.3 Lemmas that are needed for the refinement proof . . . 80
2.7 Parametrized dataplane protocol for undirected protocols . . . 87
2.7.1 Hop validation check, authorized segments, and path extraction. . . . 87

3 Instances **92**
3.1 SCION . . . 93
3.1.1 Hop validation check and extract functions . . . 93
3.1.2 Definitions and properties of the added intruder knowledge . . . 94
3.1.3 Properties of the intruder knowledge, including *ik-add* and *ik-oracle* . 95
3.1.4 Direct proof goals for interpretation of *dataplane-3-directed* . . . 96
3.1.5 Instantiation of *dataplane-3-directed* locale . . . 97
3.2 SCION . . . 99
3.2.1 Hop validation check and extract functions . . . 99
3.2.2 Definitions and properties of the added intruder knowledge . . . 100
3.2.3 Properties of the intruder knowledge, including *ik-add* and *ik-oracle* . 101
3.2.4 Direct proof goals for interpretation of *dataplane-3-directed* . . . 102
3.2.5 Instantiation of *dataplane-3-directed* locale . . . 103
3.3 EPIC Level 1 in the Basic Attacker Model . . . 105
3.3.1 Hop validation check and extract functions . . . 105
3.3.2 Definitions and properties of the added intruder knowledge . . . 107
3.3.3 Properties of the intruder knowledge, including *ik-add* and *ik-oracle* . 107
3.3.4 Direct proof goals for interpretation of *dataplane-3-directed* . . . 110
3.3.5 Instantiation of *dataplane-3-directed* locale . . . 111
3.4 EPIC Level 1 in the Strong Attacker Model . . . 112
3.4.1 Hop validation check and extract functions . . . 112
3.4.2 Definitions and properties of the added intruder knowledge . . . 114
3.4.3 Properties of the intruder knowledge, including *ik-add* and *ik-oracle* . 115
3.4.4 Direct proof goals for interpretation of *dataplane-3-directed* . . . 118

3.4.5 Instantiation of *dataplane-3-directed* locale 119
3.5 EPIC Level 1 Example instantiation of locale 120
3.5.1 Left segment 120
3.5.2 Right segment 120
3.5.3 Executability 124
3.6 EPIC Level 2 in the Strong Attacker Model 128
3.6.1 Hop validation check and extract functions 128
3.6.2 Definitions and properties of the added intruder knowledge 130
3.6.3 Properties of the intruder knowledge, including *ik-add* and *ik-oracle* . 131
3.6.4 Direct proof goals for interpretation of *dataplane-3-directed* 134
3.6.5 Instantiation of *dataplane-3-directed* locale 135
3.7 ICING 137
3.7.1 Hop validation check and extract functions 137
3.7.2 Definitions and properties of the added intruder knowledge 139
3.7.3 Properties of the intruder knowledge, including *ik-add* and *ik-oracle* . 140
3.7.4 Direct proof goals for interpretation of *dataplane-3-undirected* 141
3.7.5 Instantiation of *dataplane-3-undirected* locale 142
3.8 ICING variant 143
3.8.1 Hop validation check and extract functions 143
3.8.2 Definitions and properties of the added intruder knowledge 145
3.8.3 Properties of the intruder knowledge, including *ik-add* and *ik-oracle* . 145
3.8.4 Direct proof goals for interpretation of *dataplane-3-undirected* 146
3.8.5 Instantiation of *dataplane-3-undirected* locale 147
3.9 ICING variant 148
3.9.1 Hop validation check and extract functions 148
3.9.2 Definitions and properties of the added intruder knowledge 149
3.9.3 Properties of the intruder knowledge, including *ik-add* and *ik-oracle* . 150
3.9.4 Direct proof goals for interpretation of *dataplane-3-undirected* 151
3.9.5 Instantiation of *dataplane-3-undirected* locale 152
3.10 All Protocols 153

This is a generated file containing all of our models, from abstract to parametrized to protocol instances, that we formalized in Isabelle/HOL in a human-readable form. The theory dependencies given in the figure on the next page are useful. Nevertheless, the most convenient way of browsing the Isabelle theories is to use the Isabelle GUI. See the README for details.

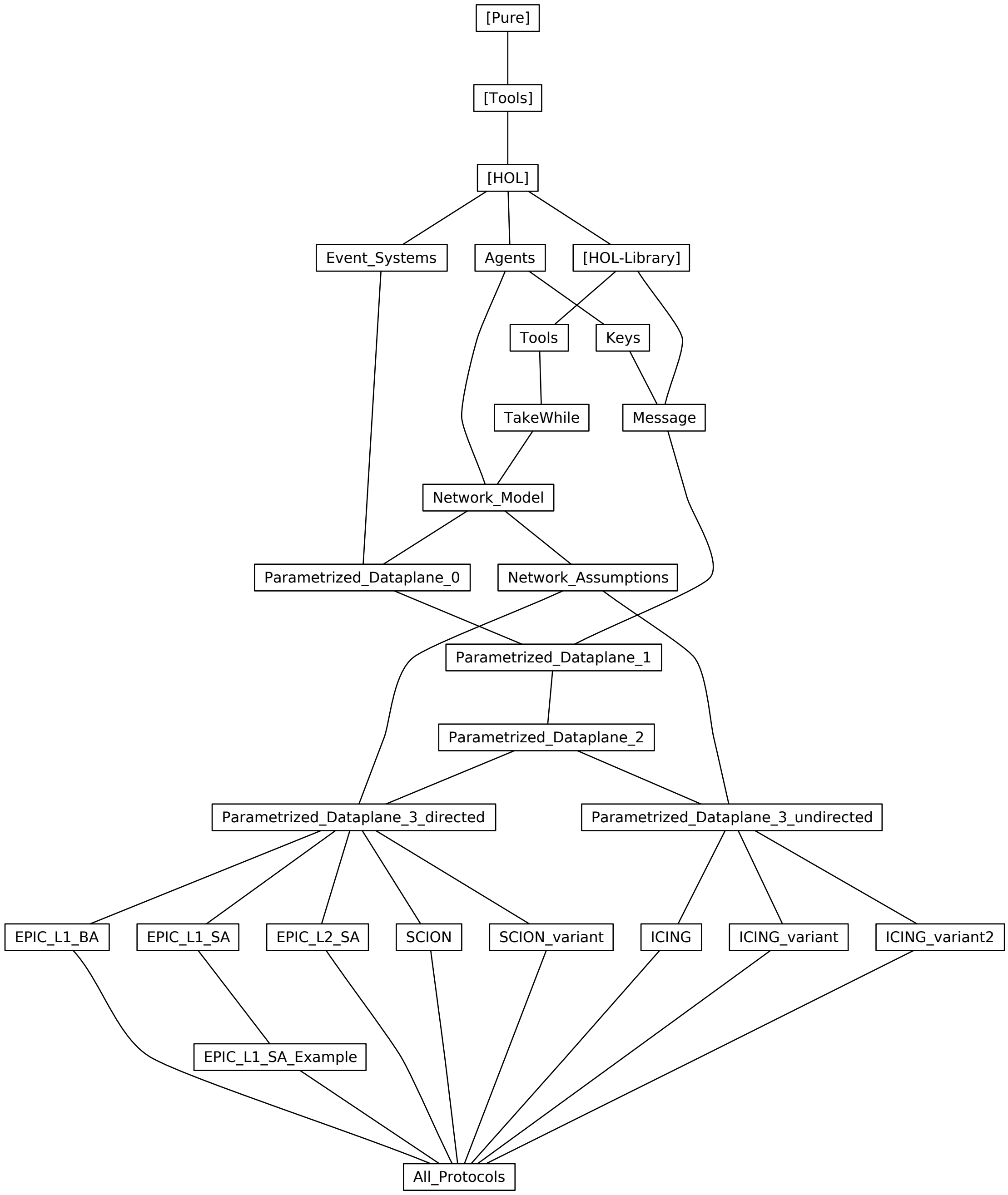

Figure 1: Theory dependencies

Chapter 1

Verification Infrastructure

Here we define event systems, the term algebra, and the Dolev–Yao adversary

1.1 Event Systems

This theory contains definitions of event systems, trace, traces, reachability, simulation, and proves the soundness of simulation for proving trace inclusion. We also derive some related simulation rules.

```
theory Event-Systems
  imports Main
begin

record ('e, 's) ES =
  init :: 's ⇒ bool
  trans :: 's ⇒ 'e ⇒ 's ⇒ bool   ((4-: -−-→ -) [50, 50, 50] 90)
```

1.1.1 Reachable states and invariants

```
inductive
  reach :: ('e, 's) ES ⇒ 's ⇒ bool for E
  where
    reach-init [simp, intro]:  init E s ⟹ reach E s
  | reach-trans [intro]: ⟦ E: s −e→ s'; reach E s ⟧ ⟹ reach E s'

thm reach.induct
```

Abbreviation for stating that a predicate is an invariant of an event system.

```
definition Inv :: ('e, 's) ES ⇒ ('s ⇒ bool) ⇒ bool where
  Inv E I ⟷ (∀ s. reach E s ⟶ I s)

lemmas InvI = Inv-def [THEN iffD2, rule-format]
lemmas InvE [elim] = Inv-def [THEN iffD1, elim-format, rule-format]

lemma Invariant-rule [case-names Inv-init Inv-trans]:
  assumes ⋀s0. init E s0 ⟹ I s0
     and ⋀s e s'. ⟦E: s −e→ s'; reach E s; I s⟧ ⟹ I s'
  shows Inv E I
  unfolding Inv-def
proof (intro allI impI)
  fix s
  assume reach E s
  then show I s using assms
    by (induction s rule: reach.induct) (auto)
qed
```

Invariant rule that allows strengthening the proof with another invariant.

```
lemma Invariant-rule-Inv [case-names Inv-other Inv-init Inv-trans]:
  assumes Inv E J
     and ⋀s0. init E s0 ⟹ I s0
     and ⋀s e s'. ⟦E: s −e→ s'; reach E s; I s; J s; J s'⟧ ⟹ I s'
  shows Inv E I
  unfolding Inv-def
proof (intro allI impI)
  fix s
  assume reach E s
```

then show $I\ s$ **using** *assms*
by (*induction s rule*: *reach.induct*)(*auto 3 4*)
qed

1.1.2 Traces

type-synonym $'e\ trace = 'e\ list$

inductive
trace :: $('e, 's)\ ES \Rightarrow 's \Rightarrow 'e\ trace \Rightarrow 's \Rightarrow bool$ $((4\text{-: - }-\langle\text{-}\rangle\rightarrow\text{ -})\ [50, 50, 50]\ 90)$
for $E\ s$
where
trace-nil [*simp*,*intro*!]:
E: $s\ -\langle[]\rangle\rightarrow\ s$
| *trace-snoc* [*intro*]:
$[\![\ E\text{: } s\ -\langle\tau\rangle\rightarrow\ s';\ E\text{: } s'\ -e\rightarrow\ s''\ ]\!] \Longrightarrow E\text{: } s\ -\langle\tau\ @\ [e]\rangle\rightarrow\ s''$

thm *trace.induct*

inductive-cases *trace-nil-invert* [*elim*!]: E: $s\ -\langle[]\rangle\rightarrow\ t$
inductive-cases *trace-snoc-invert* [*elim*]: E: $s\ -\langle\tau\ @\ [e]\rangle\rightarrow\ t$

lemma *trace-init-independence* [*elim*]:
assumes E: $s\ -\langle\tau\rangle\rightarrow\ s'$ $trans\ E = trans\ F$
shows F: $s\ -\langle\tau\rangle\rightarrow\ s'$
using *assms*
by (*induction rule*: *trace.induct*) *auto*

lemma *trace-single* [*simp*, *intro*!]: $[\![\ E\text{: } s\ -e\rightarrow\ s'\ ]\!] \Longrightarrow E\text{: } s\ -\langle[e]\rangle\rightarrow\ s'$
by (*auto intro*: *trace-snoc*[**where** $\tau = []$, *simplified*])

Next, we prove an introduction rule for a "cons" trace and a case analysis rule distinguishing the empty trace and a "cons" trace.

lemma *trace-consI*:
assumes
E: $s''\ -\langle\tau\rangle\rightarrow\ s'$ E: $s\ -e\rightarrow\ s''$
shows
E: $s\ -\langle e\ \#\ \tau\rangle\rightarrow\ s'$
using *assms*
by (*induction rule*: *trace.induct*) (*auto dest*: *trace-snoc*)

lemma *trace-cases-cons*:
assumes
E: $s\ -\langle\tau\rangle\rightarrow\ s'$
$[\![\ \tau = [];\ s' = s\ ]\!] \Longrightarrow P$
$\bigwedge e\ \tau'\ s''.\ [\![\ \tau = e\ \#\ \tau';\ E\text{: } s\ -e\rightarrow\ s'';\ E\text{: } s''\ -\langle\tau'\rangle\rightarrow\ s'\ ]\!] \Longrightarrow P$
shows P
using *assms*
by (*induction rule*: *trace.induct*) *fastforce*+

lemma *trace-consD*: $(E\text{: } s\ -\langle e\ \#\ \tau\rangle\rightarrow\ s') \Longrightarrow \exists\ s''.\ (E\text{: } s\ -e\rightarrow\ s'') \wedge (E\text{: } s''\ -\langle\tau\rangle\rightarrow\ s')$
by(*auto elim*: *trace-cases-cons*)

We show how a trace can be appended to another.

lemma *trace-append*: $(E\colon s\ -\langle\tau_1\rangle\!\rightarrow\ s') \wedge (E\colon s'\ -\langle\tau_2\rangle\!\rightarrow\ s'') \Longrightarrow E\colon s\ -\langle\tau_1 @ \tau_2\rangle\!\rightarrow\ s''$
  by(*induction* τ_1 *arbitrary*: *s*)
    (*auto dest*!: *trace-consD intro*: *trace-consI*)

lemma *trace-append-invert*: $(E\colon s\ -\langle\tau_1 @ \tau_2\rangle\!\rightarrow\ s'') \Longrightarrow \exists\, s'.\ (E\colon s\ -\langle\tau_1\rangle\!\rightarrow\ s') \wedge (E\colon s'\ -\langle\tau_2\rangle\!\rightarrow\ s'')$
  by (*induction* τ_1 *arbitrary*: *s*) (*auto intro*!: *trace-consI dest*!: *trace-consD*)

We prove an induction scheme for combining two traces, similar to *list-induct2*.

lemma *trace-induct2* [*consumes 3*, *case-names Nil Snoc*]:
  $[\![E\colon s\ -\langle\tau\rangle\!\rightarrow\ s''$; $F\colon t\ -\langle\sigma\rangle\!\rightarrow\ t''$; $length\ \tau = length\ \sigma$;
    $P\ []\ s\ []\ t$;
    $\bigwedge\tau\ s'\ e\ s''\ \sigma\ t'\ f\ t''.$
    $[\![E\colon s\ -\langle\tau\rangle\!\rightarrow\ s'$; $E\colon s' - e\!\rightarrow\ s''$; $F\colon t\ -\langle\sigma\rangle\!\rightarrow\ t'$; $F\colon t' - f\!\rightarrow\ t''$; $P\ \tau\ s'\ \sigma\ t']\!]$
      $\Longrightarrow P\ (\tau\ @\ [e])\ s''\ (\sigma\ @\ [f])\ t'']\!]$
  $\Longrightarrow P\ \tau\ s''\ \sigma\ t''$
proof (*induction* τ s'' *arbitrary*: σ t'' *rule*: *trace.induct*)
  case *trace-nil*
  then show *?case* **by** *auto*
next
  case (*trace-snoc* τ s' e s'')
  from ‹$length\ (\tau\ @\ [e]) = length\ \sigma$› **and** ‹$F\colon t\ -\langle\sigma\rangle\!\rightarrow\ t''$›
  obtain $f\ \sigma'\ t'$
    where $\sigma = \sigma'\ @\ [f]$ $length\ \tau = length\ \sigma'$ $F\colon t\ -\langle\sigma'\rangle\!\rightarrow\ t'$ $F\colon t'\ -f\!\rightarrow\ t''$
    by (*auto elim*: *trace.cases*)
  then show *?case* **using** *trace-snoc* **by** *blast*
qed

Relate traces to reachability and invariants

lemma *reach-trace-equiv*: $reach\ E\ s \longleftrightarrow (\exists\, s0\ \tau.\ init\ E\ s0 \wedge E\colon s0\ -\langle\tau\rangle\!\rightarrow\ s)$ (**is** $?A \longleftrightarrow ?B$)
proof
  assume *?A* **then show** *?B*
    by (*induction s rule*: *reach.induct*) *auto*
next
  assume *?B*
  then obtain $s0\ \tau$ **where** $E\colon s0\ -\langle\tau\rangle\!\rightarrow\ s$ $init\ E\ s0$ **by** *blast*
  then show *?A*
    by (*induction* τ *s rule*: *trace.induct*) *auto*
qed

lemma *reach-traceI*: $[\![init\ E\ s0;\ E\colon s0\ -\langle\tau\rangle\!\rightarrow\ s]\!] \Longrightarrow reach\ E\ s$
  by(*auto simp add*: *reach-trace-equiv*)

lemma *reach-trace-extend*: $[\![E\colon s\ -\langle\tau\rangle\!\rightarrow\ s';\ reach\ E\ s]\!] \Longrightarrow reach\ E\ s'$
  by (*induction* τ s' *rule*: *trace.induct*) *auto*

lemma *Inv-trace*: $[\![Inv\ E\ I;\ init\ E\ s0;\ E\colon s0\ -\langle\tau\rangle\!\rightarrow\ s']\!] \Longrightarrow I\ s'$
  by (*auto simp add*: *Inv-def reach-trace-equiv*)

Trace semantics of event systems

We define the set of traces of an event system.

definition *traces* :: $('e, 's)$ *ES* $\Rightarrow$ *'e trace set* **where**
  traces $E = \{\tau.\ \exists s\ s'.\ init\ E\ s \wedge E\colon s\ -\langle\tau\rangle\rightarrow\ s'\}$

lemma *tracesI* [*intro*]: $[\![\ init\ E\ s;\ E\colon s\ -\langle\tau\rangle\rightarrow\ s'\ ]\!] \Longrightarrow \tau \in traces\ E$
  by (*auto simp add*: *traces-def*)

lemma *tracesE* [*elim*]: $[\![\ \tau \in traces\ E;\ \bigwedge s\ s'.\ [\![\ init\ E\ s;\ E\colon s\ -\langle\tau\rangle\rightarrow\ s'\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
  by (*auto simp add*: *traces-def*)

lemma *traces-nil* [*simp*, *intro*!]: $init\ E\ s \Longrightarrow [] \in traces\ E$
  by (*auto simp add*: *traces-def*)

We now define a trace property satisfaction relation: an event system satisfies a property φ, if its traces are contained in φ.

definition *trace-property* :: $('e, 's)$ *ES* $\Rightarrow$ *'e trace set* $\Rightarrow$ *bool* (**infix** $\models_{ES}$ *90*) **where**
  $E \models_{ES} \varphi \longleftrightarrow traces\ E \subseteq \varphi$

lemmas *trace-propertyI* = *trace-property-def* [*THEN iffD2*, *OF subsetI*, *rule-format*]
lemmas *trace-propertyE* [*elim*] = *trace-property-def* [*THEN iffD1*, *THEN subsetD*, *elim-format*]
lemmas *trace-propertyD* = *trace-property-def* [*THEN iffD1*, *THEN subsetD*, *rule-format*]

Rules for showing trace properties using a stronger trace-state invariant.

lemma *trace-invariant*:
  assumes
    $\tau \in traces\ E$
    $\bigwedge s\ s'.\ [\![\ init\ E\ s;\ E\colon s\ -\langle\tau\rangle\rightarrow\ s'\ ]\!] \Longrightarrow I\ \tau\ s'$
    $\bigwedge s.\ I\ \tau\ s \Longrightarrow \tau \in \varphi$
  shows $\tau \in \varphi$ **using** *assms*
  by (*auto*)

lemma *trace-property-rule*:
  assumes
    $\bigwedge s0.\ init\ E\ s0 \Longrightarrow I\ []\ s0$
    $\bigwedge s\ s'\ \tau\ e\ s''.$
      $[\![\ init\ E\ s;\ E\colon s\ -\langle\tau\rangle\rightarrow\ s';\ E\colon s'\ -e\rightarrow\ s'';\ I\ \tau\ s';\ reach\ E\ s'\ ]\!] \Longrightarrow I\ (\tau @ [e])\ s''$
    $\bigwedge\tau\ s.\ [\![\ I\ \tau\ s;\ reach\ E\ s\ ]\!] \Longrightarrow \tau \in \varphi$
  shows $E \models_{ES} \varphi$
proof (*rule trace-propertyI*, *erule trace-invariant*[**where** $I{=}\lambda\tau\ s.\ I\ \tau\ s \wedge reach\ E\ s$])
  fix $\tau\ s\ s'$
  assume $E\colon s\ -\langle\tau\rangle\rightarrow\ s'$ **and** $init\ E\ s$
  then show $I\ \tau\ s' \wedge reach\ E\ s'$
    by (*induction* $\tau\ s'$ *rule*: *trace.induct*) (*auto simp add*: *assms*)
qed (*auto simp add*: *assms*)

Similar to $[\![\bigwedge s0.\ init\ ?E\ s0 \Longrightarrow\ ?I\ []\ s0;\ \bigwedge s\ s'\ \tau\ e\ s''.\ [\![init\ ?E\ s;\ ?E\colon s\ -\langle\tau\rangle\rightarrow\ s';\ ?E\colon s'{-}e\rightarrow\ s'';\ ?I\ \tau\ s';\ reach\ ?E\ s']\!] \Longrightarrow\ ?I\ (\tau\ @\ [e])\ s'';\ \bigwedge\tau\ s.\ [\![?I\ \tau\ s;\ reach\ ?E\ s]\!] \Longrightarrow \tau \in\ ?\varphi]\!] \Longrightarrow\ ?E \models_{ES}\ ?\varphi$, but allows matching pure state invariants directly.

lemma *Inv-trace-property*:

assumes $Inv\ E\ I$ **and** $[] \in \varphi$
and $(\bigwedge s\ \tau\ s'\ e\ s''.$
$[\![init\ E\ s;\ E\colon s\ -\langle\tau\rangle\to\ s';\ E\colon s'\ -e\to\ s'';\ I\ s;\ I\ s';\ reach\ E\ s';\ \tau \in \varphi]\!] \Longrightarrow \tau@[e] \in \varphi)$
shows $E \models_{ES} \varphi$
using *assms(1,2)*
by $(intro\ trace\text{-}property\text{-}rule[$**where** $I{=}\lambda\tau\ s.\ \tau \in \varphi])\ (auto\ intro\colon assms(3))$

1.1.3 Simulation

We first define the simulation preorder on pairs of states and derive a series of useful coinduction principles.

coinductive
$sim :: ('e,\ 's\ )\ ES \Rightarrow ('f,\ 't\ )\ ES \Rightarrow ('e \Rightarrow 'f) \Rightarrow 's \Rightarrow 't \Rightarrow bool$
for $E\ F\ \pi$
where
$[\![\ \bigwedge e\ s'.\ (E\colon s\ -e\to\ s') \Longrightarrow \exists\, t'.\ (F\colon t\ -\pi\ e\to\ t') \wedge sim\ E\ F\ \pi\ s'\ t'\ ]\!] \Longrightarrow sim\ E\ F\ \pi\ s\ t$

abbreviation
$simS :: ('e,\ 's\ )\ ES \Rightarrow ('f,\ 't\ )\ ES \Rightarrow 's \Rightarrow ('e \Rightarrow 'f) \Rightarrow 't \Rightarrow bool$
$((5\text{-},\text{-}\colon\ \text{-}\ \sqsubseteq_{\text{-}}\ \text{-})\ [50,\ 50,\ 50,\ 60,\ 50]\ 90)$
where
$simS\ E\ F\ s\ \pi\ t \equiv sim\ E\ F\ \pi\ s\ t$

lemmas $sim\text{-}coinduct\text{-}id = sim.coinduct[$**where** $\pi{=}id,\ consumes\ 1,\ case\text{-}names\ sim]$

We prove a simplified and slightly weaker coinduction rule for simulation and register it as the default rule for *sim*.

lemma $sim\text{-}coinduct\text{-}weak\ [consumes\ 1,\ case\text{-}names\ sim,\ coinduct\ pred\colon sim]$:
assumes
$R\ s\ t$
$\bigwedge s\ t\ a\ s'.\ [\![\ R\ s\ t;\ \ E\colon s{-}a\to\ s']\!] \Longrightarrow (\exists\, t'.\ (F\colon t{-}\pi\ a\to\ t') \wedge R\ s'\ t')$
shows
$E,F\colon s \sqsubseteq_\pi\ t$
using *assms*
by $(coinduction\ arbitrary\colon s\ t\ rule\colon sim.coinduct)\ (fastforce)$

lemma $sim\text{-}refl\colon E,E\colon s \sqsubseteq_i d\ s$
by $(coinduction\ arbitrary\colon s)\ auto$

lemma $sim\text{-}trans\colon [\![\ E,F\colon s \sqsubseteq_\pi 1\ t;\ F,G\colon t \sqsubseteq_\pi 2\ u\ ]\!] \Longrightarrow E,G\colon s \sqsubseteq_(\pi 2 \circ \pi 1)\ u$
proof $(coinduction\ arbitrary\colon s\ t\ u)$
case $(sim\ a\ s'\ s\ t)$
with $\langle E,F\colon s \sqsubseteq_\pi 1\ t\rangle$ **obtain** t' **where** $F\colon t\ -\pi 1\ a\to\ t'\ E,F\colon s' \sqsubseteq_\pi 1\ t'$
by $(cases\ rule\colon sim.cases)\ auto$
moreover
from $\langle F,G\colon t \sqsubseteq_\pi 2\ u\rangle\ \langle F\colon t\ -\pi 1\ a\to\ t'\rangle$ **obtain** u' **where** $G\colon u\ -\pi 2\ (\pi 1\ a)\to\ u'\ F,G\colon t' \sqsubseteq_\pi 2\ u'$
by $(cases\ rule\colon sim.cases)\ auto$
ultimately
have $\exists\, t'\ u'.\ (G\colon u\ -\pi 2\ (\pi 1\ a)\to\ u') \wedge (E,F\colon s' \sqsubseteq_\pi 1\ t') \wedge (F,G\colon t' \sqsubseteq_\pi 2\ u')$

by *auto*
then show *?case* **by** *auto*
qed

Extend transition simulation to traces.

lemma *trace-sim*:
assumes E: $s\ -\langle\tau\rangle\rightarrow\ s'$ E,F: $s \sqsubseteq_\pi t$
shows $\exists\, t'.\ (F\colon t\ -\langle map\ \pi\ \tau\rangle\rightarrow\ t') \wedge (E,F\colon s' \sqsubseteq_\pi t')$
using *assms*
proof (*induction* τ s' *rule*: *trace.induct*)
case *trace-nil*
then show *?case* **by** *auto*
next
case (*trace-snoc* τ s' e s'')
then obtain t' **where** F: $t\ -\langle map\ \pi\ \tau\rangle\rightarrow\ t'$ E,F: $s' \sqsubseteq_\pi t'$ **by** *auto*
from ‹E,F: $s' \sqsubseteq_\pi t'$› ‹E: $s'-e\rightarrow\ s''$›
obtain t'' **where** F: $t'\ -\pi\ e\rightarrow\ t''$ E,F: $s'' \sqsubseteq_\pi t''$ **by** (*elim sim.cases*) *fastforce*
then show *?case* **using** ‹F: $t\ -\langle map\ \pi\ \tau\rangle\rightarrow\ t'$› ‹$E$: $s\ -\langle\tau\rangle\rightarrow\ s'$› ‹$E$: $s'-e\rightarrow\ s''$› **by** *auto*
qed

Simulation for event systems

definition
sim-ES :: $('e,\ 's\ )\ ES \Rightarrow ('e \Rightarrow 'f) \Rightarrow ('f,\ 't\ )\ ES \Rightarrow bool$ ((*3-* $\sqsubseteq_{\text{-}}$ *-*) [*50*, *60*, *50*] *95*)
where
$E \sqsubseteq_\pi F \longleftrightarrow (\exists\, R.$
$(\forall\, s0.\ init\ E\ s0 \longrightarrow (\exists\, t0.\ init\ F\ t0 \wedge R\ s0\ t0)) \wedge$
$(\forall\, s\ t.\ R\ s\ t \longrightarrow E,F\colon s \sqsubseteq_\pi t))$

lemma *sim-ES-I*:
assumes
$\bigwedge s0.\ init\ E\ s0 \Longrightarrow (\exists\, t0.\ init\ F\ t0 \wedge R\ s0\ t0)$ **and**
$\bigwedge s\ t.\ R\ s\ t \Longrightarrow E,F\colon s \sqsubseteq_\pi t$
shows $E \sqsubseteq_\pi F$
using *assms*
by (*auto simp add*: *sim-ES-def*)

lemma *sim-ES-E*:
assumes
$E \sqsubseteq_\pi F$
$\bigwedge R.\ [\![\ \bigwedge s0.\ init\ E\ s0 \Longrightarrow (\exists\, t0.\ init\ F\ t0 \wedge R\ s0\ t0);\ \bigwedge s\ t.\ R\ s\ t \Longrightarrow E,F\colon s \sqsubseteq_\pi t\ ]\!] \Longrightarrow P$
shows P
using *assms*
by (*auto simp add*: *sim-ES-def*)

Different rules to set up a simulation proof. Include reachability or weaker invariant(s) in precondition of "simulation square".

lemma *simulate-ES*:
assumes
init: $\bigwedge s0.\ init\ E\ s0 \Longrightarrow (\exists\, t0.\ init\ F\ t0 \wedge R\ s0\ t0)$ **and**
step: $\bigwedge s\ t\ a\ s'.\ [\![\ R\ s\ t;\ reach\ E\ s;\ reach\ F\ t;\ E\colon s-a\rightarrow\ s'\ ]\!]$
$\Longrightarrow (\exists\, t'.\ (F\colon t-\pi\ a\rightarrow\ t') \wedge R\ s'\ t')$

shows $E \sqsubseteq_\pi F$
by ($auto$ 4 4 $intro$!: $sim\text{-}ES\text{-}I$[**where** $R{=}\lambda s\ t.\ R\ s\ t \wedge reach\ E\ s \wedge reach\ F\ t$] $dest$: $init$
$intro$: $sim\text{-}coinduct\text{-}weak$[**where** $R{=}\lambda s\ t.\ R\ s\ t \wedge reach\ E\ s \wedge reach\ F\ t$] $dest$: $step$)

lemma *simulate-ES-with-invariants*:
assumes
init: $\bigwedge s0.\ init\ E\ s0 \Longrightarrow (\exists t0.\ init\ F\ t0 \wedge R\ s0\ t0)$ **and**
step: $\bigwedge s\ t\ a\ s'.$
$[\![\ R\ s\ t;\ I\ s;\ J\ t;\ E{:}\ s{-}a{\rightarrow}\ s'\ ]\!] \Longrightarrow (\exists t'.\ (F{:}\ t{-}\pi\ a{\rightarrow}\ t') \wedge R\ s'\ t')$ **and**
invE: $\bigwedge s.\ reach\ E\ s \longrightarrow I\ s$ **and**
invE: $\bigwedge t.\ reach\ F\ t \longrightarrow J\ t$
shows $E \sqsubseteq_\pi F$ **using** *assms*
by (*auto intro*: *simulate-ES*[**where** $R{=}R$])

lemmas *simulate-ES-with-invariant* = *simulate-ES-with-invariants*[**where** $J{=}\lambda s.\ True$, *simplified*]

Variants with a functional simulation relation, aka refinement mapping.

lemma *simulate-ES-fun*:
assumes
init: $\bigwedge s0.\ init\ E\ s0 \Longrightarrow init\ F\ (h\ s0)$ **and**
step: $\bigwedge s\ a\ s'.\ [\![\ E{:}\ s{-}a{\rightarrow}\ s';\ reach\ E\ s;\ reach\ F\ (h\ s)\ ]\!] \Longrightarrow F{:}\ h\ s{-}\pi\ a{\rightarrow}\ h\ s'$
shows $E \sqsubseteq_\pi F$
using *assms*
by (*auto intro*!: *simulate-ES*[**where** $R{=}\lambda s\ t.\ t = h\ s$])

lemma *simulate-ES-fun-with-invariants*:
assumes
init: $\bigwedge s0.\ init\ E\ s0 \Longrightarrow init\ F\ (h\ s0)$ **and**
step: $\bigwedge s\ a\ s'.\ [\![\ E{:}\ s{-}a{\rightarrow}\ s';\ I\ s;\ J\ (h\ s)\ ]\!] \Longrightarrow F{:}\ h\ s{-}\pi\ a{\rightarrow}\ h\ s'$ **and**
invE: $\bigwedge s.\ reach\ E\ s \longrightarrow I\ s$ **and**
invF: $\bigwedge t.\ reach\ F\ t \longrightarrow J\ t$
shows $E \sqsubseteq_\pi F$
using *assms*
by (*auto intro*!: *simulate-ES-fun*)

lemmas *simulate-ES-fun-with-invariant* =
simulate-ES-fun-with-invariants[**where** $J{=}\lambda t.\ True$, *simplified*]

Reflexivity and transitivity for ES simulation.

lemma *sim-ES-refl*: $E \sqsubseteq_i d\ E$
by (*auto intro*: *sim-ES-I*[**where** $R{=}(=)$] *sim-refl*)

lemma *sim-ES-trans*:
assumes $E \sqsubseteq_\pi 1\ F$ **and** $F \sqsubseteq_\pi 2\ G$ **shows** $E \sqsubseteq_(\pi 2 \circ \pi 1)\ G$
proof –
from $\langle E \sqsubseteq_\pi 1\ F\rangle$ **obtain** R_1 **where**
$\bigwedge s0.\ init\ E\ s0 \Longrightarrow (\exists t0.\ init\ F\ t0 \wedge R_1\ s0\ t0)$
$\bigwedge s\ t.\ R_1\ s\ t \Longrightarrow E{,}F{:}\ s \sqsubseteq_\pi 1\ t$
by (*auto elim*!: *sim-ES-E*)
moreover
from $\langle F \sqsubseteq_\pi 2\ G\rangle$ **obtain** R_2 **where**
$\bigwedge t0.\ init\ F\ t0 \Longrightarrow (\exists u0.\ init\ G\ u0 \wedge R_2\ t0\ u0)$

$\bigwedge t\ u.\ R_2\ t\ u \Longrightarrow F,G\colon t \sqsubseteq_\pi 2\ u$
by (*auto elim*!: *sim-ES-E*)
ultimately show *?thesis*
by (*auto intro*!: *sim-ES-I*[**where** $R=R_1$ *OO* R_2] *sim-trans simp add*: *OO-def*) *blast*
qed

Soundness for trace inclusion and property preservation

lemma *simulation-soundness*: $E \sqsubseteq_\pi F \Longrightarrow (map\ \pi)`traces\ E \subseteq traces\ F$
by (*fastforce simp add*: *sim-ES-def image-def dest*: *trace-sim*)

lemmas *simulation-rule* = *simulate-ES* [*THEN simulation-soundness*]
lemmas *simulation-rule-id* = *simulation-rule*[**where** π=*id*, *simplified*]

This allows us to show that properties are preserved under simulation.

corollary *property-preservation*:
$[\![E \sqsubseteq_\pi F;\ F \models_{ES} P;\ \bigwedge\tau.\ map\ \pi\ \tau \in P \Longrightarrow \tau \in Q\ ]\!] \Longrightarrow E \models_{ES} Q$
by (*auto simp add*: *trace-property-def dest*: *simulation-soundness*)

1.1.4 Simulation up to simulation preorder

lemma *sim-coinduct-upto-sim* [*consumes 1*, *case-names sim*]:
assumes
major: $R\ s\ t$ **and**
S: $\bigwedge s\ t\ a\ s'.\ [\![\ R\ s\ t;\ E\colon s\ {-a\rightarrow}\ s']\!] \Longrightarrow$
$\exists t'.\ (F\colon t\ {-\pi\ a\rightarrow}\ t') \wedge ((sim\ E\ E\ id)\ OO\ R\ OO\ (sim\ F\ F\ id))\ s'\ t'$
shows
$E,F\colon s \sqsubseteq_\pi t$
proof −
let *?R-upto* = ((*sim E E id*) *OO R OO* (*sim F F id*))
from *major* **have** *?R-upto s t* **by** (*auto intro*: *sim-refl*)
then show *?thesis*
proof (*coinduction arbitrary*: *s t*)
case (*sim a s' s t*)
from ‹((*sim E E id*) *OO R OO* (*sim F F id*)) *s t*› **obtain** *s1 t1* **where**
$E,E\colon s \sqsubseteq_i d\ s1\ R\ s1\ t1\ F,F\colon t1 \sqsubseteq_i d\ t$ **by** (*elim relcomppE*)
from ‹$E,E\colon s \sqsubseteq_i d\ s1$› ‹$E\colon s{-a\rightarrow}\ s'$›
obtain $s1'$ **where** $E\colon s1{-a\rightarrow}\ s1'\ E,E\colon s' \sqsubseteq_i d\ s1'$ **by** (*cases rule*: *sim.cases*) *auto*
from ‹*R s1 t1*› ‹$E\colon s1{-a\rightarrow}\ s1'$› S
obtain $t1'$ **where** $F\colon t1{-\pi\ a\rightarrow}\ t1'$ *?R-upto* $s1'\ t1'$ **by** *force*
from ‹$F,F\colon t1 \sqsubseteq_i d\ t$› ‹$F\colon t1{-\pi\ a\rightarrow}\ t1'$›
obtain t' **where** $F\colon t{-\pi\ a\rightarrow}\ t'\ F,F\colon t1' \sqsubseteq_i d\ t'$ **by** (*cases rule*: *sim.cases*) *auto*
from ‹$F\colon t{-\pi\ a\rightarrow}\ t'$› ‹$E,E\colon s' \sqsubseteq_i d\ s1'$› ‹*?R-upto* $s1'\ t1'$› ‹$F,F\colon t1' \sqsubseteq_i d\ t'$›
have ((*sim E E id*) *OO R OO* (*sim F F id*)) $s'\ t'$
apply(*auto simp add*: *OO-def*) **using** *comp-id sim-trans* **by** *metis*
then have $\exists t'.\ (F\colon t{-\pi\ a\rightarrow}\ t') \wedge$ *?R-upto* $s'\ t'$
using ‹$F\colon t{-\pi\ a\rightarrow}\ t'$› **by** (*auto intro*: *sim-trans*)
then show *?case* **using** S **by** *fastforce*
qed
qed

end

1.2 Atomic messages

theory *Agents* **imports** *Main*
begin

The definitions below are moved here from the message theory, since the higher levels of protocol abstraction do not know about cryptographic messages.

1.2.1 Agents

type-synonym *as* = *nat*

type-synonym *aso* = *as option*

type-synonym *ases* = *as set*

locale *compromised* =
fixes
  bad :: *as set* — compromised ASes
begin

abbreviation
  good :: *as set*
where
  good ≡ −*bad*
end

1.2.2 Nonces and keys

We have an unspecified type of freshness identifiers. For executability, we may need to assume that this type is infinite.

typedecl *fid-t*

datatype *fresh-t* =
  mk-fresh fid-t nat (**infixr** $ *65*)

fun *fid* :: *fresh-t* ⇒ *fid-t* **where**
  fid (*f* $ *n*) = *f*

fun *num* :: *fresh-t* ⇒ *nat* **where**
  num (*f* $ *n*) = *n*

Nonces

type-synonym
  nonce = *fresh-t*

end

1.3 Symmetric and Asymetric Keys

theory *Keys* **imports** *Agents* **begin**

Divide keys into session and long-term keys. Define different kinds of long-term keys in second step.

datatype *key* = — long-term keys
  macK as — local MACing key
| *pubK as* — as's public key
| *priK as* — as's private key

The inverse of a symmetric key is itself; that of a public key is the private key and vice versa

fun *invKey* :: *key* ⇒ *key* **where**
  invKey (*pubK A*) = *priK A*
| *invKey* (*priK A*) = *pubK A*
| *invKey K* = *K*

definition
  symKeys :: *key set* **where**
  symKeys ≡ {*K*. *invKey K* = *K*}

lemma *invKey-K*: *K* ∈ *symKeys* ⟹ *invKey K* = *K*
by (*simp add*: *symKeys-def*)

Most lemmas we need come for free with the inductive type definition: injectiveness and distinctness.

lemma *invKey-invKey-id* [*simp*]: *invKey* (*invKey K*) = *K*
by (*cases K*, *auto*)

lemma *invKey-eq* [*simp*]: (*invKey K* = *invKey K′*) = (*K*=*K′*)
apply (*safe*)
apply (*drule-tac f*=*invKey* **in** *arg-cong*, *simp*)
done

We get most lemmas below for free from the inductive definition of type *key*. Many of these are just proved as a reality check.

1.3.1 Asymmetric Keys

No private key equals any public key (essential to ensure that private keys are private!). A similar statement an axiom in Paulson's theory!

lemma *privateKey-neq-publicKey*: *priK A* ≠ *pubK A′*
by *auto*

lemma *publicKey-neq-privateKey*: *pubK A* ≠ *priK A′*
by *auto*

1.3.2 Basic properties of *pubK* and *priK*

lemma *publicKey-inject* [*iff*]: (*pubK A* = *pubK A′*) = (*A* = *A′*)
by (*auto*)

lemma *not-symKeys-pubK* [*iff*]: $pubK\ A \notin symKeys$
by (*simp add*: *symKeys-def*)

lemma *not-symKeys-priK* [*iff*]: $priK\ A \notin symKeys$
by (*simp add*: *symKeys-def*)

lemma *symKey-neq-priK*: $K \in symKeys \Longrightarrow K \neq priK\ A$
by (*auto simp add*: *symKeys-def*)

lemma *symKeys-neq-imp-neq*: $(K \in symKeys) \neq (K' \in symKeys) \Longrightarrow K \neq K'$
by *blast*

lemma *symKeys-invKey-iff* [*iff*]: $(invKey\ K \in symKeys) = (K \in symKeys)$
by (*unfold symKeys-def*, *auto*)

1.3.3 "Image" equations that hold for injective functions

lemma *invKey-image-eq* [*simp*]: $(invKey\ x \in invKey`A) = (x \in A)$
by *auto*

lemma *invKey-pubK-image-priK-image* [*simp*]: $invKey\ `\ pubK\ `\ AS = priK\ `\ AS$
by (*auto simp add*: *image-def*)

lemma *publicKey-notin-image-privateKey*: $pubK\ A \notin priK\ `\ AS$
by *auto*

lemma *privateKey-notin-image-publicKey*: $priK\ x \notin pubK\ `\ AA$
by *auto*

lemma *publicKey-image-eq* [*simp*]: $(pubK\ x \in pubK\ `\ AA) = (x \in AA)$
by *auto*

lemma *privateKey-image-eq* [*simp*]: $(priK\ A \in priK\ `\ AS) = (A \in AS)$
by *auto*

1.3.4 Symmetric Keys

The following was stated as an axiom in Paulson's theory.

lemma *sym-shrK*: $macK\ X \in symKeys$ — All shared keys are symmetric
by (*simp add*: *symKeys-def*)

Symmetric keys and inversion

lemma *symK-eq-invKey*: $[\![\ SK = invKey\ K;\ SK \in symKeys\ ]\!] \Longrightarrow K = SK$
by (*auto simp add*: *symKeys-def*)

Image-related lemmas.

lemma *publicKey-notin-image-shrK*: $pubK\ x \notin macK\ `\ AA$
by *auto*

lemma *privateKey-notin-image-shrK*: $priK\ x \notin macK\ `\ AA$
by *auto*

lemma *shrK-notin-image-publicKey*: $macK\ x \notin pubK\ `\ AA$
by *auto*

lemma *shrK-notin-image-privateKey*: $macK\ x \notin priK\ `\ AA$
by *auto*

lemma *shrK-image-eq* [*simp*]: $(macK\ x \in macK\ `\ AA) = (x \in AA)$
by *auto*

end

1.4 Theory of ASes and Messages for Security Protocols

theory *Message* **imports** *Keys HOL−Library.Sublist*
begin

datatype *msgterm* =
  ε
  | *AS as* — Autonomous Systems, i.e. agents
  | *Num nat* — Ordinary integers, timestamps, ...
  | *Key key* — Crypto keys
  | *Nonce nonce* — Unguessable nonces
  | *L msgterm list* — Lists
  | *MPair msgterm msgterm* — Compound messages
  | *Hash msgterm* — Hashing
  | *Crypt key msgterm* — Encryption, public- or shared-key

Syntax sugar

syntax
  -MTuple :: $['a,\ args] \Rightarrow 'a * 'b$ ((*2*⟨-,/ -⟩))

syntax (*xsymbols*)
  -MTuple :: $['a,\ args] \Rightarrow 'a * 'b$ ((*2*⟨-,/ -⟩))

translations
  $\langle x,\ y,\ z\rangle \rightleftharpoons \langle x,\ \langle y,\ z\rangle\rangle$
  $\langle x,\ y\rangle \rightleftharpoons$ *CONST MPair x y*

syntax
  -MHF :: $['a,\ 'b\ ,\ 'c,\ 'd,\ 'e] \Rightarrow 'a * 'b * 'c * 'd * 'e$ ((*5HF*◁-,/ -,/ -,/ -,/ -▷))

abbreviation
  Mac :: [*msgterm*,*msgterm*] ⇒ *msgterm*   ((*4Mac*[-] /-) [*0*, *1000*])
where
  — Message Y paired with a MAC computed with the help of X
  Mac[*X*] *Y* ≡ *Hash* ⟨*X*,*Y*⟩

abbreviation *macKey* **where** *macKey a* ≡ *Key* (*macK a*)

definition
  keysFor :: *msgterm set* ⇒ *key set*
where
  — Keys useful to decrypt elements of a message set
  keysFor H ≡ *invKey* ‘ $\{K.\ \exists X.$ *Crypt K X* $\in H\}$

Inductive Definition of "All Parts" of a Message

inductive-set
  parts :: *msgterm set* ⇒ *msgterm set*
  for *H* :: *msgterm set*
  where
    Inj [*intro*]: $X \in H \Longrightarrow X \in$ *parts H*
  | *Fst*: $\langle X,\text{-}\rangle \in$ *parts H* $\Longrightarrow X \in$ *parts H*
  | *Snd*: $\langle \text{-},Y\rangle \in$ *parts H* $\Longrightarrow Y \in$ *parts H*

| *Lst*: $[\![L\ xs \in parts\ H;\ X \in set\ xs]\!] \Longrightarrow X \in parts\ H$

| *Body*: $Crypt\ K\ X \in parts\ H \Longrightarrow X \in parts\ H$

Monotonicity

lemma *parts-mono*: $G \subseteq H \Longrightarrow parts\ G \subseteq parts\ H$
apply *auto*
apply (*erule parts.induct*)
apply (*blast dest*: *parts.Fst parts.Snd parts.Lst parts.Body*)+
done

Equations hold because constructors are injective.

lemma *Other-image-eq* [*simp*]: $(AS\ x \in AS`A) = (x{:}A)$
by *auto*

lemma *Key-image-eq* [*simp*]: $(Key\ x \in Key`A) = (x{\in}A)$
by *auto*

lemma *AS-Key-image-eq* [*simp*]: $(AS\ x \notin Key`A)$
by *auto*

lemma *Num-Key-image-eq* [*simp*]: $(Num\ x \notin Key`A)$
by *auto*

1.4.1 keysFor operator

lemma *keysFor-empty* [*simp*]: $keysFor\ \{\} = \{\}$
by (*unfold keysFor-def*, *blast*)

lemma *keysFor-Un* [*simp*]: $keysFor\ (H \cup H') = keysFor\ H \cup keysFor\ H'$
by (*unfold keysFor-def*, *blast*)

lemma *keysFor-UN* [*simp*]: $keysFor\ (\bigcup i{\in}A.\ H\ i) = (\bigcup i{\in}A.\ keysFor\ (H\ i))$
by (*unfold keysFor-def*, *blast*)

Monotonicity

lemma *keysFor-mono*: $G \subseteq H \Longrightarrow keysFor\ G \subseteq keysFor\ H$
by (*unfold keysFor-def*, *blast*)

lemma *keysFor-insert-AS* [*simp*]: $keysFor\ (insert\ (AS\ A)\ H) = keysFor\ H$
by (*unfold keysFor-def*, *auto*)

lemma *keysFor-insert-Num* [*simp*]: $keysFor\ (insert\ (Num\ N)\ H) = keysFor\ H$
by (*unfold keysFor-def*, *auto*)

lemma *keysFor-insert-Key* [*simp*]: $keysFor\ (insert\ (Key\ K)\ H) = keysFor\ H$
by (*unfold keysFor-def*, *auto*)

lemma *keysFor-insert-Nonce* [*simp*]: $keysFor\ (insert\ (Nonce\ n)\ H) = keysFor\ H$
by (*unfold keysFor-def*, *auto*)

lemma *keysFor-insert-L* [*simp*]: $keysFor\ (insert\ (L\ X)\ H) = keysFor\ H$

by $(unfold\ keysFor\text{-}def,\ auto)$

lemma *keysFor-insert-Hash* [*simp*]: $keysFor\ (insert\ (Hash\ X)\ H) = keysFor\ H$
by $(unfold\ keysFor\text{-}def,\ auto)$

lemma *keysFor-insert-MPair* [*simp*]: $keysFor\ (insert\ \langle X, Y\rangle\ H) = keysFor\ H$
by $(unfold\ keysFor\text{-}def,\ auto)$

lemma *keysFor-insert-Crypt* [*simp*]:
$keysFor\ (insert\ (Crypt\ K\ X)\ H) = insert\ (invKey\ K)\ (keysFor\ H)$
by $(unfold\ keysFor\text{-}def,\ auto)$

lemma *keysFor-image-Key* [*simp*]: $keysFor\ (Key`E) = \{\}$
by $(unfold\ keysFor\text{-}def,\ auto)$

lemma *Crypt-imp-invKey-keysFor*: $Crypt\ K\ X \in H \Longrightarrow invKey\ K \in keysFor\ H$
by $(unfold\ keysFor\text{-}def,\ blast)$

1.4.2 Inductive relation "parts"

lemma *MPair-parts*:
$[\![$
$\langle X, Y\rangle \in parts\ H;$
$[\![\ X \in parts\ H;\ Y \in parts\ H\ ]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
by $(blast\ dest{:}\ parts.Fst\ parts.Snd)$

lemma *L-parts*:
$[\![$
$L\ l \in parts\ H;$
$[\![\ set\ l \subseteq parts\ H\ ]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
by $(blast\ dest{:}\ parts.Lst)$

declare *MPair-parts* [*elim!*] *L-parts* [*elim!*] *parts.Body* [*dest!*]

NB These two rules are UNSAFE in the formal sense, as they discard the compound message. They work well on THIS FILE. *MPair-parts* is left as SAFE because it speeds up proofs. The Crypt rule is normally kept UNSAFE to avoid breaking up certificates.

lemma *parts-increasing*: $H \subseteq parts\ H$
by *blast*

lemmas *parts-insertI* = *subset-insertI* [*THEN parts-mono*, *THEN subsetD*]

lemma *parts-empty* [*simp*]: $parts\{\} = \{\}$
apply *safe*
apply $(erule\ parts.induct,\ blast+)$
done

lemma *parts-emptyE* [*elim!*]: $X \in parts\{\} \Longrightarrow P$
by *simp*

WARNING: loops if H = Y, therefore must not be repeated!

lemma *parts-singleton*: $X \in parts\ H \Longrightarrow \exists\, Y \in H.\ X \in parts\ \{Y\}$
by (*erule parts.induct*, *fast*+)

lemma *parts-singleton-set*: $x \in parts\ \{s\ .\ P\ s\} \Longrightarrow \exists\, Y.\ P\ Y \wedge x \in parts\ \{Y\}$
by(*auto dest*: *parts-singleton*)

lemma *parts-singleton-set-rev*: $[\![x \in parts\ \{Y\};\ P\ Y]\!] \Longrightarrow x \in parts\ \{s\ .\ P\ s\}$
by (*induction rule*: *parts.induct*)
(*blast dest*: *parts.Fst parts.Snd parts.Lst parts.Body*)+

lemma *parts-Hash*: $[\![\bigwedge t\ .\ t \in H \Longrightarrow \exists\, t'\ .\ t = Hash\ t']\!] \Longrightarrow parts\ H = H$
by(*auto*, *erule parts.induct*, *blast*+)

Unions

lemma *parts-Un-subset1*: $parts\ G \cup parts\ H \subseteq parts(G \cup H)$
by (*intro Un-least parts-mono Un-upper1 Un-upper2*)

lemma *parts-Un-subset2*: $parts(G \cup H) \subseteq parts\ G \cup parts\ H$
apply (*rule subsetI*)
apply (*erule parts.induct*, *blast*+)
done

lemma *parts-Un* [*simp*]: $parts(G \cup H) = parts\ G \cup parts\ H$
by (*intro equalityI parts-Un-subset1 parts-Un-subset2*)

lemma *parts-insert*: $parts\ (insert\ X\ H) = parts\ \{X\} \cup parts\ H$
apply (*subst insert-is-Un* [*of - H*])
apply (*simp only*: *parts-Un*)
done

TWO inserts to avoid looping. This rewrite is better than nothing. Not suitable for Addsimps: its behaviour can be strange.

lemma *parts-insert2*:
$parts\ (insert\ X\ (insert\ Y\ H)) = parts\ \{X\} \cup parts\ \{Y\} \cup parts\ H$
apply (*simp add*: *Un-assoc*)
apply (*simp add*: *parts-insert* [*symmetric*])
done

lemma *parts-two*: $[\![x \in parts\ \{e1,\ e2\};\ x \notin parts\ \{e1\}]\!] \Longrightarrow x \in parts\ \{e2\}$
by (*simp add*: *parts-insert2*)

lemma *parts-UN-subset1*: $(\bigcup x{\in}A.\ parts(H\ x)) \subseteq parts(\bigcup x{\in}A.\ H\ x)$
by (*intro UN-least parts-mono UN-upper*)

lemma *parts-UN-subset2*: $parts(\bigcup x{\in}A.\ H\ x) \subseteq (\bigcup x{\in}A.\ parts(H\ x))$
apply (*rule subsetI*)
apply (*erule parts.induct*, *blast*+)
done

lemma *parts-UN* [*simp*]: $parts(\bigcup x{\in}A.\ H\ x) = (\bigcup x{\in}A.\ parts(H\ x))$
by (*intro equalityI parts-UN-subset1 parts-UN-subset2*)

Added to simplify arguments to parts, analz and synth. NOTE: the UN versions are no longer used!

This allows *blast* to simplify occurrences of *parts* $(G \cup H)$ in the assumption.

lemmas *in-parts-UnE* = *parts-Un* [*THEN equalityD1*, *THEN subsetD*, *THEN UnE*]
declare *in-parts-UnE* [*elim*!]

lemma *parts-insert-subset*: *insert X* (*parts H*) $\subseteq$ *parts*(*insert X H*)
by (*blast intro*: *parts-mono* [*THEN* [*2*] *rev-subsetD*])

Idempotence

lemma *parts-partsD* [*dest*!]: $X \in$ *parts* (*parts H*) $\Longrightarrow X \in$ *parts H*
by (*erule parts.induct*, *blast*+)

lemma *parts-idem* [*simp*]: *parts* (*parts H*) = *parts H*
by *blast*

lemma *parts-subset-iff* [*simp*]: (*parts G* $\subseteq$ *parts H*) = ($G \subseteq$ *parts H*)
apply (*rule iffI*)
apply (*iprover intro*: *subset-trans parts-increasing*)
apply (*frule parts-mono*, *simp*)
done

Transitivity

lemma *parts-trans*: $[\![$ $X \in$ *parts G*; $G \subseteq$ *parts H* $]\!] \Longrightarrow X \in$ *parts H*
by (*drule parts-mono*, *blast*)

Unions, revisited

You can take the union of parts h for all h in H

lemma *parts-split*: *parts H* = $\bigcup$ { *parts* {*h*} | *h* . $h \in H$}
apply *auto*
apply (*erule parts.induct*)
apply (*blast dest*: *parts.Fst parts.Snd parts.Lst parts.Body*)+
using *parts-trans* **apply** *blast*
done

Cut

lemma *parts-cut*:
  $[\![$ $Y \in$ *parts* (*insert X G*); $X \in$ *parts H* $]\!] \Longrightarrow Y \in$ *parts* ($G \cup H$)
by (*blast intro*: *parts-trans*)

lemma *parts-cut-eq* [*simp*]: $X \in$ *parts H* $\Longrightarrow$ *parts* (*insert X H*) = *parts H*
by (*force dest*!: *parts-cut intro*: *parts-insertI*)

Rewrite rules for pulling out atomic messages

lemmas *parts-insert-eq-I* = *equalityI* [*OF subsetI parts-insert-subset*]

lemma *parts-insert-AS* [*simp*]:
  parts (*insert* (*AS agt*) *H*) = *insert* (*AS agt*) (*parts H*)
apply (*rule parts-insert-eq-I*)
by (*erule parts.induct*, *auto*)

lemma *parts-insert-Epsilon* [*simp*]:
  parts (*insert* ε *H*) = *insert* ε (*parts H*)
apply (*rule parts-insert-eq-I*)
  by (*erule parts.induct*, *auto*)

lemma *parts-insert-Num* [*simp*]:
  parts (*insert* (*Num N*) *H*) = *insert* (*Num N*) (*parts H*)
apply (*rule parts-insert-eq-I*)
by (*erule parts.induct*, *auto*)

lemma *parts-insert-Key* [*simp*]:
  parts (*insert* (*Key K*) *H*) = *insert* (*Key K*) (*parts H*)
apply (*rule parts-insert-eq-I*)
by (*erule parts.induct*, *auto*)

lemma *parts-insert-Nonce* [*simp*]:
  parts (*insert* (*Nonce n*) *H*) = *insert* (*Nonce n*) (*parts H*)
apply (*rule parts-insert-eq-I*)
by (*erule parts.induct*, *auto*)

lemma *parts-insert-Hash* [*simp*]:
  parts (*insert* (*Hash X*) *H*) = *insert* (*Hash X*) (*parts H*)
apply (*rule parts-insert-eq-I*)
by (*erule parts.induct*, *auto*)

lemma *parts-insert-Crypt* [*simp*]:
  parts (*insert* (*Crypt K X*) *H*) = *insert* (*Crypt K X*) (*parts* (*insert X H*))
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule parts.induct*, *auto*)
by (*blast intro*: *parts.Body*)

lemma *parts-insert-MPair* [*simp*]:
  parts (*insert* $\langle X, Y\rangle$ *H*) =
    insert $\langle X, Y\rangle$ (*parts* (*insert X* (*insert Y H*)))
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule parts.induct*, *auto*)
by (*blast intro*: *parts.Fst parts.Snd*)+

lemma *parts-insert-L* [*simp*]:
  parts (*insert* (*L xs*) *H*) =
    insert (*L xs*) (*parts* ((*set xs*) $\cup$ *H*))
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule parts.induct*, *auto*)

by (*blast intro*: *parts.Lst*)+

lemma *parts-image-Key* [*simp*]: *parts* (*Key*'*N*) = *Key*'*N*
apply *auto*
apply (*erule parts.induct*, *auto*)
done

In any message, there is an upper bound N on its greatest nonce.

lemma *parts-list-set* :
  parts (*L*'*ls*) = (*L*'*ls*) ∪ ($\bigcup$ *l* ∈ *ls*. *parts* (*set l*))
apply (*rule equalityI*, *rule subsetI*)
apply (*erule parts.induct*, *auto*)
by (*meson L-parts image-subset-iff parts-increasing parts-trans*)

lemma *parts-insert-list-set* :
  parts ((*L*'*ls*) ∪ *H*) = (*L*'*ls*) ∪ ($\bigcup$ *l* ∈ *ls*. *parts* ((*set l*))) ∪ *parts H*
apply (*rule equalityI*, *rule subsetI*)
by (*erule parts.induct*, *auto simp add*: *parts-list-set*)

suffix of parts

lemma *suffix-in-parts*:
  suffix (*x*#*xs*) *ys* ⟹ *x* ∈ *parts* {*L ys*}
by (*auto simp add*: *suffix-def*)

lemma *parts-L-set*:
  ⟦*x* ∈ *parts* {*L ys*}; *ys* ∈ *St*⟧ ⟹ *x* ∈ *parts* (*L*'*St*)
by (*metis* (*no-types*, *lifting*) *image-insert insert-iff mk-disjoint-insert parts.Inj*
    parts-cut-eq parts-insert parts-insert2)

lemma *suffix-in-parts-set*:
  ⟦*suffix* (*x*#*xs*) *ys*; *ys* ∈ *St*⟧ ⟹ *x* ∈ *parts* (*L*'*St*)
using *parts-L-set suffix-in-parts*
by *blast*

1.4.3 Inductive relation "analz"

Inductive definition of "analz" – what can be broken down from a set of messages, including keys. A form of downward closure. Pairs can be taken apart; messages decrypted with known keys.

inductive-set
  analz :: *msgterm set* ⇒ *msgterm set*
  for *H* :: *msgterm set*
  where
    Inj [*intro*,*simp*] : *X* ∈ *H* ⟹ *X* ∈ *analz H*
  | *Fst*: ⟨*X*,*Y*⟩ ∈ *analz H* ⟹ *X* ∈ *analz H*
  | *Snd*: ⟨*X*,*Y*⟩ ∈ *analz H* ⟹ *Y* ∈ *analz H*
  | *Lst*: (*L y*) ∈ *analz H* ⟹ *x* ∈ *set* (*y*) ⟹ *x* ∈ *analz H*
  | *Decrypt* [*dest*]: ⟦ *Crypt K X* ∈ *analz H*; *Key* (*invKey K*) ∈ *analz H* ⟧ ⟹ *X* ∈ *analz H*

Monotonicity; Lemma 1 of Lowe's paper

lemma *analz-mono*: *G* ⊆ *H* ⟹ *analz*(*G*) ⊆ *analz*(*H*)

apply *auto*
apply (*erule analz.induct*)
apply (*auto dest*: *analz.Fst analz.Snd analz.Lst*)
done

lemmas *analz-monotonic* = *analz-mono* [*THEN* [*2*] *rev-subsetD*]

Making it safe speeds up proofs

lemma *MPair-analz* [*elim*!]:
 $\llbracket$
  $\langle X, Y\rangle \in$ *analz* H;
  $\llbracket$ $X \in$ *analz* H; $Y \in$ *analz* H $\rrbracket \Longrightarrow P$
 $\rrbracket \Longrightarrow P$
by (*blast dest*: *analz.Fst analz.Snd*)

lemma *L-analz* [*elim*!]:
 $\llbracket$
  $L\ l \in$ *analz* H;
  $\llbracket$ *set* $l \subseteq$ *analz* H $\rrbracket \Longrightarrow P$
 $\rrbracket \Longrightarrow P$
by (*blast dest*: *analz.Lst*)

lemma *analz-increasing*: $H \subseteq$ *analz*(H)
by *blast*

lemma *analz-subset-parts*: *analz* $H \subseteq$ *parts* H
apply (*rule subsetI*)
apply (*erule analz.induct*, *blast*+)
done

If there is no cryptography, then analz and parts is equivalent.

lemma *no-crypt-analz-is-parts*:
 $\neg$ ($\exists$ K X . *Crypt* K $X \in$ *parts* A) $\Longrightarrow$ *analz* A = *parts* A
apply (*rule equalityI*, *simp add*: *analz-subset-parts*)
apply (*rule subsetI*)
by (*erule parts.induct*, *blast*+, *simp*)

lemmas *analz-into-parts* = *analz-subset-parts* [*THEN subsetD*]

lemmas *not-parts-not-analz* = *analz-subset-parts* [*THEN contra-subsetD*]

lemma *parts-analz* [*simp*]: *parts* (*analz* H) = *parts* H
apply (*rule equalityI*)
apply (*rule analz-subset-parts* [*THEN parts-mono*, *THEN subset-trans*], *simp*)
apply (*blast intro*: *analz-increasing* [*THEN parts-mono*, *THEN subsetD*])
done

lemma *analz-parts* [*simp*]: *analz* (*parts* H) = *parts* H
apply *auto*
apply (*erule analz.induct*, *auto*)
done

lemmas *analz-insertI* = *subset-insertI* [*THEN analz-mono*, *THEN* [*2*] *rev-subsetD*]

General equational properties

lemma *analz-empty* [*simp*]: *analz* {} = {}
apply *safe*
apply (*erule analz.induct*, *blast*+)
done

Converse fails: we can analz more from the union than from the separate parts, as a key in one might decrypt a message in the other

lemma *analz-Un*: *analz*(G) ∪ *analz*(H) ⊆ *analz*(G ∪ H)
by (*intro Un-least analz-mono Un-upper1 Un-upper2*)

lemma *analz-insert*: *insert* X (*analz* H) ⊆ *analz*(*insert* X H)
by (*blast intro*: *analz-mono* [*THEN* [*2*] *rev-subsetD*])

Rewrite rules for pulling out atomic messages

lemmas *analz-insert-eq-I* = *equalityI* [*OF subsetI analz-insert*]

lemma *analz-insert-AS* [*simp*]:
  analz (*insert* (*AS agt*) H) = *insert* (*AS agt*) (*analz* H)
apply (*rule analz-insert-eq-I*)
by (*erule analz.induct*, *auto*)

lemma *analz-insert-Num* [*simp*]:
  analz (*insert* (*Num* N) H) = *insert* (*Num* N) (*analz* H)
apply (*rule analz-insert-eq-I*)
by (*erule analz.induct*, *auto*)

Can only pull out Keys if they are not needed to decrypt the rest

lemma *analz-insert-Key* [*simp*]:
  K ∉ *keysFor* (*analz* H) ⟹
    analz (*insert* (*Key* K) H) = *insert* (*Key* K) (*analz* H)
apply (*unfold keysFor-def*)
apply (*rule analz-insert-eq-I*)
by (*erule analz.induct*, *auto*)

lemma *analz-insert-LEmpty* [*simp*]:
  analz (*insert* (*L* []) H) = *insert* (*L* []) (*analz* H)
apply (*rule analz-insert-eq-I*)
by (*erule analz.induct*, *auto*)

lemma *analz-insert-L* [*simp*]:
  analz (*insert* (*L l*) H) = *insert* (*L l*) (*analz* (*set l* ∪ H))
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule analz.induct*, *auto*)
apply (*erule analz.induct*, *auto*)
using *analz.Inj* **by** *blast*

lemma $L[] \in analz\ \{L[L[]]\}$
using *analz.Inj* **by** *simp*

lemma *analz-insert-Hash* [*simp*]:
  $analz\ (insert\ (Hash\ X)\ H) = insert\ (Hash\ X)\ (analz\ H)$
apply (*rule analz-insert-eq-I*)
by (*erule analz.induct*, *auto*)

lemma *analz-insert-MPair* [*simp*]:
  $analz\ (insert\ \langle X,Y\rangle\ H) =$
    $insert\ \langle X,Y\rangle\ (analz\ (insert\ X\ (insert\ Y\ H)))$
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule analz.induct*, *auto*)
apply (*erule analz.induct*, *auto*)
using *Fst Snd analz.Inj insertI1*
by (*metis*)+

Can pull out enCrypted message if the Key is not known

lemma *analz-insert-Crypt*:
  $Key\ (invKey\ K) \notin analz\ H$
    $\Longrightarrow analz\ (insert\ (Crypt\ K\ X)\ H) = insert\ (Crypt\ K\ X)\ (analz\ H)$
apply (*rule analz-insert-eq-I*)
by (*erule analz.induct*, *auto*)

lemma *lemma1*:
  $Key\ (invKey\ K) \in analz\ H \Longrightarrow$
    $analz\ (insert\ (Crypt\ K\ X)\ H) \subseteq$
    $insert\ (Crypt\ K\ X)\ (analz\ (insert\ X\ H))$
apply (*rule subsetI*)
by (*erule-tac* $x = x$ **in** *analz.induct*, *auto*)

lemma *lemma2*:
  $Key\ (invKey\ K) \in analz\ H \Longrightarrow$
    $insert\ (Crypt\ K\ X)\ (analz\ (insert\ X\ H)) \subseteq$
    $analz\ (insert\ (Crypt\ K\ X)\ H)$
apply *auto*
apply (*erule-tac* $x = x$ **in** *analz.induct*, *auto*)
by (*blast intro*: *analz-insertI analz.Decrypt*)

lemma *analz-insert-Decrypt*:
  $Key\ (invKey\ K) \in analz\ H \Longrightarrow$
    $analz\ (insert\ (Crypt\ K\ X)\ H) =$
    $insert\ (Crypt\ K\ X)\ (analz\ (insert\ X\ H))$
by (*intro equalityI lemma1 lemma2*)

Case analysis: either the message is secure, or it is not! Effective, but can cause subgoals to blow up! Use with *split-if*; apparently *split-tac* does not cope with patterns such as $analz\ (insert\ (Crypt\ K\ X)\ H)$

lemma *analz-Crypt-if* [*simp*]:
  $analz\ (insert\ (Crypt\ K\ X)\ H) =$

(*if* (*Key* (*invKey K*) ∈ *analz H*)
 then insert (*Crypt K X*) (*analz* (*insert X H*))
 else insert (*Crypt K X*) (*analz H*))
by (*simp add*: *analz-insert-Crypt analz-insert-Decrypt*)

This rule supposes "for the sake of argument" that we have the key.

lemma *analz-insert-Crypt-subset*:
 analz (*insert* (*Crypt K X*) *H*) ⊆
 insert (*Crypt K X*) (*analz* (*insert X H*))
apply (*rule subsetI*)
by (*erule analz.induct*, *auto*)

lemma *analz-image-Key* [*simp*]: *analz* (*Key‘N*) = *Key‘N*
apply *auto*
apply (*erule analz.induct*, *auto*)
done

Idempotence and transitivity

lemma *analz-analzD* [*dest!*]: *X*∈ *analz* (*analz H*) ⟹ *X*∈ *analz H*
by (*erule analz.induct*, *blast+*)

lemma *analz-idem* [*simp*]: *analz* (*analz H*) = *analz H*
by *blast*

lemma *analz-subset-iff* [*simp*]: (*analz G* ⊆ *analz H*) = (*G* ⊆ *analz H*)
apply (*rule iffI*)
apply (*iprover intro*: *subset-trans analz-increasing*)
apply (*frule analz-mono*, *simp*)
done

lemma *analz-trans*: ⟦ *X*∈ *analz G*; *G* ⊆ *analz H* ⟧ ⟹ *X*∈ *analz H*
by (*drule analz-mono*, *blast*)

Cut; Lemma 2 of Lowe

lemma *analz-cut*: ⟦ *Y*∈ *analz* (*insert X H*); *X*∈ *analz H* ⟧ ⟹ *Y*∈ *analz H*
by (*erule analz-trans*, *blast*)

This rewrite rule helps in the simplification of messages that involve the forwarding of unknown components (X). Without it, removing occurrences of X can be very complicated.

lemma *analz-insert-eq*: *X*∈ *analz H* ⟹ *analz* (*insert X H*) = *analz H*
by (*blast intro*: *analz-cut analz-insertI*)

A congruence rule for "analz"

lemma *analz-subset-cong*:
 ⟦ *analz G* ⊆ *analz G′*; *analz H* ⊆ *analz H′* ⟧
 ⟹ *analz* (*G* ∪ *H*) ⊆ *analz* (*G′* ∪ *H′*)
apply *simp*
apply (*iprover intro*: *conjI subset-trans analz-mono Un-upper1 Un-upper2*)
done

lemma *analz-cong*:

$\llbracket$ *analz* G = *analz* G'; *analz* H = *analz* H' $\rrbracket$
  $\Longrightarrow$ *analz* $(G \cup H)$ = *analz* $(G' \cup H')$
by (*intro equalityI analz-subset-cong*, *simp-all*)

lemma *analz-insert-cong*:
 analz H = *analz* H' $\Longrightarrow$ *analz*(*insert* X H) = *analz*(*insert* X H')
by (*force simp only*: *insert-def intro*!: *analz-cong*)

If there are no pairs, lists or encryptions then analz does nothing

lemma *analz-trivial*:
 $\llbracket$
  $\forall X\ Y.\ \langle X,Y\rangle \notin H$; $\forall xs.\ L\ xs \notin H$;
  $\forall X\ K.\ Crypt\ K\ X \notin H$
 $\rrbracket \Longrightarrow$ *analz* $H = H$
apply *safe*
by (*erule analz.induct*, *auto*)

These two are obsolete (with a single Spy) but cost little to prove...

lemma *analz-UN-analz-lemma*:
 $X \in$ *analz* $(\bigcup i{\in}A.$ *analz* $(H\ i)) \Longrightarrow X \in$ *analz* $(\bigcup i{\in}A.\ H\ i)$
apply (*erule analz.induct*)
by (*blast intro*: *analz-mono* [*THEN* [*2*] *rev-subsetD*])+

lemma *analz-UN-analz* [*simp*]: *analz* $(\bigcup i{\in}A.$ *analz* $(H\ i))$ = *analz* $(\bigcup i{\in}A.\ H\ i)$
by (*blast intro*: *analz-UN-analz-lemma analz-mono* [*THEN* [*2*] *rev-subsetD*])

Lemmas assuming absense of keys

If there are no keys in analz H, you can take the union of analz h for all h in H

lemma *analz-split*:
 $\neg(\exists\ K\ .\ Key\ K \in analz\ H)$
  $\Longrightarrow$ *analz* $H = \bigcup\ \{\ analz\ \{h\}\ |\ h\ .\ h \in H\}$
apply *auto*
subgoal
 apply (*erule analz.induct*)
 apply (*blast dest*: *analz.Fst analz.Snd analz.Lst*)+
done
 apply (*erule analz.induct*)
 apply (*blast dest*: *analz.Fst analz.Snd analz.Lst*)+
 done

lemma *analz-Un-eq*:
 assumes $\neg(\exists\ K\ .\ Key\ K \in analz\ H)$ **and** $\neg(\exists\ K\ .\ Key\ K \in analz\ G)$
 shows *analz* $(H \cup G)$ = *analz* $H \cup$ *analz* G
apply (*intro equalityI*, *rule subsetI*)
apply (*erule analz.induct*)
using *assms* **by** *auto*

lemma *analz-Un-eq-Crypt*:
 assumes $\neg(\exists\ K\ .\ Key\ K \in analz\ G)$ **and** $\neg(\exists\ K\ X\ .\ Crypt\ K\ X \in analz\ G)$
 shows *analz* $(H \cup G)$ = *analz* $H \cup$ *analz* G
apply (*intro equalityI*, *rule subsetI*)

apply (*erule analz.induct*)
using *assms* **by** *auto*

lemma *analz-list-set* :
  $\neg(\exists\ K\ .\ Key\ K \in analz\ (L`ls))$
    $\Longrightarrow analz\ (L`ls) =\ (L`ls) \cup (\bigcup l \in ls.\ analz\ (set\ l))$
apply (*rule equalityI*, *rule subsetI*)
apply (*erule analz.induct*, *auto*)
using *L-analz image-subset-iff analz-increasing analz-trans* **by** *metis*

lemma *analz-insert-list-set* :
  $\neg(\exists\ K\ .\ Key\ K \in analz\ ((L`ls) \cup H))$
    $\Longrightarrow analz\ ((L`ls) \cup H) =\ (L`ls) \cup (\bigcup l \in ls.\ analz\ ((set\ l))) \cup analz\ H$
apply (*rule equalityI*, *rule subsetI*)
apply (*erule analz.induct*, *auto*)
by (*smt L-analz Set.set-insert analz-increasing analz-trans image-insert insert-subset*
    sup.bounded-iff)

1.4.4 Inductive relation "synth"

Inductive definition of "synth" – what can be built up from a set of messages. A form of upward closure. Pairs can be built, messages encrypted with known keys. AS names are public domain. Nums can be guessed, but Nonces cannot be.

inductive-set
  $synth :: msgterm\ set \Rightarrow msgterm\ set$
  for $H :: msgterm\ set$
  where
    Inj [*intro*]: $X \in H \Longrightarrow X \in synth\ H$
  | ε [*simp*,*intro*!]: $\varepsilon \in synth\ H$
  | *AS* [*simp*,*intro*!]: $AS\ agt \in synth\ H$
  | *Num* [*simp*,*intro*!]: $Num\ n\ \in synth\ H$
  | *Lst* [*intro*]: $[\![\ \bigwedge x\ .\ x \in set\ xs \Longrightarrow x \in synth\ H\ ]\!] \Longrightarrow L\ xs \in synth\ H$
  | *Hash* [*intro*]: $X \in synth\ H \Longrightarrow Hash\ X \in synth\ H$
  | *MPair* [*intro*]: $[\![\ X \in synth\ H;\ \ Y \in synth\ H\ ]\!] \Longrightarrow \langle X, Y\rangle \in synth\ H$
  | *Crypt* [*intro*]: $[\![\ X \in synth\ H;\ \ Key\ K \in H\ ]\!] \Longrightarrow Crypt\ K\ X \in synth\ H$

Monotonicity

lemma *synth-mono*: $G \subseteq H \Longrightarrow synth(G) \subseteq synth(H)$
by (*auto*, *erule synth.induct*, *auto*)

NO *AS-synth*, as any AS name can be synthesized. The same holds for *Num*

inductive-cases *Key-synth* [*elim*!]: $Key\ K \in synth\ H$
inductive-cases *Nonce-synth* [*elim*!]: $Nonce\ n \in synth\ H$
inductive-cases *Hash-synth* [*elim*!]: $Hash\ X \in synth\ H$
inductive-cases *MPair-synth* [*elim*!]: $\langle X, Y\rangle \in synth\ H$
inductive-cases *L-synth* [*elim*!]: $L\ X \in synth\ H$
inductive-cases *Crypt-synth* [*elim*!]: $Crypt\ K\ X \in synth\ H$

lemma *synth-increasing*: $H \subseteq synth(H)$
by *blast*

lemma *synth-analz-self*: $x \in H \Longrightarrow x \in synth\ (analz\ H)$
  by *blast*

Unions

Converse fails: we can synth more from the union than from the separate parts, building a compound message using elements of each.

lemma *synth-Un*: $synth(G) \cup synth(H) \subseteq synth(G \cup H)$
by (*intro Un-least synth-mono Un-upper1 Un-upper2*)

lemma *synth-insert*: $insert\ X\ (synth\ H) \subseteq synth(insert\ X\ H)$
by (*blast intro*: *synth-mono* [*THEN* [*2*] *rev-subsetD*])

Idempotence and transitivity

lemma *synth-synthD* [*dest!*]: $X\in synth\ (synth\ H) \Longrightarrow X \in synth\ H$
apply (*erule synth.induct*, *blast*)
apply *auto*
done

lemma *synth-idem*: $synth\ (synth\ H) = synth\ H$
by *blast*

lemma *synth-subset-iff* [*simp*]: $(synth\ G \subseteq synth\ H) = (G \subseteq synth\ H)$
apply (*rule iffI*)
apply (*iprover intro*: *subset-trans synth-increasing*)
apply (*frule synth-mono*, *simp add*: *synth-idem*)
done

lemma *synth-trans*: $[\![\ X\in synth\ G;\ \ G \subseteq synth\ H\ ]\!] \Longrightarrow X\in synth\ H$
by (*drule synth-mono*, *blast*)

Cut; Lemma 2 of Lowe

lemma *synth-cut*: $[\![\ Y\in synth\ (insert\ X\ H);\ \ X\in synth\ H\ ]\!] \Longrightarrow Y\in synth\ H$
by (*erule synth-trans*, *blast*)

lemma *Nonce-synth-eq* [*simp*]: $(Nonce\ N \in synth\ H) = (Nonce\ N \in H)$
  try
by *blast*

lemma *Key-synth-eq* [*simp*]: $(Key\ K \in synth\ H) = (Key\ K \in H)$
by *blast*

lemma *Crypt-synth-eq* [*simp*]:
  $Key\ K \notin H \Longrightarrow (Crypt\ K\ X \in synth\ H) = (Crypt\ K\ X \in H)$
by *blast*

lemma *keysFor-synth* [*simp*]:
  $keysFor\ (synth\ H) = keysFor\ H \cup invKey`\{K.\ Key\ K \in H\}$
by (*unfold keysFor-def*, *blast*)

lemma *L-cons-synth* [*simp*]:
  (*set xs* $\subseteq$ *H*) $\Longrightarrow$ (*L xs* $\in$ *synth H*)
by *auto*

Combinations of parts, analz and synth

lemma *parts-synth* [*simp*]: *parts* (*synth H*) = *parts H* $\cup$ *synth H*
proof (*safe del*: *UnCI*)
  fix *X*
  assume *X* $\in$ *parts* (*synth H*)
  thus *X* $\in$ *parts H* $\cup$ *synth H*
  by (*induct rule*: *parts.induct*)
    (*blast intro*: *parts.Fst parts.Snd parts.Lst parts.Body*)+
next
  fix *X*
  assume *X* $\in$ *parts H*
  thus *X* $\in$ *parts* (*synth H*)
  by (*induction rule*: *parts.induct*)
    (*blast intro*: *parts.Fst parts.Snd parts.Lst parts.Body*)+
next
  fix *X*
  assume *X* $\in$ *synth H*
  thus *X* $\in$ *parts* (*synth H*)
  by (*induction rule*: *synth.induct*)
    (*blast intro*: *parts.Fst parts.Snd parts.Lst parts.Body*)+
qed

lemma *analz-analz-Un* [*simp*]: *analz* (*analz G* $\cup$ *H*) = *analz* (*G* $\cup$ *H*)
apply (*intro equalityI analz-subset-cong*)+
apply *simp-all*
done

lemma *analz-synth-Un* [*simp*]: *analz* (*synth G* $\cup$ *H*) = *analz* (*G* $\cup$ *H*) $\cup$ *synth G*
proof (*safe del*: *UnCI*)
  fix *X*
  assume *X* $\in$ *analz* (*synth G* $\cup$ *H*)
  thus *X* $\in$ *analz* (*G* $\cup$ *H*) $\cup$ *synth G*
  by (*induction rule*: *analz.induct*)
    (*blast intro*: *analz.Fst analz.Snd analz.Lst analz.Decrypt*)+
qed (*auto elim*: *analz-mono* [*THEN* [*2*] *rev-subsetD*])

lemma *analz-synth* [*simp*]: *analz* (*synth H*) = *analz H* $\cup$ *synth H*
apply (*cut-tac H* = {} **in** *analz-synth-Un*)
apply (*simp* (*no-asm-use*))
done

chsp: added

lemma *analz-Un-analz* [*simp*]: *analz* (*G* $\cup$ *analz H*) = *analz* (*G* $\cup$ *H*)
by (*subst Un-commute*, *auto*)+

lemma *analz-synth-Un2* [*simp*]: *analz* (*G* $\cup$ *synth H*) = *analz* (*G* $\cup$ *H*) $\cup$ *synth H*
by (*subst Un-commute*, *auto*)+

For reasoning about the Fake rule in traces

lemma *parts-insert-subset-Un*: $X \in G \Longrightarrow parts(insert\ X\ H) \subseteq parts\ G \cup parts\ H$
by (*rule subset-trans* [*OF parts-mono parts-Un-subset2*], *blast*)

More specifically for Fake. Very occasionally we could do with a version of the form *parts* $\{X\} \subseteq synth\ (analz\ H) \cup parts\ H$

lemma *Fake-parts-insert*:
  $X \in synth\ (analz\ H) \Longrightarrow$
    $parts\ (insert\ X\ H) \subseteq synth\ (analz\ H) \cup parts\ H$
apply (*drule parts-insert-subset-Un*)
apply (*simp* (*no-asm-use*))
apply *blast*
done

lemma *Fake-parts-insert-in-Un*:
  $[\![Z \in parts\ (insert\ X\ H);\ \ X \in synth\ (analz\ H)]\!]$
    $\Longrightarrow Z \in\ synth\ (analz\ H) \cup parts\ H$
by (*blast dest*: *Fake-parts-insert* [*THEN subsetD*, *dest*])

H is sometimes $Key\ `\ KK \cup spies\ evs$, so can't put $G = H$.

lemma *Fake-analz-insert*:
  $X \in synth\ (analz\ G) \Longrightarrow$
    $analz\ (insert\ X\ H) \subseteq synth\ (analz\ G) \cup analz\ (G \cup H)$
apply (*rule subsetI*)
apply (*subgoal-tac* $x \in analz\ (synth\ (analz\ G) \cup H)$)
prefer *2*
  apply (*blast intro*: *analz-mono* [*THEN* [*2*] *rev-subsetD*]
                    analz-mono [*THEN synth-mono*, *THEN* [*2*] *rev-subsetD*])
apply (*simp* (*no-asm-use*))
apply *blast*
done

lemma *analz-conj-parts* [*simp*]:
  $(X \in analz\ H\ \&\ X \in parts\ H) = (X \in analz\ H)$
by (*blast intro*: *analz-subset-parts* [*THEN subsetD*])

lemma *analz-disj-parts* [*simp*]:
  $(X \in analz\ H\ |\ X \in parts\ H) = (X \in parts\ H)$
by (*blast intro*: *analz-subset-parts* [*THEN subsetD*])

Without this equation, other rules for synth and analz would yield redundant cases

lemma *MPair-synth-analz* [*iff*]:
  $(\langle X,Y\rangle \in synth\ (analz\ H)) =$
    $(X \in synth\ (analz\ H)\ \&\ Y \in synth\ (analz\ H))$
by *blast*

lemma *L-cons-synth-analz* [*iff*]:
  $(L\ xs \in synth\ (analz\ H)) =$
    $(set\ xs \subseteq synth\ (analz\ H))$
by *blast*

lemma *L-cons-synth-parts* [*iff*]:
  $(L\ xs \in synth\ (parts\ H)) =$
    $(set\ xs \subseteq synth\ (parts\ H))$
by *blast*

lemma *Crypt-synth-analz*:
  $[\![\ Key\ K \in analz\ H;\ \ Key\ (invKey\ K) \in analz\ H\ ]\!]$
    $\Longrightarrow (Crypt\ K\ X \in synth\ (analz\ H)) = (X \in synth\ (analz\ H))$
by *blast*

lemma *Hash-synth-analz* [*simp*]:
  $X \notin synth\ (analz\ H)$
    $\Longrightarrow (Hash\langle X,Y\rangle \in synth\ (analz\ H)) = (Hash\langle X,Y\rangle \in analz\ H)$
by *blast*

1.4.5 HPair: a combination of Hash and MPair

We do NOT want Crypt... messages broken up in protocols!!

declare *parts.Body* [*rule del*]

Rewrites to push in Key and Crypt messages, so that other messages can be pulled out using the *analz-insert* rules

lemmas *pushKeys* =
  insert-commute [*of Key K AS C* **for** *K C*]
  insert-commute [*of Key K Nonce N* **for** *K N*]
  insert-commute [*of Key K Num N* **for** *K N*]
  insert-commute [*of Key K Hash X* **for** *K X*]
  insert-commute [*of Key K MPair X Y* **for** *K X Y*]
  insert-commute [*of Key K Crypt X K'* **for** *K K' X*]

lemmas *pushCrypts* =
  insert-commute [*of Crypt X K AS C* **for** *X K C*]
  insert-commute [*of Crypt X K AS C* **for** *X K C*]
  insert-commute [*of Crypt X K Nonce N* **for** *X K N*]
  insert-commute [*of Crypt X K Num N* **for** *X K N*]
  insert-commute [*of Crypt X K Hash X'* **for** *X K X'*]
  insert-commute [*of Crypt X K MPair X' Y* **for** *X K X' Y*]

Cannot be added with [*simp*] – messages should not always be re-ordered.

lemmas *pushes* = *pushKeys pushCrypts*

By default only *o-apply* is built-in. But in the presence of eta-expansion this means that some terms displayed as $f \circ g$ will be rewritten, and others will not!

declare *o-def* [*simp*]

lemma *Crypt-notin-image-Key* [*simp*]: $Crypt\ K\ X \notin Key\ `\ A$
by *auto*

lemma *Hash-notin-image-Key* [*simp*] :$Hash\ X \notin Key\ `\ A$
by *auto*

lemma *synth-analz-mono*: $G \subseteq H \Longrightarrow$ *synth* (*analz*(G)) $\subseteq$ *synth* (*analz*(H))
by (*iprover intro*: *synth-mono analz-mono*)

lemma *synth-parts-mono*: $G \subseteq H \Longrightarrow$ *synth* (*parts* G) $\subseteq$ *synth* (*parts* H)
by (*iprover intro*: *synth-mono parts-mono*)

lemma *Fake-analz-eq* [*simp*]:
  $X \in$ *synth*(*analz* H) $\Longrightarrow$ *synth* (*analz* (*insert* X H)) = *synth* (*analz* H)
apply (*drule Fake-analz-insert*[*of - - H*])
apply (*simp add*: *synth-increasing*[*THEN Un-absorb2*])
apply (*drule synth-mono*)
apply (*simp add*: *synth-idem*)
apply (*rule equalityI*)
apply (*simp add*:)
apply (*rule synth-analz-mono*, *blast*)
done

Two generalizations of *analz-insert-eq*

lemma *gen-analz-insert-eq* [*rule-format*]:
  $X \in$ *analz* $H \Longrightarrow$ *ALL* G. $H \subseteq G$ $-->$ *analz* (*insert* X G) = *analz* G
by (*blast intro*: *analz-cut analz-insertI analz-mono* [*THEN* [*2*] *rev-subsetD*])

lemma *synth-analz-insert-eq* [*rule-format*]:
  $X \in$ *synth* (*analz* H)
    $\Longrightarrow$ *ALL* G. $H \subseteq G$ $-->$ (*Key* $K \in$ *analz* (*insert* X G)) = (*Key* $K \in$ *analz* G)
apply (*erule synth.induct*)
apply (*auto simp add*: *gen-analz-insert-eq subset-trans* [*OF - subset-insertI*])

oops

lemma *Fake-parts-sing*:
  $X \in$ *synth* (*analz* H) $\Longrightarrow$ *parts*$\{X\}$ $\subseteq$ *synth* (*analz* H) $\cup$ *parts* H
apply (*rule subset-trans*)
 apply (*erule-tac* [*2*] *Fake-parts-insert*)
apply (*rule parts-mono*, *blast*)
done

lemmas *Fake-parts-sing-imp-Un* = *Fake-parts-sing* [*THEN* [*2*] *rev-subsetD*]

For some reason, moving this up can make some proofs loop!

declare *invKey-K* [*simp*]

lemma *synth-analz-insert*:
  assumes *analz* $H \subseteq$ *synth* (*analz* H')
  shows *analz* (*insert* X H) $\subseteq$ *synth* (*analz* (*insert* X H'))
proof
  fix x
  assume $x \in$ *analz* (*insert* X H)
  then have $x \in$ *analz* (*insert* X (*synth* (*analz* H')))
    using *assms* **by** (*meson analz-increasing analz-monotonic insert-mono*)

then show $x \in synth\ (analz\ (insert\ X\ H'))$
by $(metis\ (no\text{-}types)\ Un\text{-}iff\ analz\text{-}idem\ analz\text{-}insert\ analz\text{-}monotonic\ analz\text{-}synth\ synth.Inj$
$synth\text{-}insert\ synth\text{-}mono)$
qed

lemma *synth-parts-insert*:
assumes $parts\ H \subseteq synth\ (parts\ H')$
shows $parts\ (insert\ X\ H) \subseteq synth\ (parts\ (insert\ X\ H'))$
proof
fix x
assume $x \in parts\ (insert\ X\ H)$
then have $x \in parts\ (insert\ X\ (synth\ (parts\ H')))$
using *assms parts-increasing*
by $(metis\ UnE\ UnI1\ analz\text{-}monotonic\ analz\text{-}parts\ parts\text{-}insert\ parts\text{-}insertI)$
then show $x \in synth\ (parts\ (insert\ X\ H'))$
using *Un-iff parts-idem parts-insert parts-synth synth.Inj*
by $(metis\ Un\text{-}subset\text{-}iff\ synth\text{-}increasing\ synth\text{-}trans)$
qed

lemma *parts-insert-subset-impl*:
$[\![x \in parts\ (insert\ a\ G);\ x \in parts\ G \Longrightarrow x \in synth\ (parts\ H);\ a \in synth\ (parts\ H)]\!]$
$\Longrightarrow x \in synth\ (parts\ H)$
using *Fake-parts-sing in-parts-UnE insert-is-Un*
parts-idem parts-synth subsetCE sup.absorb2 synth-idem synth-increasing
by $(metis\ (no\text{-}types,\ lifting)\ analz\text{-}parts)$

lemma *synth-parts-subset-elem*:
$[\![A \subseteq synth\ (parts\ B);\ x \in parts\ A]\!] \Longrightarrow x \in synth\ (parts\ B)$
by $(meson\ parts\text{-}emptyE\ parts\text{-}insert\text{-}subset\text{-}impl\ parts\text{-}singleton\ subset\text{-}iff)$

lemma *synth-parts-subset*:
$A \subseteq synth\ (parts\ B) \Longrightarrow parts\ A \subseteq synth\ (parts\ B)$
by $(auto\ simp\ add:\ synth\text{-}parts\text{-}subset\text{-}elem)$

lemma *parts-synth-parts*[*simp*]: $parts\ (synth\ (parts\ H)) = synth\ (parts\ H)$
by *auto*

lemma *synth-parts-trans*:
assumes $A \subseteq synth\ (parts\ B)$ **and** $B \subseteq synth\ (parts\ C)$
shows $A \subseteq synth\ (parts\ C)$
using *assms* **by** $(metis\ order\text{-}trans\ parts\text{-}synth\text{-}parts\ synth\text{-}idem\ synth\text{-}parts\text{-}mono)$

lemma *synth-parts-trans-elem*:
assumes $x \in A$ **and** $A \subseteq synth\ (parts\ B)$ **and** $B \subseteq synth\ (parts\ C)$
shows $x \in synth\ (parts\ C)$
using *synth-parts-trans assms* **by** *auto*

lemma *synth-un-parts-split*:
assumes $x \in synth\ (parts\ A \cup parts\ B)$
and $\bigwedge x\ .\ x \in A \Longrightarrow x \in synth\ (parts\ C)$
and $\bigwedge x\ .\ x \in B \Longrightarrow x \in synth\ (parts\ C)$

shows $x \in$ *synth* (*parts* C)
proof $-$
have *parts* $A \subseteq$ *synth* (*parts* C) *parts* $B \subseteq$ *synth* (*parts* C)
using *assms*(*2*) *assms*(*3*) *synth-parts-subset* **by** *blast*+
then have $x \in$ *synth* (*synth* (*parts* C) $\cup$ *synth* (*parts* C)) **using** *assms*(*1*)
using *synth-trans* **by** *auto*
then show *?thesis* **by** *auto*
qed
end

1.5 Tools

theory *Tools* **imports** *Main HOL−Library.Sublist*
begin

1.5.1 Prefixes, suffixes, and fragments

lemma *suffix-nonempty-extendable*:
  $[\![suffix\ xs\ l;\ xs \neq l]\!] \Longrightarrow \exists\ x\ .\ suffix\ (x\#xs)\ l$
apply (*auto simp add*: *suffix-def*)
  by (*metis append-butlast-last-id*)

lemma *set-suffix*:
  $[\![x \in set\ l';\ suffix\ l'\ l]\!] \Longrightarrow x \in set\ l$
by (*auto simp add*: *suffix-def*)

lemma *set-prefix*:
  $[\![x \in set\ l';\ prefix\ l'\ l]\!] \Longrightarrow x \in set\ l$
by (*auto simp add*: *prefix-def*)

lemma *set-suffix-elem*: $suffix\ (x\#xs)\ p \Longrightarrow x \in set\ p$
  by (*meson list.set-intros*(*1*) *set-suffix*)

lemma *set-prefix-elem*: $prefix\ (x\#xs)\ p \Longrightarrow x \in set\ p$
  by (*meson list.set-intros*(*1*) *set-prefix*)

lemma *Cons-suffix-set*: $x \in set\ y \Longrightarrow \exists\ xs\ .\ suffix\ (x\#xs)\ y$
  using *suffix-def* **by** (*metis split-list*)

1.5.2 Fragments

definition $fragment :: 'a\ list \Rightarrow 'a\ list\ set \Rightarrow bool$
  where $fragment\ xs\ St \longleftrightarrow (\exists\, zs1\ zs2.\ zs1\ @\ xs\ @\ zs2 \in St)$

lemma *fragmentI*: $[\![\ zs1\ @\ xs\ @\ zs2 \in St\ ]\!] \Longrightarrow fragment\ xs\ St$
by (*auto simp add*: *fragment-def*)

lemma *fragmentE* [*elim*]: $[\![fragment\ xs\ St;\ \bigwedge zs1\ zs2.\ [\![\ zs1\ @\ xs\ @\ zs2 \in St\ ]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
by (*auto simp add*: *fragment-def*)

lemma *fragment-Nil* [*simp*]: $fragment\ []\ St \longleftrightarrow St \neq \{\}$
by (*force simp add*: *fragment-def dest*: *spec* [**where** x=[]])

lemma *fragment-subset*: $[\![St \subseteq St';\ fragment\ l\ St]\!] \Longrightarrow fragment\ l\ St'$
by(*auto simp add*: *fragment-def*)

lemma *fragment-prefix*: $[\![prefix\ l'\ l;\ fragment\ l\ St]\!] \Longrightarrow fragment\ l'\ St$
by(*auto simp add*: *fragment-def prefix-def*) *blast*

lemma *fragment-suffix*: $[\![suffix\ l'\ l;\ fragment\ l\ St]\!] \Longrightarrow fragment\ l'\ St$
by(*auto simp add*: *fragment-def suffix-def*)
  (*metis append.assoc*)

lemma *fragment-self* [*simp*, *intro*]: $[\![l \in St]\!] \Longrightarrow$ *fragment l St*
by(*auto simp add*: *fragment-def intro*!: *exI* [**where** x=[]])

lemma *fragment-prefix-self* [*simp*, *intro*]:
  $[\![l \in St;$ *prefix* l' $l]\!] \Longrightarrow$ *fragment* l' *St*
using *fragment-prefix fragment-self* **by** *blast*

lemma *fragment-suffix-self* [*simp*, *intro*]:
  $[\![l \in St;$ *suffix* l' $l]\!] \Longrightarrow$ *fragment* l' *St*
using *fragment-suffix fragment-self* **by** *metis*

lemma *fragment-is-prefix-suffix*:
  fragment l St $\Longrightarrow \exists$ *pre suff* . *prefix l pre* $\wedge$ *suffix pre suff* $\wedge$ *suff* $\in$ *St*
  by (*meson fragment-def prefixI suffixI*)

1.5.3 Pair Fragments

definition *pfragment* :: $'a \Rightarrow$ ($'b$ *list*) $\Rightarrow$ ($'a \times$ ($'b$ *list*)) *set* $\Rightarrow$ *bool*
  where *pfragment a xs St* $\longleftrightarrow$ ($\exists$ *zs1 zs2*. (*a*, *zs1* @ *xs* @ *zs2*) $\in$ *St*)

lemma *pfragmentI*: $[\![$ (*ainf*, *zs1* @ *xs* @ *zs2*) $\in$ *St* $]\!] \Longrightarrow$ *pfragment ainf xs St*
by (*auto simp add*: *pfragment-def*)

lemma *pfragmentE* [*elim*]: $[\![$*pfragment ainf xs St*; $\bigwedge$*zs1 zs2*. $[\![$ (*ainf*, *zs1* @ *xs* @ *zs2*) $\in$ *St* $]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
by (*auto simp add*: *pfragment-def*)

lemma *pfragment-prefix*:
  pfragment ainf (*xs* @ *ys*) *St* $\Longrightarrow$ *pfragment ainf xs St*
  by(*auto simp add*: *pfragment-def*)

lemma *pfragment-prefix*$'$:
  $[\![$*pfragment ainf ys St*; *prefix xs ys*$]\!] \Longrightarrow$ *pfragment ainf xs St*
  by(*auto 3 4 simp add*: *pfragment-def prefix-def*)

lemma *pfragment-suffix*: $[\![$*suffix* l' l; *pfragment ainf l St*$]\!] \Longrightarrow$ *pfragment ainf* l' *St*
  by(*auto simp add*: *pfragment-def suffix-def*)
  (*metis append.assoc*)

lemma *pfragment-self* [*simp*, *intro*]: $[\![$(*ainf*, *l*) $\in$ *St*$]\!] \Longrightarrow$ *pfragment ainf l St*
by(*auto simp add*: *pfragment-def intro*!: *exI* [**where** x=[]])

lemma *pfragment-suffix-self* [*simp*, *intro*]:
  $[\![$(*ainf*, *l*) $\in$ *St*; *suffix* l' $l]\!] \Longrightarrow$ *pfragment ainf* l' *St*
using *pfragment-suffix pfragment-self* **by** *metis*

lemma *pfragment-self-eq*:
$[\![$*pfragment ainf l S*; $\bigwedge$*zs1 zs2* . (*ainf*, *zs1*@*l*@*zs2*) $\in$ *S* $\Longrightarrow$ (*ainf*, *zs1*@l'@*zs2*) $\in$ *S*$]\!] \Longrightarrow$ *pfragment ainf* l' *S*
  by(*auto simp add*: *pfragment-def*)

lemma *pfragment-self-eq-nil*:

$[\![$*pfragment ainf l S*; $\bigwedge$*zs1 zs2* . (*ainf*, *zs1*@*l*@*zs2*) $\in S \Longrightarrow$ (*ainf*, *l'*@*zs2*) $\in S]\!] \Longrightarrow$ *pfragment ainf l' S*
  apply(*auto simp add*: *pfragment-def*)
  apply(*rule exI*[*of* - []])
  by *auto*

lemma *pfragment-cons*: *pfragment ainfo* (*x* # *fut*) *S* $\Longrightarrow$ *pfragment ainfo fut S*
  apply(*auto 3 4 simp add*: *pfragment-def*)
  subgoal for *zs1 zs2*
  apply(*rule exI*[*of* - *zs1*@[*x*]])
    by *auto*
  done

1.5.4 Head and Tails

fun *head* **where** *head* [] = *None* | *head* (*x*#*xs*) = *Some x*
fun *ifhead* **where** *ifhead* [] *n* = *n* | *ifhead* (*x*#*xs*) - = *Some x*
fun *tail* **where** *tail* [] = *None* | *tail xs* = *Some* (*last xs*)

lemma *head-cons*: *xs* $\neq$ [] $\Longrightarrow$ *head xs* = *Some* (*hd xs*) **by**(*cases xs*, *auto*)
lemma *tail-cons*: *xs* $\neq$ [] $\Longrightarrow$ *tail xs* = *Some* (*last xs*) **by**(*cases xs*, *auto*)
lemma *tail-snoc*: *tail* (*xs* @ [*x*]) = *Some x* **by**(*cases xs*, *auto*)
lemma $\forall$ *y ys* . *l* $\neq$ *ys* @ [*y*] $\Longrightarrow$ *l* = []
  using *rev-exhaust* **by** *blast*

lemma *tl-append2*: *tl* (*pref* @ [*a*, *b*]) = *tl* (*pref* @ [*a*])@[*b*]
  by(*induction pref*, *auto*)

end

theory *TakeWhile* **imports** *Tools*
begin

1.6 takeW, holds and extract: Applying context-sensitive checks on list elements

This theory defines three functions, takeW, holds and extract. It is embedded in a locale that takes predicate P as an input that works on three arguments: pre, x, and z. x is an element of a list, while pre is the left neighbour on that list and z is the right neighbour. They are all of the same type 'a, except that pre and z are of 'a option type, since neighbours don't always exist at the beginning and the end of lists. The functions takeW and holds work on an 'a list (with an additional pre and z 'a option parameter). Both repeatedly apply P on elements xi in the list with their neighbours as context:

```
holds pre (x1#x2#...#xn#[]) z =
    P pre x1 x2 /\ P x1 x2 x3 /\ ... /\ P (xn-2) (xn-1) xn /\ P xn-1 xn z
takeW pre (x1#x2#...#xn#[]) z = the prefix of the list for which 'holds' holds.
```

extract is a function that returns the last element of the list, or z if the list is empty.

holds-takeW-extract is an interesting lemma that relates all three functions.

In our applications, we usually invoke takeW and holds with the parameters None l None, where l is a list of elements which we want to check for P (using their neighboring elements as context). takeW and holds thus mostly have the pre and z parameters for their recursive definition and induction schemes.

locale *TW* =
  fixes *P* :: (*'a option* ⇒ *'a* ⇒ *'a option* ⇒ *bool*)
begin

1.6.1 Definitions

holds returns true iff every element of a list, together with its context, satisfies P.

fun *holds* :: *'a option* ⇒ *'a list* ⇒ *'a option* ⇒ *bool*
where
  holds pre (*x* # *y* # *ys*) *nxt* ⟷ *P pre x* (*Some y*) ∧ *holds* (*Some x*) (*y* # *ys*) *nxt*
| *holds pre* [*x*] *nxt* ⟷ *P pre x nxt*
| *holds pre* [] *nxt* ⟷ *True*

holds returns the longest prefix of a list for every element, together with its context, satisfies P.

function *takeW* :: *'a option* ⇒ *'a list* ⇒ *'a option* ⇒ *'a list* **where**
  takeW - [] - = []
| *P pre x xo* ⟹ *takeW pre* [*x*] *xo* = [*x*]
| ¬ *P pre x xo* ⟹ *takeW pre* [*x*] *xo* = []
| *P pre x* (*Some y*) ⟹ *takeW pre* (*x* # *y* # *xs*) *xo* = *x* # *takeW* (*Some x*) (*y* # *xs*) *xo*
| ¬ *P pre x* (*Some y*) ⟹ *takeW pre* (*x* # *y* # *xs*) *xo* = []
apply *auto*
  by (*metis remdups-adj.cases*)
termination
  by *lexicographic-order*

extract returns the last element of a list, or nxt if the list is empty.

fun *extract* :: *'a option* ⇒ *'a list* ⇒ *'a option* ⇒ *'a option*
where
  extract pre (*x* # *y* # *ys*) *nxt* = (*if P pre x* (*Some y*) *then extract* (*Some x*) (*y* # *ys*) *nxt else Some x*)
| *extract pre* [*x*] *nxt* = (*if P pre x nxt then nxt else* (*Some x*))
| *extract pre* [] *nxt* = *nxt*

1.6.2 Lemmas

Lemmas packing singleton and at least two element cases into a single equation.

lemma *takeW-singleton*:
  takeW pre [*x*] *xo* = (*if P pre x xo then* [*x*] *else* [])
by (*simp*)

lemma *takeW-two-or-more*:
  takeW pre (*x* # *y* # *zs*) *xo* = (*if P pre x* (*Some y*) *then x* # *takeW* (*Some x*) (*y* # *zs*) *xo else* [])
by (*simp*)

Some lemmas for splitting the tail of the list argument.

Splitting lemma formulated with if-then-else rather than case.

lemma *takeW-split-tail*:
  takeW pre (*x* # *xs*) *nxt* =
    (*if xs* = []
     then (*if P pre x nxt then* [*x*] *else* [])
     else (*if P pre x* (*Some* (*hd xs*)) *then x* # *takeW* (*Some x*) *xs nxt else* []))
by (*cases xs*, *auto*)

lemma *extract-split-tail*:
  extract pre (*x* # *xs*) *nxt* =
    (*case xs of*
        [] ⇒ (*if P pre x nxt then nxt else* (*Some x*))
      | (*y* # *ys*) ⇒ (*if P pre x* (*Some y*) *then extract* (*Some x*) (*y* # *ys*) *nxt else Some x*))
by (*cases xs*, *auto*)

lemma *holds-split-tail*:
  holds pre (*x* # *xs*) *nxt* ⟷
     (*case xs of*
        [] ⇒ *P pre x nxt*
      | (*y* # *ys*) ⇒ *P pre x* (*Some y*) ∧ *holds* (*Some x*) (*y* # *ys*) *nxt*)
by (*cases xs*, *auto*)

lemma *holds-Cons-P*:
  holds pre (*x* # *xs*) *nxt* ⟹ ∃ *y* . *P pre x y*
by (*cases xs*, *auto*)

lemma *holds-Cons-holds*:
  holds pre (*x* # *xs*) *nxt* ⟹ *holds* (*Some x*) *xs nxt*
by (*cases xs*, *auto*)

lemmas *tail-splitting-lemmas* =
  extract-split-tail holds-split-tail

Interaction between *holds*, *takeWhile*, and *extract*.

declare *if-split-asm* [*split*]

lemma *holds-takeW-extract*: *holds pre* (*takeW pre xs nxt*) (*extract pre xs nxt*)
apply(*induction pre xs nxt rule*: *takeW.induct*)
apply *auto*
subgoal for *pre x y ys*
  apply(*cases ys*)
  apply(*simp-all*)
  done
done

Interaction of *holds*, *takeWhile*, and *extract* with (@).

lemma *takeW-append*:
  takeW pre (*xs* @ *ys*) *nxt* =
    (*let y* = *case ys of* [] ⇒ *nxt* | *x* # - ⇒ *Some x in*
    (*let new-pre* = *case xs of* [] ⇒ *pre* | - ⇒ (*Some* (*last xs*)) *in*
      if holds pre xs y then xs @ *takeW new-pre ys nxt*
        else takeW pre xs y))
apply(*induction pre xs nxt rule*: *takeW.induct*)
apply (*simp-all add*: *Let-def split*: *list.split*)
done

lemma *holds-append*:
  holds pre (*xs* @ *ys*) *nxt* =
    (*let y* = *case ys of* [] ⇒ *nxt* | *x* # - ⇒ *Some x in*
    (*let new-pre* = *case xs of* [] ⇒ *pre* | - ⇒ (*Some* (*last xs*)) *in*
      holds pre xs y ∧ *holds new-pre ys nxt*))
apply(*induction pre xs nxt rule*: *takeW.induct*)
apply (*auto simp add*: *Let-def split*: *list.split*)
done

corollary *holds-cutoff*:
  holds pre (*l1*@*l2*) *nxt* ⟹ ∃ *nxt'* . *holds pre l1 nxt'*
by (*meson holds-append*)

lemma *extract-append*:
  extract pre (*xs* @ *ys*) *nxt* =
    (*let y* = *case ys of* [] ⇒ *nxt* | *x* # - ⇒ *Some x in*
    (*let new-pre* = *case xs of* [] ⇒ *pre* | - ⇒ (*Some* (*last xs*)) *in*
      if holds pre xs y then extract new-pre ys nxt else extract pre xs y))
apply(*induction pre xs nxt rule*: *takeW.induct*)
apply (*simp-all add*: *Let-def split*: *list.split*)
done

lemma *takeW-prefix*:
  prefix (*takeW pre l nxt*) *l*
by (*induction pre l nxt rule*: *takeW.induct*) *auto*

lemma *takeW-set*: *t* ∈ *set* (*TW.takeW P pre l nxt*) ⟹ *t* ∈ *set l*
by(*auto intro*: *takeW-prefix elim*: *set-prefix*)

lemma *holds-implies-takeW-is-identity*:
holds pre l nxt $\Longrightarrow$ *takeW pre l nxt* = *l*
by (*induction pre l nxt rule*: *takeW.induct*) *auto*

lemma *holds-takeW-is-identity*[*simp*]:
takeW pre l nxt = *l* $\longleftrightarrow$ *holds pre l nxt*
by (*induction pre l nxt rule*: *takeW.induct*) *auto*

lemma *takeW-takeW-extract*:
takeW pre (*takeW pre l nxt*) (*extract pre l nxt*)
= *takeW pre l nxt*
using *holds-takeW-extract holds-implies-takeW-is-identity*
by *blast*

Show the equivalence of two takeW with different pres

lemma *takeW-pre-eqI*:
$[\![\bigwedge x\ .\ l = [x] \Longrightarrow P\ pre\ x\ nxt \longleftrightarrow P\ pre'\ x\ nxt;$
$\bigwedge x1\ x2\ l'\ .\ l = x1\#x2\#l' \Longrightarrow P\ pre\ x1\ (Some\ x2) \longleftrightarrow P\ pre'\ x1\ (Some\ x2)]\!] \Longrightarrow$
takeW pre l nxt = *takeW pre′ l nxt*
apply(*cases l*)
subgoal by *auto*
subgoal for *a list*
by(*cases list*, *auto simp add*: *takeW-singleton takeW-split-tail*)
done

lemma *takeW-replace-pre*:
$[\![P\ pre\ x1\ n;\ n = ifhead\ xs\ nxt]\!] \Longrightarrow$ *prefix* (*TW.takeW P pre′* (*x1*#*xs*) *nxt*) (*TW.takeW P pre* (*x1*#*xs*) *nxt*)
apply(*cases xs*)
by(*auto simp add*: *takeW-singleton takeW-split-tail*)

Holds unfolding

This section contains various lemmas that show how one can deduce P pre' x' nxt' for some of pre' x' nxt' out of a list l, for which we know that holds pre l nxt is true.

lemma *holds-set-list*: $[\![holds\ pre\ l\ nxt;\ x \in set\ l]\!] \Longrightarrow \exists\ p\ y\ .\ P\ p\ x\ y$
by (*metis TW.holds-append holds-Cons-P split-list-first*)

lemma *holds-unfold*: *holds pre l None* $\Longrightarrow$
$l = [] \lor$
$(\exists\ x\ .\ l = [x] \land P\ pre\ x\ None) \lor$
$(\exists\ x\ y\ ys\ .\ l = (x\#y\#ys) \land P\ pre\ x\ (Some\ y) \land holds\ (Some\ x)\ (y\#ys)\ None)$
apply *auto* **by** (*meson holds.elims*(*2*))

lemma *holds-unfold-prexnxt*:
$[\![suffix\ (x0\#x1\#x2\#xs)\ l;\ holds\ pre\ l\ nxt]\!]$
$\Longrightarrow P\ (Some\ x0)\ x1\ (Some\ x2)$
by (*auto simp add*: *suffix-def TW.holds-append*)

lemma *holds-unfold-prexnxt′*:

$[\![$*holds pre l nxt*; l = (*zs*@(*x0*#*x1*#*x2*#*xs*))$]\!]$
$\Longrightarrow$ *P* (*Some x0*) *x1* (*Some x2*)
by (*auto simp add*: *TW.holds-append*)

lemma *holds-unfold-xz*:
$[\![$*suffix* (*x1*#*x2*#*xs*) *l*; *holds pre l nxt*$]\!]$ $\Longrightarrow$ $\exists$ *pre'*. *P pre' x1* (*Some x2*)
by (*auto simp add*: *suffix-def TW.holds-append*)

lemma *holds-unfold-prex*:
$[\![$*suffix* (*x1*#*x2*#*xs*) *l*; *holds pre l nxt*$]\!]$ $\Longrightarrow$ $\exists$ *nxt'*. *P* (*Some x1*) *x2 nxt'*
by (*auto simp add*: *suffix-def TW.holds-append dest*: *holds-Cons-P*)

lemma *holds-suffix*:
$[\![$*holds pre l nxt*; *suffix l' l*$]\!]$ $\Longrightarrow$ $\exists$ *pre'*. *holds pre' l' nxt*
by (*metis holds-append suffix-def*)

lemma *holds-unfold-prelnil*:
$[\![$*holds pre l nxt*; l = (*zs*@(*x0*#*x1*#[]))$]\!]$
$\Longrightarrow$ *P* (*Some x0*) *x1 nxt*
by (*auto simp add*: *TW.holds-append*)

end
end

Chapter 2

Abstract, and Concrete Parametrized Models

This is the core of our verification – the abstract and parametrized models that cover a wide range of protocols.

2.1 Network model

theory *Network-Model*
imports
infrastructure/Agents
infrastructure/Tools
infrastructure/TakeWhile
begin

as is already defined as a type synonym for nat.

type-synonym *ifs = nat*

The authenticated hop information consists of the interface identifiers UpIF, DownIF and an identifier of the AS to which the hop information belongs. Furthermore, this record is extensible and can include additional authenticated hop information (aahi).

record *ahi* =
UpIF :: *ifs option*
DownIF :: *ifs option*
ASID :: *as*

type-synonym *'aahi ahis = 'aahi ahi-scheme*

locale *network-model = compromised* +
fixes
auth-seg0 :: (*'ainfo* × *'aahi ahi-scheme list*) *set*
and *tgtas* :: *as* ⇒ *ifs* ⇒ *as option*
and *tgtif* :: *as* ⇒ *ifs* ⇒ *ifs option*
begin

2.1.1 Interface check

Check if the interfaces of two adjacent hop fields match. If both hops are compromised we also interpret the link as valid.

fun *if-valid* :: *'aahi ahis option* ⇒ *'aahi ahis* => *'aahi ahis option* ⇒ *bool* **where**
if-valid None hf - — this is the case for the leaf AS
= *True*
| *if-valid* (*Some hf1*) (*hf2*) -
= ((∃ *downif* . *DownIF hf2* = *Some downif* ∧
tgtas (*ASID hf2*) *downif* = *Some* (*ASID hf1*) ∧
tgtif (*ASID hf2*) *downif* = *UpIF hf1*)
∨ *ASID hf1* ∈ *bad* ∧ *ASID hf2* ∈ *bad*)

makes sure that: the segment is terminated, i.e. the first AS's HF has Eo = None

fun *terminated* :: *'aahi ahis list* ⇒ *bool* **where**
terminated (*hf*#*xs*) ⟷ *DownIF hf* = *None* ∨ *ASID hf* ∈ *bad*
| *terminated* [] = *True*

makes sure that: the segment is rooted, i.e. the last HF has UpIF = None

fun *rooted* :: *'aahi ahis list* ⇒ *bool* **where**
rooted [*hf*] ⟷ *UpIF hf* = *None* ∨ *ASID hf* ∈ *bad*
| *rooted* (*hf*#*xs*) = *rooted xs*

| *rooted* [] = *True*

abbreviation *ifs-valid* **where**
ifs-valid pre l nxt ≡ *TW.holds if-valid pre l nxt*

abbreviation *ifs-valid-prefix* **where**
ifs-valid-prefix pre l nxt ≡ *TW.takeW if-valid pre l nxt*

abbreviation *ifs-valid-None* **where**
ifs-valid-None l ≡ *ifs-valid None l None*

abbreviation *ifs-valid-None-prefix* **where**
ifs-valid-None-prefix l ≡ *ifs-valid-prefix None l None*

lemma *strip-ifs-valid-prefix*:
pfragment ainfo l auth-seg0 ⟹ *pfragment ainfo* (*ifs-valid-prefix pre l nxt*) *auth-seg0*
by (*auto elim*: *pfragment-prefix′ intro*: *TW.takeW-prefix*)

Given the AS and an interface identifier of a channel, obtain the AS and interface at the other end of the same channel.

abbreviation *rev-link* :: *as* ⇒ *ifs* ⇒ *as option* × *ifs option* **where**
rev-link a1 i1 ≡ (*tgtas a1 i1*, *tgtif a1 i1*)

end
end

2.2 Abstract Model

theory *Parametrized-Dataplane-0*
imports
Network-Model
infrastructure/Event-Systems
begin

A packet consists of an authenticated info field (e.g., the timestamp of the control plane level beacon creating the segment), as well as past and future paths. Furthermore, there is a history variable *history* that accurately records the actual path – this is only used for the purpose of expressing the desired security property ("Detectability", see below).

record $('aahi, 'ainfo)$ *pkt0* =
AInfo :: *'ainfo*
past :: *'aahi ahi-scheme list*
future :: *'aahi ahi-scheme list*
history :: *'aahi ahi-scheme list*

In this model, the state consists of channel state and local state, each containing sets of packets (which we occasionally also call messages).

record $('aahi, 'ainfo)$ *dp0-state* =
chan :: $(as \times ifs \times as \times ifs) \Rightarrow ('aahi, 'ainfo)$ *pkt0 set*
loc :: $as \Rightarrow ('aahi, 'ainfo)$ *pkt0 set*

We now define the events type; it will be explained below.

datatype $('aahi, 'ainfo)$ *evt0* =
evt-dispatch-int0 as $('aahi, 'ainfo)$ *pkt0*
| *evt-recv0 as ifs* $('aahi, 'ainfo)$ *pkt0*
| *evt-send0 as ifs* $('aahi, 'ainfo)$ *pkt0*
| *evt-deliver0 as* $('aahi, 'ainfo)$ *pkt0*
| *evt-dispatch-ext0 as ifs* $('aahi, 'ainfo)$ *pkt0*
| *evt-observe0* $('aahi, 'ainfo)$ *dp0-state*
| *evt-skip0*

context *network-model*
begin

We define shortcuts denoting that from a state s, a packet pkt is added to either a local state or a channel, yielding state s'. No other part of the state is modified.

definition *dp0-add-loc* :: $('aahi, 'ainfo)$ *dp0-state* $\Rightarrow ('aahi, 'ainfo)$ *dp0-state*
$\Rightarrow as \Rightarrow ('aahi, 'ainfo)$ *pkt0* $\Rightarrow bool$
where
$dp0\text{-}add\text{-}loc\ s\ s'\ asid\ pkt \equiv s' = s(\!|loc := (loc\ s)(asid := loc\ s\ asid \cup \{pkt\})|\!)$

This is a shortcut to denote adding a message to an inter-AS channel. Note that it requires the link to exist.

definition *dp0-add-chan* :: $('aahi, 'ainfo)$ *dp0-state* $\Rightarrow ('aahi, 'ainfo)$ *dp0-state*
$\Rightarrow as \Rightarrow ifs \Rightarrow ('aahi, 'ainfo)$ *pkt0* $\Rightarrow bool$ **where**
$dp0\text{-}add\text{-}chan\ s\ s'\ a1\ i1\ pkt \equiv$
$\exists\, a2\ i2\ .\ rev\text{-}link\ a1\ i1 = (Some\ a2,\ Some\ i2) \wedge$
$s' = s(\!|chan := (chan\ s)((a1,\ i1,\ a2,\ i2) := chan\ s\ (a1,\ i1,\ a2,\ i2) \cup \{pkt\})|\!)$

Predicate that returns true if a given packet is contained in a given channel.

definition *dp0-in-chan* :: $('aahi, 'ainfo)$ *dp0-state* $\Rightarrow$ *as* $\Rightarrow$ *ifs* $\Rightarrow$ $('aahi, 'ainfo)$ *pkt0* $\Rightarrow$ *bool* **where**

dp0-in-chan s a1 i1 pkt $\equiv$
$\exists$ *a2 i2* . *rev-link a1 i1* = (*Some a2*, *Some i2*) $\wedge$ *pkt* $\in$ (*chan s*)(*a2*, *i2*, *a1*, *i1*)

lemmas *dp0-msgs* = *dp0-add-loc-def dp0-add-chan-def dp0-in-chan-def*

2.2.1 Events

A typical sequence of events is the following:

- An AS creates a new packet using *evt-dispatch-int0* event and puts the packet into its local state.
- The AS forwards the packet to the next AS with the *evt-send0* event, which puts the message into an inter-AS channel.
- The next AS takes the packet from the channel and puts it in the local state in *evt-recv0*.
- The last two steps are repeated as the packet gets forwarded from hop to hop through the network, until it reaches the final AS.
- The final AS delivers the packet internally to the intended destination with the event *evt-deliver0*.

definition
dp0-dispatch-int
where
dp0-dispatch-int s m ainfo asid pas fut hist s' $\equiv$
— guard: check that the future path is a fragment of an authorized segment. In reality, honest agents will always choose a path that is a prefix of an authorized segment, but for our models this difference is not significant.
m = (| *AInfo* = *ainfo*, *past* = *pas*, *future* = *fut*, *history* = *hist* |) $\wedge$
hist = [] $\wedge$
pfragment ainfo fut auth-seg0 $\wedge$
— action: Update the state to include m
dp0-add-loc s s' asid m

definition
dp0-recv
where
dp0-recv s m asid ainfo hf1 downif pas fut hist s' $\equiv$
— guard: there are at least two hop fields left, which means we can advance the packet by one hop.
m = (| *AInfo* = *ainfo*, *past* = *pas*, *future* = *hf1* # *fut*, *history* = *hist* |) $\wedge$
dp0-in-chan s asid downif m $\wedge$

ASID hf1 = *asid* $\wedge$

— action: Update state to include message
dp0-add-loc s s' asid (|
AInfo = *ainfo*,

$\quad$ *past* = *pas*,
future = *hf1* # *fut*,
history = *hist*
⦈

definition
dp0-send
where
dp0-send s m asid ainfo hf1 upif pas fut hist s' ≡
— guard: there are at least two hop fields left, which means we can advance the packet by one hop.
$m = (\!|$ *AInfo* = *ainfo*, *past* = *pas*, *future* = *hf1#fut*, *history* = *hist* $|\!) \wedge$
$m \in (loc\ s)\ asid \wedge$
UpIF hf1 = *Some upif* ∧
ASID hf1 = *asid* ∧

— action: Update state to include modified message
dp0-add-chan s s' asid upif ⦇
AInfo = *ainfo*,
past = *hf1* # *pas*,
future = *fut*,
history = *hf1* # *hist*
⦈

This event represents the destination receiving the packet. Our properties are not expressed over what happens when an end hosts receives a packet (but rather what happens with a packet while it traverses the network). We only need this event to push the last hop field from the future path into the past path, as the detectability property is expressed over the past path.

definition
dp0-deliver
where
dp0-deliver s m asid ainfo hf1 pas fut hist s' ≡
$m = (\!|$ *AInfo* = *ainfo*, *past* = *pas*, *future* = *hf1#fut*, *history* = *hist* $|\!) \wedge$
ASID hf1 = *asid* ∧
$m \in (loc\ s)\ asid \wedge$
fut = [] ∧

— action: Update state to include modified message
dp0-add-loc s s' asid
⦇
AInfo = *ainfo*,
past = *hf1* # *pas*,
future = [],
history = *hf1* # *hist*
⦈

— Direct dispatch event. A node with asid sends a packet on its outgoing interface upif.
Note that the attacker is NOT part of the real past path. However, detectability is still achieved in practice, since hf (the hop field of the next AS) points with its downif towards the attacker node.
definition
dp0-dispatch-ext
where

$dp0\text{-}dispatch\text{-}ext\ s\ m\ asid\ ainfo\ upif\ pas\ fut\ hist\ s' \equiv$
$m = (\!|\ AInfo = ainfo,\ past = pas,\ future = fut,\ history = hist\ |\!) \wedge$
$hist = [] \wedge$

$pfragment\ ainfo\ fut\ auth\text{-}seg0 \wedge$

— action: Update state to include attacker message
$dp0\text{-}add\text{-}chan\ s\ s'\ asid\ upif\ m$

2.2.2 Transition system

fun *dp0-trans* **where**
$dp0\text{-}trans\ s\ (evt\text{-}dispatch\text{-}int0\ asid\ m)\ s' \longleftrightarrow$
$(\exists\, ainfo\ pas\ fut\ hist.\ dp0\text{-}dispatch\text{-}int\ s\ m\ ainfo\ asid\ pas\ fut\ hist\ s')\ |$
$dp0\text{-}trans\ s\ (evt\text{-}recv0\ asid\ downif\ m)\ s' \longleftrightarrow$
$(\exists\, ainfo\ hf1\ pas\ fut\ hist.\ dp0\text{-}recv\ s\ m\ asid\ ainfo\ hf1\ downif\ pas\ fut\ hist\ s')\ |$
$dp0\text{-}trans\ s\ (evt\text{-}send0\ asid\ upif\ m)\ s' \longleftrightarrow$
$(\exists\, ainfo\ hf1\ pas\ fut\ hist.\ dp0\text{-}send\ s\ m\ asid\ ainfo\ hf1\ upif\ pas\ fut\ hist\ s')\ |$
$dp0\text{-}trans\ s\ (evt\text{-}deliver0\ asid\ m)\ s' \longleftrightarrow$
$(\exists\, ainfo\ hf1\ pas\ fut\ hist.\ dp0\text{-}deliver\ s\ m\ asid\ ainfo\ hf1\ pas\ fut\ hist\ s')\ |$
$dp0\text{-}trans\ s\ (evt\text{-}dispatch\text{-}ext0\ asid\ upif\ m)\ s' \longleftrightarrow$
$(\exists\, ainfo\ pas\ fut\ hist.\ dp0\text{-}dispatch\text{-}ext\ s\ m\ asid\ ainfo\ upif\ pas\ fut\ hist\ s')\ |$
$dp0\text{-}trans\ s\ (evt\text{-}observe0\ s'')\ s' \longleftrightarrow s = s' \wedge s = s''\ |$
$dp0\text{-}trans\ s\ evt\text{-}skip0\ s' \longleftrightarrow s = s'$

definition $dp0\text{-}init :: ('aahi,\ 'ainfo)\ dp0\text{-}state$ **where**
$dp0\text{-}init \equiv (\!|chan = (\lambda\text{-}.\ \{\}),\ loc = (\lambda\text{-}.\ \{\})|\!)$

definition $dp0 :: (('aahi,\ 'ainfo)\ evt0,\ ('aahi,\ 'ainfo)\ dp0\text{-}state)\ ES$ **where**
$dp0 \equiv (\!|$
$init = (=)\ dp0\text{-}init,$
$trans = dp0\text{-}trans$
$|\!)$

lemmas *dp0-trans-defs = dp0-dispatch-int-def dp0-recv-def dp0-send-def dp0-deliver-def dp0-dispatch-ext-def*
lemmas *dp0-defs = dp0-def dp0-init-def dp0-trans-defs*

soup is a predicate that is true for a packet m and a state s, if m is contained anywhere in the system (either in the local state or channels).

definition *soup* **where** $soup\ m\ s \equiv \exists\, x.\ m \in (loc\ s)\ x \vee (\exists\, x.\ m \in (chan\ s)\ x)$

declare *soup-def* [*simp*]
declare *if-split-asm* [*split*]

lemma *dp0-add-chan-msgs*:
assumes $dp0\text{-}add\text{-}chan\ s\ s'\ asid\ upif\ m$ **and** $soup\ n\ s'$ **and** $n \neq m$
shows $soup\ n\ s$
using *assms* **by** (*auto simp add*: *dp0-add-chan-def*)

2.2.3 Path authorization property

Path authorization is defined as: For all messages in the system: the future path is a fragment of an authorized path. We strengthen this property by including the real past path (the recorded history that can not be faked by the attacker). The concatenation of these path remains invariant during forwarding, which simplifies our proof. Note that the history path is in reverse order.

definition *auth-path* :: ('*aahi*, '*ainfo*) *pkt0* ⇒ *bool* **where**
auth-path m ≡ *pfragment* (*AInfo m*) (*rev* (*history m*) @ *future m*) *auth-seg0*

definition *inv-auth* :: ('*aahi*, '*ainfo*) *dp0-state* ⇒ *bool* **where**
inv-auth s ≡ ∀ *m* . *soup m s* ⟶ *auth-path m*

lemma *inv-authI*:
assumes ⋀*m* . *soup m s* ⟹ *pfragment* (*AInfo m*) (*rev* (*history m*) @ *future m*) *auth-seg0*
shows *inv-auth s*
apply(*auto simp add*: *inv-auth-def auth-path-def*)
using *assms soup-def* **by** *blast+*

lemma *inv-authD*:
assumes *inv-auth s soup m s*
shows *pfragment* (*AInfo m*) (*rev* (*history m*) @ *future m*) *auth-seg0*
using *assms* **by**(*auto simp add*: *inv-auth-def auth-path-def*) *blast*

lemma *inv-auth-add-chan*[*elim*!]:
assumes *dp0-add-chan s s' asid upif m* **and** *inv-auth s*
and *pfragment* (*AInfo m*) (*rev* (*history m*) @ *future m*) *auth-seg0*
shows *inv-auth s'*
proof(*rule inv-authI*)
fix *n*
assume *soup n s'*
then show *pfragment* (*AInfo n*) (*rev* (*history n*) @ *future n*) *auth-seg0*
using *assms* **by**(*cases m=n*, *auto dest*!: *dp0-add-chan-msgs dest*: *inv-authD*)
qed

lemma *inv-auth-add-loc*[*elim*!]:
assumes *dp0-add-loc s s' asid m* **and** *inv-auth s*
and *pfragment* (*AInfo m*) (*rev* (*history m*) @ *future m*) *auth-seg0*
shows *inv-auth s'*
proof(*rule inv-authI*)
fix *n*
assume *soup n s'*
then show *pfragment* (*AInfo n*) (*rev* (*history n*) @ *future n*) *auth-seg0*
using *assms* **apply**(*cases m=n*, *auto 3 4 simp add*: *dp0-add-loc-def dest*: *inv-authD*)
by (*meson auth-path-def inv-auth-def soup-def*)
qed

lemma *Inv-inv-auth*: *Inv dp0 inv-auth*
proof(*rule Invariant-rule*)
fix *s0*
show *init dp0 s0* ⟹ *inv-auth s0*
by (*auto simp add*: *dp0-def dp0-init-def intro*!: *inv-authI*)

next
  fix *s e s'*
  show $[\![dp0: s{-}e{\rightarrow}\ s'; \mathit{inv\text{-}auth}\ s]\!] \Longrightarrow \mathit{inv\text{-}auth}\ s'$
  proof (*auto simp add*: *dp0-def elim*!: *dp0-trans.elims*)
    fix *m asid ainfo hf1 downif pas fut hist*
    assume *inv-auth s dp0-recv s m asid ainfo hf1 downif pas fut hist s'*
    then show *inv-auth s'*
      by(*auto simp add*: *dp0-defs dp0-add-loc-def pfragment-def intro*!: *inv-authI dest*!: *inv-authD*)
        (*auto simp add*: *dp0-in-chan-def*)
  qed(*auto simp add*: *dp0-defs*, *auto intro*: *pfragment-prefix dest*!: *inv-authD*)
qed

abbreviation *TR-auth* **where** *TR-auth* ≡
  $\{\tau \mid \tau\ .\ \forall\ s\ .\ \mathit{evt\text{-}observe0}\ s \in \mathit{set}\ \tau \longrightarrow \mathit{inv\text{-}auth}\ s\}$

lemma *tr0-satisfies-pathauthorization*: $dp0 \models_{ES} \mathit{TR\text{-}auth}$
  using *Inv-inv-auth*
  apply(*intro trace-property-rule*[**where** $?I{=}\lambda\tau\ s.\ \tau \in \mathit{TR\text{-}auth}$])
  apply (*auto elim*!: *InvE simp add*: *inv-auth-def*)
  apply(*auto simp add*: *dp0-defs elim*!: *dp0-trans.elims*)
  by *blast+*

2.2.4 Detectability property

The attacker sending a packet to another AS is not part of the real path. However, the next hop's interface will point to the attacker AS (if the hop field is valid), thus the attacker remains identifiable.

Detectability, the first property: the past real path is a prefix of the past path

definition *inv-detect* :: (*'aahi*, *'ainfo*) *dp0-state* ⇒ *bool* **where**
  $\mathit{inv\text{-}detect}\ s \equiv \forall\, m\ .\ \mathit{soup}\ m\ s \longrightarrow \mathit{prefix}\ (\mathit{history}\ m)\ (\mathit{past}\ m)$

lemma *inv-detectI*:
  assumes $\bigwedge m\ x\ .\ \mathit{soup}\ m\ s \Longrightarrow \mathit{prefix}\ (\mathit{history}\ m)\ (\mathit{past}\ m)$
    shows *inv-detect s*
  using *assms* **by**(*auto simp add*: *inv-detect-def*)

lemma *inv-detectD*:
  assumes *inv-detect s*
    shows $\bigwedge m\ x\ .m \in (\mathit{loc}\ s)\ x \Longrightarrow \mathit{prefix}\ (\mathit{history}\ m)\ (\mathit{past}\ m)$
      and $\bigwedge m\ x\ .m \in (\mathit{chan}\ s)\ x \Longrightarrow \mathit{prefix}\ (\mathit{history}\ m)\ (\mathit{past}\ m)$
  using *assms* **by**(*auto simp add*: *inv-detect-def*) *blast*

lemma *inv-detect-add-chan*[*elim*!]:
  assumes *dp0-add-chan s s' asid upif m inv-detect s prefix* (*history m*) (*past m*)
  shows *inv-detect s'*
proof(*rule inv-detectI*)
  fix *n*
  assume *soup n s'*
  then show *prefix* (*history n*) (*past n*)
    using *assms* **by**(*cases m=n*, *auto dest*!: *dp0-add-chan-msgs dest*: *inv-detectD*)

qed

lemma *inv-detect-add-loc*[*elim*!]:
  assumes *dp0-add-loc s s' asid m inv-detect s prefix* (*history m*) (*past m*)
  shows *inv-detect s'*
proof(*rule inv-detectI*)
  fix *n*
  assume *soup n s'*
  then show *prefix* (*history n*) (*past n*)
    using *assms* **by**(*cases m=n*, *auto 3 4 simp add*: *dp0-add-loc-def dest*: *inv-detectD*)
qed

lemma *Inv-inv-detect*: *Inv dp0 inv-detect*
proof (*rule InvI*, *erule reach.induct*)
  fix *s0*
  show *init dp0 s0* $\Longrightarrow$ *inv-detect s0*
    by (*auto simp add*: *dp0-def dp0-init-def intro*!: *inv-detectI*)
  next
  fix *s e s'*
  show $[\![$*dp0*: *s*$-$*e*$\rightarrow$ *s'*; *inv-detect s*$]\!]$ $\Longrightarrow$ *inv-detect s'*
    by(*auto simp add*: *dp0-defs elim*!: *dp0-trans.elims*)
      (*fastforce simp add*: *dp0-in-chan-def dest*: *inv-detectD*)+
qed

abbreviation *TR-detect* **where** *TR-detect* $\equiv$ $\{\tau \mid \tau\ .\ \forall\ s\ .$ *evt-observe0 s* $\in$ *set* $\tau \longrightarrow$ *inv-detect s*$\}$

lemma *tr0-satisfies-detectability*: *dp0* $\models_{ES}$ *TR-detect*
  using *Inv-inv-detect*
  by(*intro trace-property-rule*[**where** *?I*=$\lambda\tau$ *s*. $\tau \in$ *TR-detect*])
    (*fastforce simp add*: *dp0-defs dp0-in-chan-def elim*!: *dp0-trans.elims dest*: *inv-detectD*)+

end
end

2.3 Intermediate Model

theory *Parametrized-Dataplane-1*
  imports
    Parametrized-Dataplane-0
    infrastructure/Message
begin

This model is almost identical to the previous one. The only changes are (i) that the receive event performs an interface check and (ii) that we permit the attacker to send any packet with a future path whose interface-valid prefix is authorized, as opposed to requiring that the entire future path is authorized. This means that the attacker can combine hop fields of subsequent ASes as long as the combination is either authorized, or the interfaces of the two hop fields do not correspond to each other. In the latter case the packet will not be delivered to (or accepted by) the second AS. Because (i) requires the *evt-recv0* event to check the interface over which packets are received, in the mapping from this model to the abstract model we can thus cut off all invalid hop fields from the future path.

type-synonym (*'aahi*, *'ainfo*) *dp1-state* = (*'aahi*, *'ainfo*) *dp0-state*
type-synonym (*'aahi*, *'ainfo*) *pkt1* = (*'aahi*, *'ainfo*) *pkt0*
type-synonym (*'aahi*, *'ainfo*) *evt1* = (*'aahi*, *'ainfo*) *evt0*

context *network-model*
begin

2.3.1 Events

definition
  dp1-dispatch-int
where
  dp1-dispatch-int s m ainfo asid pas fut hist s' ≡
    — guard: check that the future path is a fragment of an authorized segment. In reality, honest agents will always choose a path that is a prefix of an authorized segment, but for our models this difference is not significant.
    m = (| *AInfo* = *ainfo*, *past* = *pas*, *future* = *fut*, *history* = *hist* |) ∧
    hist = [] ∧
    pfragment ainfo (*ifs-valid-prefix None fut None*) *auth-seg0* ∧
    — action: Update the state to include m
    dp0-add-loc s s' asid m

We construct an artificial hop field that contains a specified asid and upif. The other fields are irrelevant, as we only use this artificial hop field as "previous" hop field in the *ifs-valid-prefix* function. This is used in the direct dispatch event: the interface-valid prefix must be authorized. Since the dispatching AS' own hop field is not part of the future path, but the AS directly after the it does check for the interface correctness, we need this artificial hop field.

abbreviation *prev-hf* **where**
  prev-hf asid upif ≡
  (*Some* (|*UpIF* = *Some upif*, *DownIF* = *None*, *ASID* = *asid*, ... = *undefined*|))

definition
  dp1-dispatch-ext

where
dp1-dispatch-ext s m asid ainfo upif pas fut hist s' $\equiv$
$m = (\!|$ *AInfo* $=$ *ainfo*, *past* $=$ *pas*, *future* $=$ *fut*, *history* $=$ *hist* $|\!) \wedge$
hist $= [] \wedge$
pfragment ainfo (*ifs-valid-prefix* (*prev-hf asid upif*) *fut None*) *auth-seg0* $\wedge$

— action: Update state to include attacker message
dp0-add-chan s s' asid upif m

definition
dp1-recv
where
dp1-recv s m asid ainfo hf1 downif pas fut hist s' $\equiv$
DownIF hf1 $=$ *Some downif*
$\wedge$ *dp0-recv s m asid ainfo hf1 downif pas fut hist s'*

2.3.2 Transition system

fun *dp1-trans* **where**
dp1-trans s (*evt-dispatch-int0 asid m*) *s'* $\longleftrightarrow$
($\exists$ *ainfo pas fut hist*. *dp1-dispatch-int s m ainfo asid pas fut hist s'*) |
dp1-trans s (*evt-dispatch-ext0 asid upif m*) *s'* $\longleftrightarrow$
($\exists$ *ainfo pas fut hist* . *dp1-dispatch-ext s m asid ainfo upif pas fut hist s'*) |
dp1-trans s (*evt-recv0 asid downif m*) *s'* $\longleftrightarrow$
($\exists$ *ainfo hf1 pas fut hist*. *dp1-recv s m asid ainfo hf1 downif pas fut hist s'*) |
dp1-trans s e s' $\longleftrightarrow$ *dp0-trans s e s'*

definition *dp1-init* :: (*'aahi*, *'ainfo*) *dp1-state* **where**
dp1-init $\equiv (\!|$*chan* $= (\lambda$-. $\{\})$, *loc* $= (\lambda$-. $\{\})|\!)$

definition *dp1* :: ((*'aahi*, *'ainfo*) *evt1*, (*'aahi*, *'ainfo*) *dp1-state*) *ES* **where**
dp1 $\equiv (\!|$
init $= (=)$ *dp1-init*,
trans $=$ *dp1-trans*
$|\!)$

lemmas *dp1-trans-defs* $=$ *dp0-trans-defs dp1-dispatch-ext-def dp1-recv-def*
lemmas *dp1-defs* $=$ *dp1-def dp1-dispatch-int-def dp1-init-def dp1-trans-defs*

fun *pkt1to0chan* :: *as* $\Rightarrow$ *ifs* $\Rightarrow$ (*'aahi*, *'ainfo*) *pkt1* $\Rightarrow$ (*'aahi*, *'ainfo*) *pkt0* **where**
pkt1to0chan asid upif $(\!|$ *AInfo* $=$ *ainfo*, *past* $=$ *pas*, *future* $=$ *fut*, *history* $=$ *hist* $|\!) =$
$(\!|$ *pkt0.AInfo* $=$ *ainfo*, *past* $=$ *pas*, *future* $=$ *ifs-valid-prefix* (*prev-hf asid upif*) *fut None*, *history* $=$ *hist*$|\!)$

fun *pkt1to0loc* :: (*'aahi*, *'ainfo*) *pkt1* $\Rightarrow$ (*'aahi*, *'ainfo*) *pkt0* **where**
pkt1to0loc $(\!|$ *AInfo* $=$ *ainfo*, *past* $=$ *pas*, *future* $=$ *fut*, *history* $=$ *hist* $|\!) =$
$(\!|$ *pkt0.AInfo* $=$ *ainfo*, *past* $=$ *pas*, *future* $=$ *ifs-valid-prefix None fut None*, *history* $=$ *hist*$|\!)$

definition *R10* :: (*'aahi*, *'ainfo*) *dp1-state* $\Rightarrow$ (*'aahi*, *'ainfo*) *dp0-state* **where**
R10 s $=$
$(\!|$*chan* $= \lambda(a1, i1, a2, i2)$. (*pkt1to0chan a1 i1*) ‘ ((*chan s*) (*a1*, *i1*, *a2*, *i2*)),
loc $= \lambda x$. *pkt1to0loc* ‘ ((*loc s*) *x*)$|\!)$

fun π_1 :: ('*aahi*, '*ainfo*) *evt1* ⇒ ('*aahi*, '*ainfo*) *evt0* **where**
  π_1 (*evt-dispatch-int0 asid m*) = *evt-dispatch-int0 asid* (*pkt1to0loc m*)
| π_1 (*evt-recv0 asid downif m*) = *evt-recv0 asid downif* (*pkt1to0loc m*)
| π_1 (*evt-send0 asid upif m*) = *evt-send0 asid upif* (*pkt1to0loc m*)
| π_1 (*evt-deliver0 asid m*) = *evt-deliver0 asid* (*pkt1to0loc m*)
| π_1 (*evt-dispatch-ext0 asid upif m*) = *evt-dispatch-ext0 asid upif* (*pkt1to0chan asid upif m*)
| π_1 (*evt-observe0 s*) = *evt-observe0* (*R10 s*)
| π_1 *evt-skip0* = *evt-skip0*

declare *TW.takeW.elims*[*elim*]

lemma *dp1-refines-dp0*: *dp1* $\sqsubseteq_{\pi 1}$ *dp0*
proof(*rule simulate-ES-fun*[**where** *?h* = *R10*])
  fix *s0*
  assume *init dp1 s0*
  then show *init dp0* (*R10 s0*)
    by(*auto simp add*: *dp0-defs dp1-defs R10-def*)
next
  fix *s e s'*
  assume *dp1*: *s*−*e*→ *s'*
  then show *dp0*: *R10 s*− π_1 *e*→ *R10 s'*
  proof(*auto simp add*: *dp1-def elim*!: *dp1-trans.elims dp0-trans.elims*)
    fix *m ainfo asid pas fut hist*
    assume *dp1-dispatch-int s m ainfo asid pas fut hist s'*
    then show *dp0*: *R10 s*−*evt-dispatch-int0 asid* (*pkt1to0loc m*)→ *R10 s'*
      by(*auto 3 4 simp add*: *dp0-defs dp1-defs dp0-msgs R10-def*
              intro: *TW.takeW-prefix elim*: *pfragment-prefix' dest*: *strip-ifs-valid-prefix*)
  next
    fix *m asid ainfo hf1 downif pas fut hist*
    assume *dp1-recv s m asid ainfo hf1 downif pas fut hist s'*
    then show *dp0*: *R10 s*−*evt-recv0 asid downif* (*pkt1to0loc m*)→ *R10 s'*
      by(*auto simp add*: *dp0-defs dp1-defs dp0-msgs R10-def TW.takeW-split-tail*
          elim!: *rev-image-eqI intro*!: *ext*)
  next
    fix *m asid ainfo hf1 upif pas fut hist*
    assume *dp0-send s m asid ainfo hf1 upif pas fut hist s'*
    then show *dp0*: *R10 s*−*evt-send0 asid upif* (*pkt1to0loc m*)→ *R10 s'*
      by(*cases ifs-valid-None-prefix* (*hf1* # *fut*))
        (*auto 3 4 simp add*: *dp0-defs dp1-defs dp0-msgs R10-def TW.takeW-split-tail TW.takeW.simps*
              elim!: *rev-image-eqI TW.takeW.elims intro*!: *TW.takeW-pre-eqI*)
  next
    fix *m asid ainfo hf1 pas fut hist*
    assume *dp0-deliver s m asid ainfo hf1 pas fut hist s'*
    then show *dp0*: *R10 s*−*evt-deliver0 asid* (*pkt1to0loc m*)→ *R10 s'*
      by(*auto simp add*: *dp0-defs dp1-defs dp0-msgs R10-def TW.takeW.simps*
          intro!: *ext elim*!: *rev-image-eqI TW.takeW.elims*)
  qed(*auto 3 4 simp add*: *dp0-defs dp1-defs dp0-msgs R10-def TW.takeW-split-tail*)
qed

2.3.3 Auxilliary definitions

These definitions are not directly needed in the parametrized models, but they are useful for instances.

fun ASO :: $msgterm \Rightarrow nat\ option$ **where**
$ASO\ (AS\ ifs) = Some\ ifs \mid ASO\ \varepsilon = None$

Check if interface option is matched by a msgterm.

fun $ASIF$ :: $ifs\ option \Rightarrow msgterm \Rightarrow bool$ **where**
$ASIF\ (Some\ a)\ (AS\ a') = (a{=}a')$
$\mid ASIF\ None\ \varepsilon = True$
$\mid ASIF$ - - $= False$

Turn a msgterm to an ifs option. Note that this maps both ε (the msgterm denoting the lack of an interface) and arbitrary other msgterms that are not of the form "AS t" to None. The result may thus be ambiguous. Use with care.

fun $term2if$ :: $msgterm \Rightarrow ifs\ option$ **where**
$term2if\ (AS\ a) = Some\ a$
$\mid term2if\ \varepsilon = None$
$\mid term2if$ - $= None$

fun $if2term$ :: $ifs\ option \Rightarrow msgterm$ **where** $if2term\ (Some\ a) = AS\ a \mid if2term\ None = \varepsilon$

lemma *if2term-eq*[*elim*]: $if2term\ a = if2term\ b \Longrightarrow a = b$
apply(*cases a*, *cases b*, *auto*)
by (*metis ASO.simps(1) if2term.elims msgterm.distinct(1)*)

lemma *term2if-if2termm*[*simp*]: $term2if\ (if2term\ a) = a$ **apply**(*cases a*) **by** *auto*

fun $hf2term$ :: $ahi \Rightarrow msgterm$ **where**
$hf2term\ (\!|UpIF = upif,\ DownIF = downif,\ ASID = asid|\!) = L\ [if2term\ upif,\ if2term\ downif,\ Num\ asid]$

fun $term2hf$ :: $msgterm \Rightarrow ahi$ **where**
$term2hf\ (L\ [upif,\ downif,\ Num\ asid]) = (\!|UpIF = term2if\ upif,\ DownIF = term2if\ downif,\ ASID = asid|\!)$

lemma *term2hf-hf2term*[*simp*]: $term2hf\ (hf2term\ hf) = hf$ **apply**(*cases hf*) **by** *auto*

lemma *ahi-eq*:
$[\![ASID\ ahi' = ASID\ (ahi{::}ahi);\ ASIF\ (DownIF\ ahi')\ downif;\ ASIF\ (UpIF\ ahi')\ upif;$
$ASIF\ (DownIF\ ahi)\ downif;\ ASIF\ (UpIF\ ahi)\ upif]\!] \Longrightarrow ahi = ahi'$
by(*cases ahi*, *cases ahi'*)
(*auto elim*: *ASIF.elims ahi.cases*)

end
end

2.4 Concrete Parametrized Model

This is the refinement of the intermediate dataplane model. This model is parametric, and requires instantiation of the hop validation function, (and other parameters). We do so in the *Parametrized-Dataplane-3-directed* and *Parametrized-Dataplane-3-undirected* models. Nevertheless, this model contains the complete refinement proof, albeit the hard case, the refinement of the attacker event, is assumed to hold. The crux of the refinement proof is thus shown in these directed/undirected instance models. The definitions to be given by the instance are those of the locales *dataplane-2-defs* (which contains the basic definitions needed for the protocol, such as the verification of a hop field, called *hf-valid-generic*), and *dataplane-2-ik-defs* (containing the definition of components of the intruder knowledge). The proof obligations are those in the locale *dataplane-2*.

theory *Parametrized-Dataplane-2*
  imports
    Parametrized-Dataplane-1 Network-Model
begin

record $('aahi, 'uhi)\ HF =$
  $AHI :: 'aahi\ ahi\text{-}scheme$
  $UHI :: 'uhi$
  $HVF :: msgterm$

record $('aahi, 'uhi, 'ainfo)\ pkt2 =$
  $AInfo :: 'ainfo$
  $UInfo :: msgterm$
  $past :: ('aahi, 'uhi)\ HF\ list$
  $future :: ('aahi, 'uhi)\ HF\ list$
  $history :: 'aahi\ ahi\text{-}scheme\ list$

We use pkt2 instead of pkt, but otherwise the state remains unmodified in this model.

record $('aahi, 'uhi, 'ainfo)\ dp2\text{-}state =$
  $chan2 :: (as \times ifs \times as \times ifs) \Rightarrow ('aahi, 'uhi, 'ainfo)\ pkt2\ set$
  $loc2 :: as \Rightarrow ('aahi, 'uhi, 'ainfo)\ pkt2\ set$

datatype $('aahi, 'uhi, 'ainfo)\ evt2 =$
    $evt\text{-}dispatch\text{-}int2\ as\ ('aahi, 'uhi, 'ainfo)\ pkt2$
  $|\ evt\text{-}recv2\ as\ ifs\ ('aahi, 'uhi, 'ainfo)\ pkt2$
  $|\ evt\text{-}send2\ as\ ifs\ ('aahi, 'uhi, 'ainfo)\ pkt2$
  $|\ evt\text{-}deliver2\ as\ ('aahi, 'uhi, 'ainfo)\ pkt2$
  $|\ evt\text{-}dispatch\text{-}ext2\ as\ ifs\ ('aahi, 'uhi, 'ainfo)\ pkt2$
  $|\ evt\text{-}observe2\ ('aahi, 'uhi, 'ainfo)\ dp2\text{-}state$
  $|\ evt\text{-}skip2$

definition *soup2* **where** $soup2\ m\ s \equiv \exists x.\ m \in (loc2\ s)\ x \vee (\exists x.\ m \in (chan2\ s)\ x)$

declare *soup2-def* [*simp*]

2.4.1 Hop validation check, authorized segments, and path extraction.

First we define a locale that requires a number of functions. We will later extend this to a locale *dataplane-2*, which makes assumptions on how these functions operate. We separate the assumptions in order to make use of some auxiliary definitions defined in this locale.

locale *dataplane-2-defs* = *network-model* - *auth-seg0*
  for *auth-seg0* :: ($'ainfo \times 'aahi$ *ahi-scheme list*) *set* +
— *hf-valid-generic* is the check that every hop performs. Besides the hop's own field, the check may require access to its neighboring hop fields as well as on *ainfo*, *uinfo* and the entire sequence of hop fields. Note that this check should include checking the validity of the info fields. Depending on the directed vs. undirected setting, this check may only have access to specific fields.
  fixes *hf-valid-generic* :: $'ainfo \Rightarrow$ *msgterm*
    $\Rightarrow ('aahi, 'uhi)$ *HF list*
    $\Rightarrow ('aahi, 'uhi)$ *HF option*
    $\Rightarrow ('aahi, 'uhi)$ *HF*
    $\Rightarrow ('aahi, 'uhi)$ *HF option* $\Rightarrow$ *bool*
— *hfs-valid-prefix-generic* is the longest prefix of a given future path, such that *hf-valid-generic* passes for each hop field on the prefix.
  and *hfs-valid-prefix-generic* ::
    $'ainfo \Rightarrow$ *msgterm*
     $\Rightarrow ('aahi, 'uhi)$ *HF list*
     $\Rightarrow ('aahi, 'uhi)$ *HF option*
     $\Rightarrow ('aahi, 'uhi)$ *HF list*
     $\Rightarrow ('aahi, 'uhi)$ *HF option* $\Rightarrow ('aahi, 'uhi)$ *HF list*
— We need *checkInfo* only for the empty segment (ainfo, []) since according to the definition any such ainfo will be contained in the intruder knowledge. With checkInfo we can restrict this.
  and *checkInfo* :: $'ainfo \Rightarrow$ *bool*
— *extr* extracts from a given hop validation field (*HVF hf*) the entire authenticated future path that is embedded in the HVF.
  and *extr* :: *msgterm* $\Rightarrow 'aahi$ *ahi-scheme list*
— *extr-ainfo* extracts the authenticated info field (ainfo) from a given hop validation field.
  and *extr-ainfo* :: *msgterm* $\Rightarrow 'ainfo$
— *ik-auth-ainfo* extracts what msgterms the intruder can learn from analyzing a given authenticated info field. Note that currently we do not have a similar function for the unauthenticated info field *uinfo*. Protocols should thus only use that field with terms that the intruder can already synthesize (such as Numbers).
  and *ik-auth-ainfo* :: $'ainfo \Rightarrow$ *msgterm*
— *ik-hf* extracts what msgterms the intruder can learn from analyzing a given hop field; for instance, the hop validation field HVF hf and the segment identifier UHI hf.
  and *ik-hf* :: $('aahi, 'uhi)$ *HF* $\Rightarrow$ *msgterm set*

— We require that *hfs-valid-prefix-generic* behaves as expected, i.e., that it implements the check mentioned above.
assumes *prefix-hfs-valid-prefix-generic*:
  prefix (*hfs-valid-prefix-generic ainfo uinfo pas pre fut nxt*) *fut*
    and *cons-hfs-valid-prefix-generic*:
  $[\![$*hf-valid-generic ainfo uinfo hfs* (*head pas*) *hf1* (*head fut*); *hfs* = (*rev pas*)@*hf1* #*fut*$]\!]$
$\Longrightarrow$ *hfs-valid-prefix-generic ainfo uinfo pas* (*head pas*) (*hf1* # *fut*) *None* =
    hf1 # (*hfs-valid-prefix-generic ainfo uinfo* (*hf1*#*pas*) (*Some hf1*) *fut None*)
begin

Auxiliary definitions and lemmas

This function maps hop fields of the dp2 format to hop fields of dp0 format.

definition *AHIS* :: $('aahi, 'uhi)$ *HF list* $\Rightarrow 'aahi$ *ahi-scheme list* **where**
  AHIS hfs $\equiv$ *map AHI hfs*

declare *AHIS-def* [*simp*]

fun *extr-from-hd* :: ($'aahi$, $'uhi$) *HF list* ⇒ $'aahi$ *ahi-scheme list* **where**
  extr-from-hd (*hf*#*xs*) = *extr* (*HVF hf*)
| *extr-from-hd* - = []

fun *extr-ainfoHd* **where**
  extr-ainfoHd (*hf*#*xs*) = *Some* (*extr-ainfo* (*HVF hf*))
| *extr-ainfoHd* - = *None*

lemma *prefix-AHIS*:
prefix x1 x2 ⟹ *prefix* (*AHIS x1*) (*AHIS x2*)
by (*induction x1 arbitrary*: *x2 rule*: *list.induct*)
(*auto simp add*: *prefix-def*)

lemma *AHIS-set*: *hf* ∈ *set* (*AHIS l*) ⟹ ∃ *hfc* . *hfc* ∈ *set l* ∧ *hf* = *AHI hfc*
by(*induction l*) *auto*

lemma *AHIS-set-rev*: (|*AHI* = *ahi*, *UHI* = *uhi*, *HVF* = *x*|) ∈ *set hfs* ⟹ *ahi* ∈ *set* (*AHIS hfs*)
by(*induction hfs*, *auto*)

fun *pkt2to1* :: ($'aahi$, $'uhi$, $'ainfo$) *pkt2* ⇒ ($'aahi$, $'ainfo$) *pkt1* **where**
pkt2to1 (| *AInfo* = *ainfo*, *UInfo* = *uinfo*, *past* = *pas*, *future* = *fut*, *history* = *hist* |) =
(| *pkt0.AInfo* = *ainfo*,
past = *AHIS pas*,
future = *AHIS* (*hfs-valid-prefix-generic ainfo uinfo pas* (*head pas*) *fut None*),
history = *hist*|)

abbreviation *AHIo* :: ($'aahi$, $'uhi$) *HF option* ⇒ $'aahi$ *ahi-scheme option* **where**
AHIo ≡ *map-option AHI*

Authorized segments

Main definition of authorized up-segments. Makes sure that:

- the segment is rooted
- the segment is terminated
- the segment has matching interfaces
- the projection to AS owners is an authorized segment in the abstract model.

definition *auth-seg2* :: ($'ainfo$ × ($'aahi$, $'uhi$) *HF list*) *set* **where**
auth-seg2 ≡ ({(*ainfo*, *l*) | *ainfo l uinfo* . *hfs-valid-prefix-generic ainfo uinfo* [] *None l None* = *l*
∧ *checkInfo ainfo*
∧ (*ainfo*, *AHIS l*) ∈ *auth-seg0*})

lemma *auth-seg20*:
(*x*, *y*) ∈ *auth-seg2* ⟹ (*x*, *AHIS y*) ∈ *auth-seg0* **by**(*auto simp add*: *auth-seg2-def*)

lemma *pfragment-auth-seg20*:
pfragment ainfo l auth-seg2 $\Longrightarrow$ *pfragment ainfo* (*AHIS l*) *auth-seg0*
by (*auto 3 4 simp add*: *pfragment-def map-append dest*: *auth-seg20*)

lemma *pfragment-auth-seg20'*:
$[\![$*pfragment ainfo l auth-seg2*; *l'* = *AHIS l*$]\!]$ $\Longrightarrow$ *pfragment ainfo l' auth-seg0*
using *pfragment-auth-seg20* **by** *blast*

This is a shortcut to denote adding a message to a local channel.

definition
dp2-add-loc2 ::
('*aahi*, '*uhi*, '*ainfo*, '*more*) *dp2-state-scheme* ⇒
('*aahi*, '*uhi*, '*ainfo*, '*more*) *dp2-state-scheme* ⇒ *as* ⇒ ('*aahi*, '*uhi*, '*ainfo*) *pkt2* ⇒ *bool*
where
dp2-add-loc2 s s' asid pkt ≡ *s'* = *s*(|*loc2* := (*loc2 s*)(*asid* := *loc2 s asid* ∪ {*pkt*})|)

This is a shortcut to denote adding a message to an inter-AS channel. Note that it requires the link to exist.

definition
dp2-add-chan2 ::
('*aahi*, '*uhi*, '*ainfo*, '*more*) *dp2-state-scheme* ⇒ ('*aahi*, '*uhi*, '*ainfo*, '*more*) *dp2-state-scheme*
⇒ *as* ⇒ *ifs* ⇒ ('*aahi*, '*uhi*, '*ainfo*) *pkt2* ⇒ *bool*
where
dp2-add-chan2 s s' a1 i1 pkt ≡
∃ *a2 i2* . *rev-link a1 i1* = (*Some a2*, *Some i2*) ∧
s' = *s*(|*chan2* := (*chan2 s*)((*a1*, *i1*, *a2*, *i2*) := *chan2 s* (*a1*, *i1*, *a2*, *i2*) ∪ {*pkt*})|)

This is a shortcut to denote receiving a message from an inter-AS channel. Note that it requires the link to exist.

definition
dp2-in-chan2 :: ('*aahi*, '*uhi*, '*ainfo*, '*more*) *dp2-state-scheme* ⇒ *as* ⇒ *ifs* ⇒ ('*aahi*, '*uhi*, '*ainfo*) *pkt2* ⇒ *bool*
where
dp2-in-chan2 s a1 i1 pkt ≡
∃ *a2 i2* . *rev-link a1 i1* = (*Some a2*, *Some i2*) ∧
pkt ∈ (*chan2 s*)(*a2*, *i2*, *a1*, *i1*)

lemmas *dp2-msgs* = *dp2-add-loc2-def dp2-add-chan2-def dp2-in-chan2-def*

end

2.4.2 Intruder Knowledge definition

print-locale *dataplane-2-defs*
locale *dataplane-2-ik-defs* = *dataplane-2-defs* - - - - *hf-valid-generic* - -
for *hf-valid-generic* :: '*ainfo* ⇒ *msgterm*
⇒ ('*aahi*, '*uhi*) *HF list*
⇒ ('*aahi*, '*uhi*) *HF option*
⇒ ('*aahi*, '*uhi*) *HF*
⇒ ('*aahi*, '*uhi*) *HF option* ⇒ *bool* +
— *ik-add* is Additional Intruder Knowledge, such as hop authenticators in EPIC L1.

fixes *ik-add* :: *msgterm set*
— *ik-oracle* is another type of additional Intruder Knowledge. We use it to model the attacker's ability to brute-force individual hop validation fields and segment identifiers.
and *ik-oracle* :: *msgterm set*
— As *ik-oracle* gives the attacker direct access to hop validation fields that could be used to break the property, we have to either restrict the scope of the property, or restrict the attacker such that he cannot use the oracle-obtained hop validation fields in packets whose path origin matches the path origin of the oracle query. We choose the latter approach and fix a predicate *no-oracle* that tells us if the oracle has not been queried for a path origin (ainfo, uinfo combination). This is a prophecy variable.
and *no-oracle* :: *'ainfo* ⇒ *msgterm* ⇒ *bool*
begin

This set should contain all terms that can be learned from analyzing a hop field, in particular the content of the HVF and UHI fields.

definition *ik-auth-hfs* :: *msgterm set* **where**
$ik\text{-}auth\text{-}hfs = \{t \mid t\ hf\ hfs\ ainfo.\ t \in ik\text{-}hf\ hf \wedge hf \in set\ hfs \wedge (ainfo,\ hfs) \in auth\text{-}seg2\}$

declare *ik-auth-hfs-def* [*simp*]

definition *ik* :: *msgterm set* **where**
$ik = ik\text{-}auth\text{-}hfs$
$\cup \{ik\text{-}auth\text{-}ainfo\ ainfo \mid ainfo\ hfs.\ (ainfo,\ hfs) \in auth\text{-}seg2\}$
$\cup\ Key`(macK`bad)$
$\cup\ ik\text{-}add$
$\cup\ ik\text{-}oracle$

definition *ik-pkt* :: (*'aahi*, *'uhi*, *'ainfo*) *pkt2* ⇒ *msgterm set* **where**
$ik\text{-}pkt\ m \equiv \{t \mid t\ hf.\ t \in ik\text{-}hf\ hf \wedge hf \in set\ (past\ m) \cup set\ (future\ m)\}$
$\cup \{ik\text{-}auth\text{-}ainfo\ ainfo \mid ainfo\ .\ ainfo = AInfo\ m\}$

Intruder knowledge. We make a simplifying assumption about the attacker's passive capabilities: In contrast to his ability to insert messages (which is restricted to the locality of ASes that are compromised, i.e. in the set 'bad', the attacker has global eavesdropping abilities. This simplifies modelling and does not make the proofs more difficult, while providing stronger guarantees. We will later prove that the Dolev-Yao closure of *ik-dyn* remains constant, i.e., the attacker does not learn anything new by observing messages on the network (see *Inv-inv-ik-dyn*).

definition *ik-dyn* :: (*'aahi*, *'uhi*, *'ainfo*, *'more*) *dp2-state-scheme* ⇒ *msgterm set* **where**
$ik\text{-}dyn\ s \equiv ik \cup (\bigcup\{ik\text{-}pkt\ m \mid m\ x\ .\ m \in loc2\ s\ x\}) \cup (\bigcup\{ik\text{-}pkt\ m \mid m\ x\ .\ m \in chan2\ s\ x\})$

lemma *ik-dyn-mono*: $[\![x \in ik\text{-}dyn\ s;\ \bigwedge m\ .\ soup2\ m\ s \Longrightarrow soup2\ m\ s']\!] \Longrightarrow x \in ik\text{-}dyn\ s'$
by (*auto simp add*: *ik-dyn-def*) *metis+*

lemma *ik-info*[*elim*]:
$(ainfo,\ hfs) \in auth\text{-}seg2 \Longrightarrow ik\text{-}auth\text{-}ainfo\ ainfo \in synth\ (analz\ ik)$
by(*auto simp add*: *ik-def*)

lemma *ik-ik-auth-hfs*: $t \in ik\text{-}auth\text{-}hfs \Longrightarrow t \in ik$ **by**(*auto simp add*: *ik-def*)

2.4.3 Events

This is an attacker event (but does not require the dispatching node to be compromised).

The attacker is allowed to send any message that he can derive from his intruder knowledge, except for messages whose path origin he has queried the oracle for.

definition
  dp2-dispatch-int
where
  $dp2\text{-}dispatch\text{-}int\ s\ m\ ainfo\ uinfo\ asid\ pas\ fut\ hist\ s' \equiv$
    $m = (\!|\ AInfo = ainfo,\ UInfo = uinfo,\ past = pas,\ future = fut,\ history = hist\ |\!) \wedge$
    $hist = [] \wedge$
    $ik\text{-}auth\text{-}ainfo\ ainfo \in synth\ (analz\ (ik\text{-}dyn\ s)) \wedge$
    $(\forall hf \in set\ fut \cup set\ pas\ .\ ik\text{-}hf\ hf \subseteq synth\ (analz\ (ik\text{-}dyn\ s))) \wedge$
    $no\text{-}oracle\ ainfo\ uinfo \wedge$
    — action: Update the state to include m
    $dp2\text{-}add\text{-}loc2\ s\ s'\ asid\ m$

definition
  dp2-recv
where
  $dp2\text{-}recv\ s\ m\ asid\ ainfo\ uinfo\ hf1\ downif\ pas\ fut\ hist\ s' \equiv$
    — guard: a packet with valid interfaces and valid validation fields is in the incoming channel.
    $m = (\!|\ AInfo = ainfo,\ UInfo = uinfo,\ past = pas,\ future = hf1\#fut,\ history = hist\ |\!) \wedge$
    $dp2\text{-}in\text{-}chan2\ s\ (ASID\ (AHI\ hf1))\ downif\ m \wedge$
    $DownIF\ (AHI\ hf1) = Some\ downif \wedge$
    $ASID\ (AHI\ hf1) = asid \wedge$
    $hf\text{-}valid\text{-}generic\ ainfo\ uinfo\ (rev(pas)@hf1\#fut)\ (head\ pas)\ hf1\ (head\ fut) \wedge$

    — action: Update local state to include message
    $dp2\text{-}add\text{-}loc2\ s\ s'\ asid\ m$

definition
  dp2-send
where
  $dp2\text{-}send\ s\ m\ asid\ ainfo\ uinfo\ hf1\ upif\ pas\ fut\ hist\ s' \equiv$
    — guard: forward the packet on the external channel and advance the path by one hop.
    $m = (\!|\ AInfo = ainfo,\ UInfo = uinfo,\ past = pas,\ future = hf1\#fut,\ history = hist\ |\!) \wedge$
    $m \in (loc2\ s)\ asid \wedge$
    $UpIF\ (AHI\ hf1) = Some\ upif \wedge$
    $ASID\ (AHI\ hf1) = asid \wedge$
    $hf\text{-}valid\text{-}generic\ ainfo\ uinfo\ (rev(pas)@hf1\#fut)\ (head\ pas)\ hf1\ (head\ fut) \wedge$

    — action: Update state to include modified message
    $dp2\text{-}add\text{-}chan2\ s\ s'\ asid\ upif\ (\!|$
        $AInfo = ainfo,$
        $UInfo = uinfo,$
        $past = hf1\ \#\ pas,$
        $future = fut,$
        $history = AHI\ hf1\ \#\ hist$
      $|\!)$

definition

dp2-deliver
where
dp2-deliver s m asid ainfo uinfo hf1 pas fut hist s' $\equiv$
$m = (\!|\ AInfo = ainfo,\ UInfo = uinfo,\ past = pas,\ future = hf1\#fut,\ history = hist\ |\!) \wedge$
$m \in (loc2\ s)\ asid \wedge$
$ASID\ (AHI\ hf1) = asid \wedge$
$fut = [] \wedge$
$hf\text{-}valid\text{-}generic\ ainfo\ uinfo\ (rev(pas)@hf1\#fut)\ (head\ pas)\ hf1\ (head\ fut) \wedge$

— action: Update state to include modified message
dp2-add-loc2 s s' asid
$(\!|$
$AInfo = ainfo,$
$UInfo = uinfo,$
$past = hf1\ \#\ pas,$
$future = [],$
$history = (AHI\ hf1)\ \#\ hist$
$|\!)$

This is an attacker event (but does not require the dispatching node to be compromised).

The attacker is allowed to send any message that he can derive from his intruder knowledge, except for messages whose path origin he has queried the oracle for.

definition
dp2-dispatch-ext
where
dp2-dispatch-ext s m asid ainfo uinfo upif pas fut hist s' $\equiv$
$m = (\!|\ AInfo = ainfo,\ UInfo = uinfo,\ past = pas,\ future = fut,\ history = hist\ |\!) \wedge$
$hist = [] \wedge$
$ik\text{-}auth\text{-}ainfo\ ainfo \in synth\ (analz\ (ik\text{-}dyn\ s)) \wedge$
$(\forall hf \in set\ fut \cup set\ pas\ .\ ik\text{-}hf\ hf \subseteq synth\ (analz\ (ik\text{-}dyn\ s))) \wedge$
$no\text{-}oracle\ ainfo\ uinfo \wedge$

— action
dp2-add-chan2 s s' asid upif m

2.4.4 Transition system

fun *dp2-trans* **where**
dp2-trans s (evt-dispatch-int2 asid m) s' $\longleftrightarrow$
$(\exists\ ainfo\ uinfo\ pas\ fut\ hist\ .\ dp2\text{-}dispatch\text{-}int\ s\ m\ ainfo\ uinfo\ asid\ pas\ fut\ hist\ s')$ |
dp2-trans s (evt-recv2 asid downif m) s' $\longleftrightarrow$
$(\exists\ ainfo\ uinfo\ hf1\ pas\ fut\ hist\ .\ dp2\text{-}recv\ s\ m\ asid\ ainfo\ uinfo\ hf1\ downif\ pas\ fut\ hist\ s')$ |
dp2-trans s (evt-send2 asid upif m) s' $\longleftrightarrow$
$(\exists\ ainfo\ uinfo\ hf1\ pas\ fut\ hist.\ dp2\text{-}send\ s\ m\ asid\ ainfo\ uinfo\ hf1\ upif\ pas\ fut\ hist\ s')$ |
dp2-trans s (evt-deliver2 asid m) s' $\longleftrightarrow$
$(\exists\ ainfo\ uinfo\ hf1\ pas\ fut\ hist.\ dp2\text{-}deliver\ s\ m\ asid\ ainfo\ uinfo\ hf1\ pas\ fut\ hist\ s')$ |
dp2-trans s (evt-dispatch-ext2 asid upif m) s' $\longleftrightarrow$
$(\exists\ ainfo\ uinfo\ pas\ fut\ hist\ .\ dp2\text{-}dispatch\text{-}ext\ s\ m\ asid\ ainfo\ uinfo\ upif\ pas\ fut\ hist\ s')$ |
dp2-trans s (evt-observe2 s'') s' $\longleftrightarrow s = s' \wedge s = s''$ |
dp2-trans s evt-skip2 s' $\longleftrightarrow s = s'$

definition *dp2-init* :: *('aahi, 'uhi, 'ainfo) dp2-state* **where**

dp2-init ≡ (|*chan2* = (λ-. {}), *loc2* = (λ-. {})|)

definition *dp2* :: (('*aahi*, '*uhi*, '*ainfo*) *evt2*, ('*aahi*, '*uhi*, '*ainfo*) *dp2-state*) *ES* **where**
dp2 ≡ (|
init = (=) *dp2-init*,
trans = *dp2-trans*
|)

lemmas *dp2-trans-defs* = *dp2-dispatch-int-def dp2-recv-def dp2-send-def dp2-deliver-def dp2-dispatch-ext-def*
lemmas *dp2-defs* = *dp2-def dp2-init-def dp2-trans-defs*

end

2.4.5 Assumptions of the parametrized model

We now list the assumptions of this parametrized model.

locale *dataplane-2* = *dataplane-2-ik-defs* - - - - - - - - - - *hf-valid-generic*
for *hf-valid-generic* :: '*ainfo* ⇒ *msgterm*
⇒ ('*aahi*, '*uhi*) *HF list*
⇒ ('*aahi*, '*uhi*) *HF option*
⇒ ('*aahi*, '*uhi*) *HF*
⇒ ('*aahi*, '*uhi*) *HF option* ⇒ *bool* +
assumes *ik-seg-is-auth*:
⟦⋀*hf* . *hf* ∈ *set hfs* ⟹ *ik-hf hf* ⊆ *synth* (*analz ik*); *ik-auth-ainfo ainfo* ∈ *synth* (*analz ik*);
nxt = *None*; *no-oracle ainfo uinfo*⟧
⟹ *pfragment ainfo*
(*ifs-valid-prefix prev'*
(*AHIS* (*hfs-valid-prefix-generic ainfo uinfo pas pre hfs nxt*))
None)
auth-seg0
begin

2.4.6 Mapping dp2 state to dp1 state

definition *R21* :: ('*aahi*, '*uhi*, '*ainfo*) *dp2-state* ⇒ ('*aahi*, '*ainfo*) *dp1-state* **where**
R21 s = (|*chan* = λ*x* . *pkt2to1* ‘ ((*chan2 s*) *x*),
loc = λ*x* . *pkt2to1* ‘ ((*loc2 s*) *x*)|)

lemma *auth-seg2-pfragment*:
⟦*pfragment ainfo* (*hf* # *fut*) *auth-seg2*; *AHIS* (*hf* # *fut*) = *x* # *xs*⟧
⟹ *pfragment ainfo* (*x* # *xs*) *auth-seg0*
by(*auto simp add*: *map-append auth-seg2-def pfragment-def*)

lemma *dp2-in-chan2-to-0E*[*elim*]:
⟦*dp2-in-chan2 s1 a1 i1 pkt2*; *pkt2to1 pkt2* = *pkt0*; *s0* = *R21 s1*⟧ ⟹
dp0-in-chan s0 a1 i1 pkt0
by(*auto simp add*: *R21-def dp2-in-chan2-def dp0-in-chan-def*)

lemma *dp2-in-loc2-to-0E*[*elim*]:
⟦*pkt2* ∈ (*loc2 s1*) *asid*; *pkt2to1 pkt2* = *pkt0*; *P* = *pkt2to1* ‘ *loc2 s1 asid*⟧ ⟹
pkt0 ∈ *P*
by *blast*

lemma *dp2-add-loc20E*:
$\llbracket$*dp2-add-loc2 s1 s1' asid p1*; *p0* = *pkt2to1 p1*; *s0* = *R21 s1*; *s0'* = *R21 s1'*$\rrbracket$
$\Longrightarrow$ *dp0-add-loc s0 s0' asid p0*
by(*auto simp add*: *R21-def dp2-add-loc2-def dp0-add-loc-def intro*!: *ext*)

lemma *dp2-add-chan20E*:
$\llbracket$*dp2-add-chan2 s1 s1' a1 i1 p1*; *p0* = *pkt2to1 p1*; *s0* = *R21 s1*; *s0'* = *R21 s1'*$\rrbracket$
$\Longrightarrow$ *dp0-add-chan s0 s0' a1 i1 p0*
by(*fastforce simp add*: *R21-def dp2-add-chan2-def dp0-add-chan-def*)

2.4.7 Invariant: Derivable Intruder Knowledge is constant under *dp2-trans*

Derivable Intruder Knowledge stays constant throughout all reachable states

definition *inv-ik-dyn* :: (*'aahi*, *'uhi*, *'ainfo*) *dp2-state* $\Rightarrow$ *bool* **where**
inv-ik-dyn s $\equiv$ *ik-dyn s* $\subseteq$ *synth* (*analz ik*)

lemma *inv-ik-dynI*:
assumes $\bigwedge$*t m x* . $\llbracket$*t* $\in$ *ik-pkt m*; *m* $\in$ *loc2 s x*$\rrbracket$ $\Longrightarrow$ *t* $\in$ *synth* (*analz ik*)
and $\bigwedge$*t m x* . $\llbracket$*t* $\in$ *ik-pkt m*; *m* $\in$ *chan2 s x*$\rrbracket$ $\Longrightarrow$ *t* $\in$ *synth* (*analz ik*)
shows *inv-ik-dyn s*
using *assms* **by**(*auto simp add*: *ik-dyn-def inv-ik-dyn-def*)

lemma *inv-ik-dynD*:
assumes *inv-ik-dyn s*
shows $\bigwedge$*t m x* . $\llbracket$*m* $\in$ *chan2 s x*; *t* $\in$ *ik-pkt m*$\rrbracket$ $\Longrightarrow$ *t* $\in$ *synth* (*analz ik*)
$\bigwedge$*t m x* . $\llbracket$*m* $\in$ *loc2 s x*; *t* $\in$ *ik-pkt m*$\rrbracket$ $\Longrightarrow$ *t* $\in$ *synth* (*analz ik*)
using *assms*
by(*auto simp add*: *ik-dyn-def inv-ik-dyn-def Union-eq dest*!: *subsetD intro*!: *exI*)

lemmas *inv-ik-dynE* = *inv-ik-dynD*[*elim-format*]

lemma *inv-ik-dyn-add-loc2*[*elim*!]:
$\llbracket$*dp2-add-loc2 s s' asid m*; *inv-ik-dyn s*; *ik-pkt m* $\subseteq$ *synth* (*analz ik*)$\rrbracket$
$\Longrightarrow$ *inv-ik-dyn s'*
by(*auto simp add*: *dp2-add-loc2-def intro*!: *inv-ik-dynI elim*: *inv-ik-dynE*)

lemma *inv-ik-dyn-add-chan2*[*elim*!]:
$\llbracket$*dp2-add-chan2 s s' a1 i1 m*; *inv-ik-dyn s*; *ik-pkt m* $\subseteq$ *synth* (*analz ik*)$\rrbracket$
$\Longrightarrow$ *inv-ik-dyn s'*
by(*auto simp add*: *dp2-add-chan2-def intro*!: *inv-ik-dynI elim*: *inv-ik-dynE*)

lemma *inv-ik-dyn-ik-dyn-ik*[*simp*]:
assumes *inv-ik-dyn s* **shows** *synth* (*analz* (*ik-dyn s*)) = *synth* (*analz ik*)
proof−
from *assms* **have** *ik-dyn s* $\subseteq$ *synth* (*analz ik*) **by**(*auto simp add*: *ik-dyn-def inv-ik-dyn-def*)
moreover have *ik* $\subseteq$ *ik-dyn s* **by**(*auto simp add*: *ik-dyn-def*)
ultimately show *?thesis* **using** *analz-idem analz-synth order-class.order.antisym sup.absorb2*
synth-analz-mono synth-idem synth-increasing **by** *metis*
qed

lemma *ik-hf-auth*: $\llbracket$*t* $\in$ *ik-hf hf*; (*ainfo*, *AHIS hfs*) $\in$ *auth-seg0*; *checkInfo ainfo*;

hfs-valid-prefix-generic ainfo uinfo [] *None hfs None* = *hfs*; *hf* ∈ *set hfs*⟧
$\Longrightarrow$ *t* ∈ *synth* (*analz ik*)
by(*rule synth-analz-self*)
(*auto simp add*: *ik-def auth-seg2-def intro*!: *exI*[*of - ainfo*])

lemma *Inv-inv-ik-dyn*: *reach dp2 s* $\Longrightarrow$ *inv-ik-dyn s*
proof(*induction s rule*: *reach.induct*)
case (*reach-init s*)
then show *?case*
by (*auto simp add*: *inv-ik-dyn-def dp2-defs ik-dyn-def*)
next
case (*reach-trans s e s'*)
then show *?case*

proof(*simp add*: *dp2-def*, *elim dp2-trans.elims exE sym*[*of s*, *elim-format*] *sym*[*of s'*, *elim-format*], *simp-all*)
fix *m ainfo uinfo asid pas fut hist*
assume *inv-ik-dyn s dp2-dispatch-int s m ainfo uinfo asid pas fut hist s'*
then show *inv-ik-dyn s'*
by(*auto simp add*: *dp2-defs*)
(*auto simp add*: *ik-pkt-def inv-ik-dyn-ik-dyn-ik*)
next
fix *m asid ainfo uinfo hf1 downif pas fut hist*
assume *inv-ik-dyn s dp2-recv s m asid ainfo uinfo hf1 downif pas fut hist s'*
then show *inv-ik-dyn s'*
by(*auto simp add*: *dp2-defs dp2-in-chan2-def elim*: *inv-ik-dynE*)
next
fix *m asid ainfo uinfo upif pas fut hist*
assume *inv-ik-dyn s dp2-dispatch-ext s m asid ainfo uinfo upif pas fut hist s'*
then show *inv-ik-dyn s'*
by(*auto simp add*: *dp2-defs*)
(*auto simp add*: *ik-pkt-def inv-ik-dyn-ik-dyn-ik*)
qed(*auto simp add*: *dp2-defs ik-pkt-def elim*!: *inv-ik-dynE*)
qed

This lemma shows that our definition of *dp2-dispatch-int* also works for honest senders. All packets than an honest sender would send are authorized. According to the definition of the intruder knowledge, they are then also derivable from the intruder knowledge. Hence, an honest sender can send packets with authorized segments. However, the restriction on *no-oracle* remains.

lemma *dp2-dispatch-int-also-works-for-honest*:
⟦*pfragment ainfo fut auth-seg2*; *reach dp2 s*; *pas* = []⟧ $\Longrightarrow$
ik-auth-ainfo ainfo ∈ *synth* (*analz* (*ik-dyn s*)) ∧
(∀ *hf* ∈ *set fut* ∪ *set pas* . *ik-hf hf* ⊆ *synth* (*analz* (*ik-dyn s*)))
using *Inv-inv-ik-dyn*
apply(*auto simp add*: *inv-ik-dyn-ik-dyn-ik*)
apply(*auto simp add*: *auth-seg2-def*)
apply(*auto elim*!: *pfragmentE*)
by (*metis AHIS-def UnCI ik-hf-auth map-append set-append*)

2.4.8 Refinement proof

fun π_2 :: ('aahi, 'uhi, 'ainfo) evt2 ⇒ ('aahi, 'ainfo) evt0 **where**
π_2 (evt-dispatch-int2 asid m) = evt-dispatch-int0 asid (pkt2to1 m)
| π_2 (evt-recv2 asid downif m) = evt-recv0 asid downif (pkt2to1 m)
| π_2 (evt-send2 asid upif m) = evt-send0 asid upif (pkt2to1 m)
| π_2 (evt-deliver2 asid m) = evt-deliver0 asid (pkt2to1 m)
| π_2 (evt-dispatch-ext2 asid upif m) = evt-dispatch-ext0 asid upif (pkt2to1 m)
| π_2 (evt-observe2 s) = evt-observe0 (R21 s)
| π_2 evt-skip2 = evt-skip0

lemma dp2-refines-dp1: dp2 $\sqsubseteq_{\pi_2}$ dp1
proof(rule simulate-ES-fun-with-invariant[**where** ?I = inv-ik-dyn, **where** ?h = R21])
fix s0
assume init dp2 s0
then show init dp1 (R21 s0)
by(auto simp add: R21-def dp1-defs dp2-defs)
next
fix s e s'
assume dp2: s−e→ s' **and** inv-ik-dyn s
then show dp1: R21 s−π_2 e→ R21 s'
proof(auto simp add: dp2-def elim!: dp2-trans.elims)
fix m ainfo uinfo asid hf pas fut hist
assume dp2-dispatch-int s m ainfo uinfo asid pas fut hist s'
then show dp1: R21 s−evt-dispatch-int0 asid (pkt2to1 m)→ R21 s'
by(auto simp add: dp1-defs dp2-defs ‹inv-ik-dyn s› simp del: AHIS-def
intro!: ik-seg-is-auth elim!: dp2-add-loc20E)
next
fix m asid ainfo uinfo hf1 downif pas fut hist
assume dp2-recv s m asid ainfo uinfo hf1 downif pas fut hist s'
then show dp1: R21 s−evt-recv0 asid downif (pkt2to1 m)→ R21 s'
apply(auto simp add: TW.takeW-split-tail dp1-defs dp2-defs
elim!: dp2-in-chan2-to-0E dp2-add-loc20E intro: head.cases[**where** ?x=fut]
intro!: exI[of - AHI hf1] exI[of - AHIS (hfs-valid-prefix-generic ainfo uinfo (hf1#pas)
(Some hf1) fut None)])
apply(drule cons-hfs-valid-prefix-generic) **apply** auto
apply(drule cons-hfs-valid-prefix-generic) **by** auto
next
fix m asid ainfo uinfo hf1 upif pas fut hist
assume dp2-send s m asid ainfo uinfo hf1 upif pas fut hist s'
then show dp1: R21 s−evt-send0 asid upif (pkt2to1 m)→ R21 s'
using cons-hfs-valid-prefix-generic
by(auto simp add: dp1-defs dp2-defs TW.takeW-split-tail R21-def elim!: dp2-add-chan20E)
next
fix m asid ainfo uinfo hf1 pas fut hist
assume dp2-deliver s m asid ainfo uinfo hf1 pas fut hist s'
then show dp1: R21 s−evt-deliver0 asid (pkt2to1 m)→ R21 s'
apply(auto simp add: R21-def TW.takeW.simps TW.takeW-split-tail dp1-defs dp2-defs
elim!: dp2-add-loc20E intro: head.cases[**where** ?x=fut] intro!: exI[of - AHI hf1])
using prefix-hfs-valid-prefix-generic cons-hfs-valid-prefix-generic head.simps(1) prefix-Nil
proof −
assume a1: hf-valid-generic ainfo uinfo (rev pas @ [hf1]) (head pas) hf1 None
have hfs-valid-prefix-generic ainfo uinfo (hf1 # pas) (Some hf1) [] None = []

by (*meson prefix-Nil prefix-hfs-valid-prefix-generic*)
then show *map AHI* (*hfs-valid-prefix-generic ainfo uinfo pas* (*head pas*) [*hf1*] *None*) = [*AHI hf1*]
using *a1* **by** (*simp add*: *cons-hfs-valid-prefix-generic*)
qed *blast*
next
fix *m asid ainfo uinfo upif pas fut hist*
assume *dp2-dispatch-ext s m asid ainfo uinfo upif pas fut hist s′*
then show *dp1*: *R21 s*−*evt-dispatch-ext0 asid upif* (*pkt2to1 m*)→ *R21 s′*
by(*auto simp add*: *dp1-defs dp2-defs* ‹*inv-ik-dyn s*› *simp del*: *AHIS-def intro*!: *ik-seg-is-auth elim*!: *dp2-add-chan20E*)
qed(*auto simp add*: *R21-def dp2-defs dp1-defs*)
next
fix *s*
show *reach dp2 s* ⟶ *inv-ik-dyn s* **using** *Inv-inv-ik-dyn* **by** *blast*
qed

2.4.9 Property preservation

The following property is weaker than *TR-auth* in that it does not include the future path. However, this is inconsequential, since we only included the future path in order for the original invariant to be inductive. The actual path authorization property only requires the history to be authorized. We remove the future path for clarity, as including it would require us to also restrict it using the interface- and cryptographic valid-prefix functions.

definition *auth-path2* :: (*′aahi*, *′uhi*, *′ainfo*) *pkt2* ⇒ *bool* **where**
auth-path2 m ≡ *pfragment* (*AInfo m*) (*rev* (*history m*)) *auth-seg0*

abbreviation *TR-auth2-hist* :: (*′aahi*, *′uhi*, *′ainfo*) *evt2 list set* **where** *TR-auth2-hist* ≡
{τ | τ . ∀ *s m* . *evt-observe2 s* ∈ *set* τ ∧ *soup2 m s* ⟶ *auth-path2 m*}

lemma *evt-observe2-0*:
evt-observe2 s ∈ *set* τ ⟹ *evt-observe0* (*R10* (*R21 s*)) ∈ ($\lambda x.\ \pi_1\ (\pi_2\ x)$) ‘ *set* τ
by *force*

declare *soup2-def* [*simp del*]
declare *soup-def* [*simp del*]

lemma *loc2to0*: ⟦*mc* ∈ *loc2 sc x*; *sa* = *R10* (*R21 sc*); *ma* = *pkt1to0loc* (*pkt2to1 mc*)⟧ ⟹ *ma* ∈ *loc sa x*
using *R10-def R21-def* **by** *simp*

lemma *chan2to0*: ⟦*mc* ∈ *chan2 sc* (*a1*, *i1*, *a2*, *i2*); *sa* = *R10* (*R21 sc*); *ma* = *pkt1to0chan a1 i1* (*pkt2to1 mc*)⟧
⟹ *ma* ∈ *chan sa* (*a1*, *i1*, *a2*, *i2*)
using *R10-def R21-def* **by** *simp*

lemma *loc2to0-auth*:
⟦*mc* ∈ *loc2 sc x*; *sa* = *R10* (*R21 sc*); *ma* = *pkt1to0loc* (*pkt2to1 mc*); *auth-path ma*⟧ ⟹ *auth-path2 mc*
apply(*auto simp add*: *R10-def R21-def auth-path-def auth-path2-def elim*!: *pfragmentE*)
subgoal for *zs1 zs2*
by(*cases mc*)

(*auto intro*!: *pfragmentI*[*of* - *zs1* - *pkt0.future* (*pkt1to0loc* (*pkt2to1 mc*)) @ *zs2*])
done

lemma *chan2to0-auth*:
$\llbracket$*mc* $\in$ *chan2 sc* (*a1*, *i1*, *a2*, *i2*); *sa* = *R10* (*R21 sc*); *ma* = *pkt1to0chan a1 i1* (*pkt2to1 mc*); *auth-path ma*$\rrbracket \Longrightarrow$ *auth-path2 mc*
apply(*auto simp add*: *R10-def R21-def auth-path-def auth-path2-def elim*!: *pfragmentE*)
subgoal for *zs1 zs2*
by(*cases mc*)
(*auto intro*!: *pfragmentI*[*of* - *zs1* - *pkt0.future* (*pkt1to0chan a1 i1* (*pkt2to1 mc*)) @ *zs2*])
done

lemma *tr2-satisfies-pathauthorization*: *dp2* $\models_{ES}$ *TR-auth2-hist*
apply(*rule property-preservation*[**where** π=π_1 *o* π_2, **where** *E*=*dp2*, **where** *F*=*dp0*, **where** *P*=*TR-auth*])
using *dp2-refines-dp1 dp1-refines-dp0 sim-ES-trans* **apply** *blast*
using *tr0-satisfies-pathauthorization* **apply** *blast*
apply (*auto simp del*: *soup2-def*)
subgoal for τ *s m*
apply(*auto elim*!: *allE*[*of* - *R10* (*R21 s*)]) **apply** *force*
apply(*auto simp add*: *soup2-def*)
subgoal
apply(*frule loc2to0-auth*) **apply**(*auto simp add*: *inv-auth-def elim*!: *allE*)
by (*meson loc2to0 soup-def*)
subgoal
apply(*frule chan2to0-auth*) **apply**(*auto simp add*: *inv-auth-def elim*!: *allE*)
by (*meson chan2to0 soup-def*)
done
done

definition *inv-detect2* :: ('*aahi*, '*uhi*, '*ainfo*) *dp2-state* $\Rightarrow$ *bool* **where**
inv-detect2 s $\equiv \forall$ *m* . *soup2 m s* $\longrightarrow$ *prefix* (*history m*) (*AHIS* (*past m*))

abbreviation *TR-detect2* **where** *TR-detect2* $\equiv$ {τ | τ . $\forall$ *s* . *evt-observe2 s* $\in$ *set* τ $\longrightarrow$ *inv-detect2 s*}

lemma *tr2-satisfies-detectability*: *dp2* $\models_{ES}$ *TR-detect2*
apply(*rule property-preservation*[**where** π=π_1 *o* π_2, **where** *E*=*dp2*, **where** *F*=*dp0*, **where** *P*=*TR-detect*])
using *dp2-refines-dp1 dp1-refines-dp0 sim-ES-trans* **apply** *blast*
using *tr0-satisfies-detectability* **apply** *blast*
apply (*auto simp add*: *inv-detect2-def*)
subgoal for τ *s m*
apply(*auto simp add*: *soup2-def inv-detect-def*)
apply(*auto elim*!: *allE*[*of* - *R10* (*R21 s*)])
subgoal using *evt-observe2-0* **by** *blast*
subgoal
apply(*auto elim*!: *allE*[*of* - (*pkt1to0loc* (*pkt2to1 m*))])
using *loc2to0 soup-def* **apply** *blast*
apply(*cases m*) **by** *auto*
subgoal using *evt-observe2-0* **by** *blast*
subgoal for *a1 i1*
apply(*auto elim*!: *allE*[*of* - (*pkt1to0chan a1 i1* (*pkt2to1 m*))])
using *chan2to0 soup-def* **apply** *blast*

apply(*cases m*) **by** *auto*
done
done

end
end

2.5 Network Assumptions used for authorized segments.

theory *Network-Assumptions*
imports
Network-Model
begin

locale *network-assums-generic* = *network-model* - *auth-seg0* **for**
auth-seg0 :: (*'ainfo* × *'aahi ahi-scheme list*) *set* +
assumes
— All authorized segments have valid interfaces
ASM-if-valid: (*info*, *l*) ∈ *auth-seg0* ⟹ *ifs-valid-None l* **and**
— All authorized segments are rooted, i.e., they start with None
ASM-empty [*simp*, *intro*!]: (*info*, []) ∈ *auth-seg0* **and**
ASM-rooted: (*info*, *l*) ∈ *auth-seg0* ⟹ *rooted l* **and**
ASM-terminated: (*info*, *l*) ∈ *auth-seg0* ⟹ *terminated l*

locale *network-assums-undirect* = *network-assums-generic* - - +
assumes
ASM-adversary: ⟦⋀*hf*. *hf* ∈ *set hfs* ⟹ *ASID hf* ∈ *bad*⟧ ⟹ (*info*, *hfs*) ∈ *auth-seg0*

locale *network-assums-direct* = *network-assums-generic* - - +
assumes
ASM-singleton: ⟦*ASID hf* ∈ *bad*⟧ ⟹ (*info*, [*hf*]) ∈ *auth-seg0* **and**
ASM-extension: ⟦(*info*, *hf2*#*ys*) ∈ *auth-seg0*; *ASID hf2* ∈ *bad*; *ASID hf1* ∈ *bad*⟧
⟹ (*info*, *hf1*#*hf2*#*ys*) ∈ *auth-seg0* **and**
ASM-modify: ⟦(*info*, *hf*#*ys*) ∈ *auth-seg0*; *ASID hf* = *a*; *ASID hf'* = *a*; *UpIF hf'* = *UpIF hf*; *a* ∈ *bad*⟧
⟹ (*info*, *hf'*#*ys*) ∈ *auth-seg0* **and**
ASM-cutoff: ⟦(*info*, *zs*@*hf*#*ys*) ∈ *auth-seg0*; *ASID hf* = *a*; *a* ∈ *bad*⟧ ⟹ (*info*, *hf*#*ys*) ∈ *auth-seg0*
begin

lemma *auth-seg0-non-empty* [*simp*, *intro*!]: *auth-seg0* ≠ {}
by *auto*

lemma *auth-seg0-non-empty-frag* [*simp*, *intro*!]: ∃ *info* . *pfragment info* [] *auth-seg0*
apply(*auto simp add*: *pfragment-def*)
by (*metis append-Nil2 ASM-empty*)

This lemma applies the extendability assumptions on *auth-seg0* to pfragments of *auth-seg0*.

lemma *extend-pfragment0*:
assumes *pfragment ainfo* (*hf2*#*xs*) *auth-seg0*
assumes *ASID hf1* ∈ *bad*
assumes *ASID hf2* ∈ *bad*
shows *pfragment ainfo* (*hf1*#*hf2*#*xs*) *auth-seg0*
using *assms*
by(*auto intro*!: *pfragmentI*[*of* - [] - -] *elim*!: *pfragmentE intro*: *ASM-cutoff intro*!: *ASM-extension*)

This lemma shows that the above assumptions imply that of the undirected setting

lemma ⟦⋀*hf*. *hf* ∈ *set hfs* ⟹ *ASID hf* ∈ *bad*⟧ ⟹ (*info*, *hfs*) ∈ *auth-seg0*
apply(*induction hfs*)
using *ASM-empty* **apply** *blast*
subgoal for *a hfs*

apply(*cases hfs*)
by(*auto intro*!: *ASM-singleton ASM-extension*)
done

end
end

2.6 Parametrized dataplane protocol for directed protocols

This is an instance of the *Parametrized-Dataplane-2* model, specifically for protocols that authorize paths in an undirected fashion. We specialize the *hf-valid-generic* check to a still parametrized, but more concrete *hf-valid* check. The rest of the parameters remain abstract until a later instantiation with a concrete protocols (see the instances directory).

While both the models for undirected and directed protocols import assumptions from the theory *Network-Assumptions*, they differ in strength: the assumptions made by undirected protocols are strictly weaker, since the entire forwarding path is authorized by each AS, and not only the future path from the perspective of each AS. In addition, the specific conditions that instances have to verify differs between the undirected and the directed setting (compare the locales *dataplane-3-undirected* and *dataplane-3-directed*).

This explains the need to split up the verification of the attacker event into two theories. Despite the differences that concrete protocols may exhibit, these two theories suffice to show the crux of the refinement proof. The instances merely have to show a set of static conditions

theory *Parametrized-Dataplane-3-directed*
  imports
    Parametrized-Dataplane-2 Network-Assumptions
begin

2.6.1 Hop validation check, authorized segments, and path extraction.

First we define a locale that requires a number of functions. We will later extend this to a locale *dataplane-3-directed*, which makes assumptions on how these functions operate. We separate the assumptions in order to make use of some auxiliary definitions defined in this locale.

locale *dataplane-3-directed-defs = network-assums-direct - - - auth-seg0*
  for *auth-seg0* :: (*'ainfo* × *'aahi ahi-scheme list*) *set* +
— *hf-valid* is the check that every hop performs on its own and neighboring hop fields as well as on ainfo and uinfo. Note that this includes checking the validity of the info fields. Right now, we have a restriction in the model that this check can not depend on the previous hop field (see *COND-hf-valid-no-prev*).
  fixes *hf-valid* :: *'ainfo* ⇒ *msgterm*
    ⇒ (*'aahi*, *'uhi*) *HF option*
    ⇒ (*'aahi*, *'uhi*) *HF*
    ⇒ (*'aahi*, *'uhi*) *HF option* ⇒ *bool*
— We need checkInfo only for the empty segment (ainfo, []) since according to the definition any such ainfo will be contained in the intruder knowledge. With checkInfo we can restrict this.
  and *checkInfo* :: *'ainfo* ⇒ *bool*
— extr extracts from a given hop validation field (HVF hf) the entire authenticated future path that is embedded in the HVF.
  and *extr* :: *msgterm* ⇒ *'aahi ahi-scheme list*
— *extr-ainfo* extracts the authenticated info field (ainfo) from a given hop validation field.
  and *extr-ainfo* :: *msgterm* ⇒ *'ainfo*
— *ik-auth-ainfo* extracts what msgterms the intruder can learn from analyzing a given authenticated info field. Note that currently we do not have a similar function for the unauthenticated info field *uinfo*. Protocols should thus only use that field with terms that the intruder can already synthesize (such as Numbers).
  and *ik-auth-ainfo* :: *'ainfo* ⇒ *msgterm*

— *ik-hf* extracts what msgterms the intruder can learn from analyzing a given hop field; for instance, the hop validation field HVF hf and the segment identifier UHI hf.
and *ik-hf* :: ('*aahi*, '*uhi*) *HF* ⇒ *msgterm set*
begin

abbreviation *hf-valid-generic* :: '*ainfo* ⇒ *msgterm*
⇒ ('*aahi*, '*uhi*) *HF list*
⇒ ('*aahi*, '*uhi*) *HF option*
⇒ ('*aahi*, '*uhi*) *HF*
⇒ ('*aahi*, '*uhi*) *HF option* ⇒ *bool* **where**
hf-valid-generic ainfo uinfo pas pre hf nxt ≡ *hf-valid ainfo uinfo pre hf nxt*

definition *hfs-valid-prefix-generic* ::
'*ainfo* ⇒ *msgterm* ⇒ ('*aahi*, '*uhi*) *HF list* ⇒ ('*aahi*, '*uhi*) *HF option* ⇒ ('*aahi*, '*uhi*) *HF list* ⇒
('*aahi*, '*uhi*) *HF option* ⇒ ('*aahi*, '*uhi*) *HF list***where**
hfs-valid-prefix-generic ainfo uinfo pas pre fut nxt ≡
TW.takeW (λ *pre hf nxt* . *hf-valid ainfo uinfo pre hf nxt*) *pre fut nxt*

declare *hfs-valid-prefix-generic-def* [*simp*]

lemma *prefix-hfs-valid-prefix-generic*:
prefix (*hfs-valid-prefix-generic ainfo uinfo pas pre fut nxt*) *fut*
by(*auto intro*: *TW.takeW-prefix*)

lemma *cons-hfs-valid-prefix-generic*: *hf-valid-generic ainfo uinfo pas* (*head pas*) *hf1* (*head fut*)
⟹ *hfs-valid-prefix-generic ainfo uinfo pas* (*head pas*) (*hf1* # *fut*) *None* =
hf1 # (*hfs-valid-prefix-generic ainfo uinfo* (*hf1*#*pas*) (*Some hf1*) *fut None*)
apply(*auto simp only*: *TW.takeW-split-tail*[**where** *x*=*hf1*] *hfs-valid-prefix-generic-def*)
apply *auto*
apply (*simp add*: *TW.takeW.simps*(*1*))+
using *head-cons* **apply** *fastforce*
by (*metis head.simps*(*1*) *head-cons*)

sublocale *dataplane-2-defs* - - - *auth-seg0 hf-valid-generic hfs-valid-prefix-generic checkInfo extr extr-ainfo ik-auth-ainfo ik-hf*
apply *unfold-locales*
using *prefix-hfs-valid-prefix-generic cons-hfs-valid-prefix-generic* **by** *blast*+

abbreviation *hfs-valid* **where**
hfs-valid ainfo uinfo pre l nxt ≡ *TW.holds* (*hf-valid ainfo uinfo*) *pre l nxt*

abbreviation *hfs-valid-prefix* **where**
hfs-valid-prefix ainfo uinfo pre l nxt ≡ *TW.takeW* (*hf-valid ainfo uinfo*) *pre l nxt*

abbreviation *hfs-valid-None* **where**
hfs-valid-None ainfo uinfo l ≡ *hfs-valid ainfo uinfo None l None*

abbreviation *hfs-valid-None-prefix* **where**
hfs-valid-None-prefix ainfo uinfo l ≡ *hfs-valid-prefix ainfo uinfo None l None*

end

print-locale *dataplane-3-directed-defs*

locale *dataplane-3-directed-ik-defs* = *dataplane-3-directed-defs - - - - hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf* **for**
hf-valid :: *'ainfo* ⇒ *msgterm* ⇒ (*'aahi*, *'uhi*) *HF option* ⇒ (*'aahi*, *'uhi*) *HF* ⇒ (*'aahi*, *'uhi*) *HF option* ⇒ *bool*
and *checkInfo* :: *'ainfo* ⇒ *bool*
and *extr* :: *msgterm* ⇒ *'aahi ahi-scheme list*
and *extr-ainfo* :: *msgterm* ⇒ *'ainfo*
and *ik-auth-ainfo* :: *'ainfo* ⇒ *msgterm*
and *ik-hf* :: (*'aahi*, *'uhi*) *HF* ⇒ *msgterm set*
\+
— *ik-add* is Additional Intruder Knowledge, such as hop authenticators in EPIC L1.
fixes *ik-add* :: *msgterm set*
— *ik-oracle* is another type of additional Intruder Knowledge. We use it to model the attacker's ability to brute-force individual hop validation fields and segment identifiers.
and *ik-oracle* :: *msgterm set*
— As *ik-oracle* gives the attacker direct access to hop validation fields that could be used to break the property, we have to either restrict the scope of the property, or restrict the attacker such that he cannot use the oracle-obtained hop validation fields in packets whose path origin matches the path origin of the oracle query. We choose the latter approach and fix a predicate *no-oracle* that tells us if the oracle has not been queried for a path origin (ainfo, uinfo combination). This is a prophecy variable.
and *no-oracle* :: *'ainfo* ⇒ *msgterm* ⇒ *bool*
begin

lemma *auth-seg2-elem*: ⟦(*ainfo*, *hfs*) ∈ *auth-seg2*; *hf* ∈ *set hfs*⟧
⟹ ∃ *pre nxt uinfo* . *hf-valid ainfo uinfo pre hf nxt* ∧ *checkInfo ainfo* ∧ (*ainfo*, *AHIS hfs*) ∈ *auth-seg0*
by (*auto simp add*: *auth-seg2-def TW.holds-takeW-is-identity dest*!: *TW.holds-set-list*)

print-locale *dataplane-2-ik-defs*
sublocale *dataplane-2-ik-defs - - - - hfs-valid-prefix-generic checkInfo extr extr-ainfo ik-auth-ainfo ik-hf hf-valid-generic ik-add ik-oracle no-oracle*
by *unfold-locales*
end

2.6.2 Assumptions of the parametrized model

We now list the assumptions of this parametrized model.

print-locale *dataplane-3-directed-ik-defs*
locale *dataplane-3-directed* = *dataplane-3-directed-ik-defs - - - - hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf ik-add ik-oracle no-oracle*
for *hf-valid* :: *'ainfo* ⇒ *msgterm*
⇒ (*'aahi*, *'uhi*) *HF option*
⇒ (*'aahi*, *'uhi*) *HF*
⇒ (*'aahi*, *'uhi*) *HF option* ⇒ *bool*
and *checkInfo* :: *'ainfo* ⇒ *bool*
and *extr* :: *msgterm* ⇒ *'aahi ahi-scheme list*
and *extr-ainfo* :: *msgterm* ⇒ *'ainfo*
and *ik-auth-ainfo* :: *'ainfo* ⇒ *msgterm*
and *ik-hf* :: (*'aahi*, *'uhi*) *HF* ⇒ *msgterm set*
and *ik-add* :: *msgterm set*
and *ik-oracle* :: *msgterm set*
and *no-oracle* :: *'ainfo* ⇒ *msgterm* ⇒ *bool* +

— A valid validation field that is contained in ik corresponds to an authorized hop field. (The notable exceptions being oracle-obtained validation fields.) This relates the result of *ik-hf* to its argument. *ik-hf* has to produce a msgterm that is either unique for each given hop field x, or it is only produced by an 'equivalence class' of hop fields such that either all of the hop fields of the class are authorized, or none are. While the extr function (constrained by assumptions below) also binds the hop information to the validation field, it does so only for AHI and AInfo, but not for UHI.

assumes *COND-ik-hf*:
$[\![$*hf-valid ainfo uinfo pre hf nxt*; *ik-hf hf* $\subseteq$ *analz ik*; *ik-auth-ainfo ainfo* $\in$ *analz ik*;
no-oracle ainfo uinfo$]\!]$
$\Longrightarrow \exists$ *hfs* . *hf* $\in$ *set hfs* $\wedge$ (*ainfo*, *hfs*) $\in$ *auth-seg2*

— A valid validation field that can be synthesized from the initial intruder knowledge is already contained in the initial intruder knowledge if it belongs to an honest AS. This can be combined with the previous assumption.

and *COND-honest-hf-analz*:
$[\![$*ASID* (*AHI hf*) $\notin$ *bad*; *hf-valid ainfo uinfo pre hf nxt*; *ik-hf hf* $\subseteq$ *synth* (*analz ik*);
no-oracle ainfo uinfo$]\!]$
$\Longrightarrow$ *ik-hf hf* $\subseteq$ *analz ik*

— A valid info field that can be synthesized from the initial intruder knowledge is already contained in the initial intruder knowlege.

and *COND-ainfo-analz*:
$[\![$*hf-valid ainfo uinfo pre hf nxt*; *ik-auth-ainfo ainfo* $\in$ *synth* (*analz ik*)$]\!]$
$\Longrightarrow$ *ik-auth-ainfo ainfo* $\in$ *analz ik*

— Extracting the path from the validation field of the first hop field of some path l returns an extension of the AHI-level path of the valid prefix of l.

and *COND-path-prefix-extr*:
prefix (*AHIS* (*hfs-valid-prefix ainfo uinfo pre l nxt*))
(*extr-from-hd l*)

— Extracting the path from the validation field of the first hop field of a completely valid path l returns a prefix of the AHI-level path of l. Together with *prefix* (*AHIS* (*hfs-valid-prefix ?ainfo ?uinfo ?pre ?l ?nxt*)) (*extr-from-hd ?l*), this implies that extr of a completely valid path l is exactly the same AHI-level path as l (see lemma below).

and *COND-extr-prefix-path*:
$[\![$*hfs-valid ainfo uinfo pre l nxt*; *nxt* = *None*$]\!]$ $\Longrightarrow$ *prefix* (*extr-from-hd l*) (*AHIS l*)

— The validation check does not depend on the prev hop field. For up-segments this is fine, but this is an assumption we may eventually get rid off when we verify down-segments.

and *COND-hf-valid-no-prev*:
hf-valid ainfo uinfo pre hf nxt $\longleftrightarrow$ *hf-valid ainfo uinfo pre*$'$ *hf nxt*

— A valid hop field is only valid for one specific uinfo.

and *COND-hf-valid-uinfo*:
$[\![$*hf-valid ainfo uinfo pre hf nxt*; *hf-valid ainfo*$'$ *uinfo*$'$ *pre*$'$ *hf nxt*$'$$]\!]$
$\Longrightarrow$ *uinfo*$'$ = *uinfo*

begin

lemma *holds-path-eq-extr*:
$[\![$*hfs-valid ainfo uinfo pre l nxt*; *nxt* = *None*$]\!]$ $\Longrightarrow$ *extr-from-hd l* = *AHIS l*
using *COND-extr-prefix-path COND-path-prefix-extr*
by (*metis TW.holds-implies-takeW-is-identity prefix-order.eq-iff*)

2.6.3 Lemmas that are needed for the refinement proof

lemma *honest-hf-analz-subsetI*:
$[\![$*ASID* (*AHI hf*) $\notin$ *bad*; *hf-valid ainfo uinfo prev hf nxt*; *ik-hf hf* $\subseteq$ *synth* (*analz ik*);

no-oracle ainfo uinfo; $t \in$ *ik-hf hf*⟧
$\Longrightarrow t \in$ *analz ik*
using *COND-honest-hf-analz subsetI* **by** *blast*

lemma *extr-from-hd-eq*: $(l \neq [] \wedge l' \neq [] \wedge hd\ l = hd\ l') \vee (l = [] \wedge l' = []) \Longrightarrow$ *extr-from-hd* $l =$ *extr-from-hd* l'
apply (*cases* l)
apply *auto*
apply(*cases* l')
by *auto*

lemma *path-prefix-extr-l*:
⟦$hd\ l = hd\ l'$; $l' \neq []$⟧ $\Longrightarrow$ *prefix* (*AHIS* (*hfs-valid-prefix ainfo uinfo pre l nxt*))
(*extr-from-hd* l')
using *COND-path-prefix-extr extr-from-hd.elims list.sel*(*1*) *not-prefix-cases prefix-Cons prefix-Nil*
by *metis*

lemma *path-prefix-extr-l'*:
⟦$hd\ l = hd\ l'$; $l' \neq []$; $hf = hd\ l'$⟧ $\Longrightarrow$ *prefix* (*AHIS* (*hfs-valid-prefix ainfo uinfo pre l nxt*))
(*extr* (*HVF hf*))
using *COND-path-prefix-extr extr-from-hd.elims list.sel*(*1*) *not-prefix-cases prefix-Cons prefix-Nil*
by *metis*

lemma *pfrag-extr-auth*:
assumes $hf \in$ *set p* **and** (*ainfo*, *p*) $\in$ *auth-seg2*
shows *pfragment ainfo* (*extr* (*HVF hf*)) *auth-seg0*
proof –
obtain *uinfo* **where** *p-verified*: *hfs-valid-None ainfo uinfo p*
using *assms*(*2*) *auth-seg2-def TW.holds-takeW-is-identity* **by** *fastforce*
have $\exists$ *xs*. *suffix* (*hf* # *xs*) *p* **by** (*simp add*: *Cons-suffix-set assms*)
then obtain *xs* **where** *suf*: *suffix* (*hf* # *xs*) *p* **by** (*auto*)
then have *pfragment ainfo* (*hf* # *xs*) *auth-seg2* **using** *assms*(*2*)
apply –
by (*rule pfragment-suffix-self*[**where** *?l=p*], *simp-all*)
then have *frag*: *pfragment ainfo* (*AHIS* (*hf* # *xs*)) *auth-seg0*
by (*rule pfragment-auth-seg20*)

have $\exists$ *pre* . *hfs-valid ainfo uinfo pre* (*hf* # *xs*) *None* **using** *p-verified suf* **by** (*rule TW.holds-suffix*)
then have *pfragment ainfo* (*extr-from-hd* (*hf* # *xs*)) *auth-seg0*
using *holds-path-eq-extr*[*symmetric*] *frag* **by** *force*
then show *?thesis* **by** *simp*
qed

lemma *X-in-ik-is-auth*:
assumes *ik-hf hf1* $\subseteq$ *analz ik* **and** *ik-auth-ainfo ainfo* $\in$ *analz ik* **and** *no-oracle ainfo uinfo*
shows *pfragment ainfo* (*AHIS* (*hfs-valid-prefix ainfo uinfo*
pre
(*hf1* # *fut*)
nxt))
auth-seg0
proof –
let *?pFu* = *hf1* # *fut*

let *?takW* = (*hfs-valid-prefix ainfo uinfo pre ?pFu nxt*)
have *prefix* (*AHIS* (*hfs-valid-prefix ainfo uinfo pre ?takW* (*TW.extract* (*hf-valid ainfo uinfo*) *pre ?pFu nxt*)))
(*extr-from-hd ?takW*)
by(*auto simp add*: *COND-path-prefix-extr simp del*: *AHIS-def*)
then have *prefix* (*AHIS ?takW*) (*extr-from-hd ?takW*)
by(*simp add*: *TW.takeW-takeW-extract*)
moreover from *assms* **have** *pfragment ainfo* (*extr-from-hd ?takW*) *auth-seg0*
by (*auto simp add*: *TW.takeW-split-tail dest*!: *COND-ik-hf intro*: *pfrag-extr-auth*)
ultimately show *?thesis*
by(*auto intro*: *pfragment-prefix elim*!: *prefixE*)
qed

Fragment is extendable

makes sure that: the segment is terminated, i.e. the leaf AS's HF has Eo = None

fun *terminated2* :: ($'aahi$, $'uhi$) *HF list* ⇒ *bool* **where**
terminated2 (*hf*#*xs*) ⟷ *DownIF* (*AHI hf*) = *None* ∨ *ASID* (*AHI hf*) ∈ *bad*
| *terminated2* [] = *True*

lemma *terminated20*: *terminated* (*AHIS m*) ⟹ *terminated2 m* **by**(*induction m*, *auto*)

lemma *cons-snoc*: ∃ *y ys*. *x* # *xs* = *ys* @ [*y*]
by (*metis append-butlast-last-id rev.simps*(*2*) *rev-is-Nil-conv*)

lemma *terminated2-suffix*:
⟦*terminated2 l*; *l* = *zs* @ *x* # *xs*; *DownIF* (*AHI x*) ≠ *None*; *ASID* (*AHI x*) ∉ *bad*⟧ ⟹ ∃ *y ys*. *zs* = *ys* @ [*y*]
by(*cases zs*)
(*fastforce intro*: *cons-snoc*)+

lemma *attacker-modify-cutoff*: ⟦(*info*, *zs*@*hf*#*ys*) ∈ *auth-seg0*; *ASID hf* = *a*;
ASID hf' = *a*; *UpIF hf'* = *UpIF hf*; *a* ∈ *bad*; *ys'* = *hf'*#*ys*⟧ ⟹ (*info*, *ys'*) ∈ *auth-seg0*
by(*auto simp add*: *ASM-modify dest*: *ASM-cutoff*)

lemma *auth-seg2-ik-hf*[*elim*]: ⟦*x* ∈ *ik-hf hf*; *hf* ∈ *set hfs*; (*ainfo*, *hfs*) ∈ *auth-seg2* ⟧ ⟹ *x* ∈ *analz ik*
by(*auto 3 4 simp add*: *ik-def*)

This lemma proves that an attacker-derivable segment that starts with an attacker hop field, and has a next hop field which belongs to an honest AS, when restricted to its valid prefix, is authorized. Essentially this is the case because the hop field of the honest AS already contains an interface identifier DownIF that points to the attacker-controlled AS. Thus, there must have been some attacker-owned hop field on the original authorized path. Given the assumptions we make in the directed setting, the attacker can make take a suffix of an authorized path, such that his hop field is first on the path, and he can change his own hop field if his hop field is the first on the path, thus, that segment is also authorized.

lemma *fragment-with-Eo-Some-extendable*:
assumes *ik-hf hf2* ⊆ *synth* (*analz ik*)
and *ik-auth-ainfo ainfo* ∈ *synth* (*analz ik*)
and *ASID* (*AHI hf1*) ∈ *bad*
and *ASID* (*AHI hf2*) ∉ *bad*

and *hf-valid ainfo uinfo pre hf1* (*Some hf2*)
and *no-oracle ainfo uinfo*
shows
pfragment ainfo
(*ifs-valid-prefix pre′*
(*AHIS* (*hfs-valid-prefix ainfo uinfo*
pre
(*hf1* # *hf2* # *fut*)
None))
None)
auth-seg0
proof(*cases*)
assume *hf-valid ainfo uinfo* (*Some hf1*) *hf2* (*head fut*)
∧ *if-valid* (*Some* (*AHI hf1*)) (*AHI hf2*) (*AHIo* (*head fut*))

then have *hf2true*: *hf-valid ainfo uinfo* (*Some hf1*) *hf2* (*head fut*)
if-valid (*Some* (*AHI hf1*)) (*AHI hf2*) (*AHIo* (*head fut*)) **by** *blast+*
then have ∃ *hfs* . *hf2* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
using *assms* **by**(*auto intro*!: *COND-ik-hf honest-hf-analz-subsetI COND-ainfo-analz*)
then obtain *hfs uinfo′* **where** *hfs-def*:
hf2 ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2 hfs-valid-None ainfo uinfo′ hfs*
using *COND-ik-hf* **by**(*auto simp add*: *auth-seg2-def TW.holds-takeW-is-identity*)

have *termianted-hfs*: *terminated2 hfs*
using *hfs-def*(*2*) **by** (*auto simp add*: *auth-seg2-def ASM-terminated intro*: *terminated20*)

have ∃ *pref hf1′ ys* . *hfs* = *pref*@[*hf1′*]@(*hf2*#*ys*)
using *hf2true*(*2*) *assms*(*4*) *hfs-def*(*1*) *terminated2-suffix*
by(*fastforce dest*: *split-list intro*: *termianted-hfs*)
then obtain *pref hf1′ ys* **where** *hfs-unfold*: *hfs* = *pref*@[*hf1′*]@(*hf2*#*ys*) **by** *fastforce*

have *hf2-valid*: *hf-valid ainfo uinfo′* (*Some hf1′*) *hf2* (*head ys*)
and *hf1′true*: *hf-valid ainfo uinfo′* (*tail pref*) *hf1′* (*Some hf2*)
apply(*cases ys*)
using *hfs-def*(*3*)
by (*auto simp add*: *hfs-def hfs-unfold TW.holds-unfold-prelnil tail-snoc TW.holds.simps*(*1*)
elim!: *TW.holds-unfold-prexnxt′ intro*: *rev-exhaust*[**where** *?xs*=*pref*])

have *uinfo′-eq*: *uinfo′* = *uinfo*
using *hf2-valid hf2true*(*1*) **by**(*intro COND-hf-valid-uinfo*)

have *if-valid-hf2hf1′*: *if-valid* (*Some* (*AHI hf1′*)) (*AHI hf2*) (*head* (*AHIS ys*))
apply(*cases ys*)
using *assms*(*4*) *hfs-def*(*2*) *ASM-if-valid TW.holds-unfold-prexnxt′ TW.holds-unfold-prelnil*
by(*fastforce simp add*: *hfs-unfold auth-seg2-def*)+

have *pfragment ainfo* (*AHIS* (*hfs-valid-prefix ainfo uinfo*
None
(*hf1′* # *hf2* # *fut*)
None))
auth-seg0
apply(*rule X-in-ik-is-auth*)

```
  using hfs-def(1,2) assms(2,5,6) by(fastforce simp add: hfs-unfold ik-def intro!: COND-ainfo-analz)+

 then show ?thesis
   apply−
   apply(rule strip-ifs-valid-prefix)
   apply(erule pfragment-self-eq-nil)
   apply(auto simp add: TW.takeW-split-tail[where ?x=hf1'])
   using assms(3−5) hf2true(2) if-valid-hf2hf1' hf1'true
   apply(auto elim!: attacker-modify-cutoff[where ?hf'=AHI hf1]
    simp add: TW.takeW-split-tail COND-hf-valid-no-prev[where pre=Some hf1, where pre'=Some
hf1'])
   using COND-hf-valid-no-prev uinfo'-eq by blast+
next
 assume hf2false: ¬(hf-valid ainfo uinfo (Some hf1) hf2 (head fut)
                      ∧ if-valid (Some (AHI hf1)) (AHI hf2) (AHIo (head fut)))
 then show ?thesis
   apply(cases hf-valid ainfo uinfo pre hf1 (Some hf2) ∧ if-valid pre' (AHI hf1) (Some (AHI hf2)))
     subgoal apply(cases hf-valid ainfo uinfo (Some hf1) hf2 (head fut))
       subgoal using assms(3) by(auto simp add: TW.takeW-split-tail intro: ASM-singleton)
       subgoal apply(cases fut)
         using assms(3)
         by(auto simp add: TW.takeW-split-tail[where ?x=hf1] TW.takeW.simps
               intro: ASM-singleton intro!: strip-ifs-valid-prefix)
     done
  by auto(auto simp add: TW.takeW-split-tail[where ?x=hf1] TW.takeW-split-tail[where ?x=hf2]

                    TW.takeW.simps ASM-singleton assms(3) strip-ifs-valid-prefix)
qed
```

A1 and A2 collude to make a wormhole

We lift *extend-pfragment0* to DP2.

```
lemma extend-pfragment2:
 assumes pfragment ainfo
(ifs-valid-prefix (Some (AHI hf1))
(AHIS (hfs-valid-prefix ainfo uinfo
                    (Some hf1)
                    (hf2 # fut)
                    nxt))
            None)
                  auth-seg0
 assumes hf-valid ainfo uinfo pre hf1 (Some hf2)
 assumes ASID (AHI hf1) ∈ bad
 assumes ASID (AHI hf2) ∈ bad
 shows pfragment ainfo
(ifs-valid-prefix pre'
(AHIS (hfs-valid-prefix ainfo uinfo
                  pre
                  (hf1 # hf2 # fut)
                  nxt))
            None)
                auth-seg0
```

using *assms*
apply(*auto simp add*: *TW.takeW-split-tail*[**where** *?P=hf-valid ainfo uinfo*])
by(*auto simp add*: *TW.takeW-split-tail*[**where** *?P=if-valid*] *TW.takeW.simps*(*1*)
intro: *ASM-singleton extend-pfragment0 strip-ifs-valid-prefix*)

This is the central lemma that we need to prove to show the refinement between this model and dp1. It states: If an attacker can synthesize a segment from his knowledge, and does not use a path origin that was used to query the oracle, then the valid prefix of the segment is authorized. Thus, the attacker cannot create any valid but unauthorized segments.

lemma *ik-seg-is-auth*:
assumes $\bigwedge hf\ .\ hf \in set\ hfs \Longrightarrow ik\text{-}hf\ hf \subseteq synth\ (analz\ ik)$ **and**
$ik\text{-}auth\text{-}ainfo\ ainfo \in synth\ (analz\ ik)$ **and** $nxt = None$ **and** *no-oracle ainfo uinfo*
shows *pfragment ainfo*
(*ifs-valid-prefix prev′*
(*AHIS* (*hfs-valid-prefix ainfo uinfo pre hfs nxt*))
None)
auth-seg0
using *assms*
proof(*induction pre hfs nxt arbitrary*: *prev′ rule*: *TW.takeW.induct*[**where** *?Pa=hf-valid ainfo uinfo*])
case (*1* - -)
then show *?case* **using** *append-Nil ASM-empty pfragment-def Nil-is-map-conv TW.takeW.simps*(*1*)
by (*metis AHIS-def*)
next
case (*2 pre hf nxt*)
then show *?case*
proof(*cases*)
assume $ASID\ (AHI\ hf) \in bad$
then show *?thesis* **apply**−
by(*intro strip-ifs-valid-prefix*)
(*auto simp add*: *pfragment-def ASM-singleton TW.takeW-singleton intro*!: *exI*[*of* - []])
next
assume $ASID\ (AHI\ hf) \notin bad$
then show *?thesis* **using** *2 assms*
apply(*intro strip-ifs-valid-prefix*)
by (*auto simp add*: *2.prems*(*1*) *COND-ainfo-analz COND-honest-hf-analz X-in-ik-is-auth*
simp del: *AHIS-def*)
qed
next
case (*3 pre hf nxt*)
then show *?case*
by (*intro strip-ifs-valid-prefix*, *simp add*: *TW.takeW-singleton*)
next
case (*4 pre hf1 hf2 xs nxt*)
then show *?case*
proof(*cases*)
assume *hf1bad*: $ASID\ (AHI\ hf1) \in bad$
then show *?thesis*
proof(*cases*)
assume *hf2bad*: $ASID\ (AHI\ hf2) \in bad$
show *?thesis*
apply(*intro extend-pfragment2*)
apply(*intro 4*(*2*))

```
      using 4(1,3-5) ‹no-oracle ainfo uinfo› by(auto intro: hf1bad hf2bad)
    next
      assume ASID (AHI hf2) ∉ bad
      then show ?thesis
        using 4(1,3-6) hf1bad by(auto 3 4 intro!: fragment-with-Eo-Some-extendable
                                 simp del: hfs-valid-prefix-generic-def AHIS-def)
    qed
  next
    assume ASID (AHI hf1) ∉ bad
    then show ?thesis using 4(1,3-6)
      by(intro strip-ifs-valid-prefix)
        (auto intro!: X-in-ik-is-auth simp del: hfs-valid-prefix-generic-def AHIS-def
              dest: COND-honest-hf-analz COND-ainfo-analz)
  qed
next
  case 5
  then show ?case
    by(intro strip-ifs-valid-prefix, simp add: TW.takeW-two-or-more)
qed

sublocale dataplane-2 - - - - hfs-valid-prefix-generic - - - - - - - - hf-valid-generic
  apply unfold-locales
  using ik-seg-is-auth by simp

end
end
```

2.7 Parametrized dataplane protocol for undirected protocols

This is an instance of the *Parametrized-Dataplane-2* model, specifically for protocols that authorize paths in an undirected fashion. We specialize the *hf-valid-generic* check to a still parametrized, but more concrete *hf-valid* check. The rest of the parameters remain abstract until a later instantiation with a concrete protocols (see the instances directory).

While both the models for undirected and directed protocols import assumptions from the theory *Network-Assumptions*, they differ in strength: the assumptions made by undirected protocols are strictly weaker, since the entire forwarding path is authorized by each AS, and not only the future path from the perspective of each AS. In addition, the specific conditions that instances have to verify differs between the undirected and the directed setting (compare the locales *dataplane-3-undirected* and *dataplane-3-directed*).

This explains the need to split up the verification of the attacker event into two theories. Despite the differences that concrete protocols may exhibit, these two theories suffice to show the crux of the refinement proof. The instances merely have to show a set of static conditions

theory *Parametrized-Dataplane-3-undirected*
  imports
    Parametrized-Dataplane-2 Network-Assumptions
begin

2.7.1 Hop validation check, authorized segments, and path extraction.

First we define a locale that requires a number of functions. We will later extend this to a locale *dataplane-3-undirected*, which makes assumptions on how these functions operate. We separate the assumptions in order to make use of some auxiliary definitions defined in this locale.

locale *dataplane-3-undirected-defs* = *network-assums-undirect - - - auth-seg0*
  for *auth-seg0* :: $('ainfo \times 'aahi\ ahi\text{-}scheme\ list)\ set$ +
— *hf-valid* is the check that every hop performs on its own and neighboring hop fields as well as on ainfo and uinfo. Note that this includes checking the validity of the info fields. Right now, we have a restriction in the model that this check can not depend on the previous hop field (see *COND-hf-valid-no-prev*).
  fixes *hf-valid* :: $'ainfo \Rightarrow msgterm$
    $\Rightarrow ('aahi, 'uhi)\ HF\ list$
    $\Rightarrow ('aahi, 'uhi)\ HF$
    $\Rightarrow bool$
— We need checkInfo only for the empty segment (ainfo, []) since according to the definition any such ainfo will be contained in the intruder knowledge. With checkInfo we can restrict this.
  and *checkInfo* :: $'ainfo \Rightarrow bool$
— extr extracts from a given hop validation field (HVF hf) the entire authenticated future path that is embedded in the HVF.
  and *extr* :: $msgterm \Rightarrow 'aahi\ ahi\text{-}scheme\ list$
— *extr-ainfo* extracts the authenticated info field (ainfo) from a given hop validation field.
  and *extr-ainfo* :: $msgterm \Rightarrow 'ainfo$
— *ik-auth-ainfo* extracts what msgterms the intruder can learn from analyzing a given authenticated info field. Note that currently we do not have a similar function for the unauthenticated info field *uinfo*. Protocols should thus only use that field with terms that the intruder can already synthesize (such as Numbers).
  and *ik-auth-ainfo* :: $'ainfo \Rightarrow msgterm$

— *ik-hf* extracts what msgterms the intruder can learn from analyzing a given hop field; for instance, the hop validation field HVF hf and the segment identifier UHI hf.
and *ik-hf* :: ($'aahi$, $'uhi$) *HF* $\Rightarrow$ *msgterm set*
begin

abbreviation *hf-valid-generic* :: $'ainfo \Rightarrow$ *msgterm*
$\Rightarrow$ ($'aahi$, $'uhi$) *HF list*
$\Rightarrow$ ($'aahi$, $'uhi$) *HF option*
$\Rightarrow$ ($'aahi$, $'uhi$) *HF*
$\Rightarrow$ ($'aahi$, $'uhi$) *HF option* $\Rightarrow$ *bool* **where**
hf-valid-generic ainfo uinfo hfs pre hf nxt $\equiv$ *hf-valid ainfo uinfo hfs hf*

abbreviation *hfs-valid-prefix* **where**
hfs-valid-prefix ainfo uinfo pas fut $\equiv$ (*takeWhile* (λ*hf* . *hf-valid ainfo uinfo* (*rev*(*pas*)@*fut*) *hf*) *fut*)

definition *hfs-valid-prefix-generic* ::
$'ainfo \Rightarrow$ *msgterm* $\Rightarrow$ ($'aahi$, $'uhi$) *HF list* $\Rightarrow$ ($'aahi$, $'uhi$) *HF option* $\Rightarrow$ ($'aahi$, $'uhi$) *HF list* $\Rightarrow$
($'aahi$, $'uhi$) *HF option* $\Rightarrow$ ($'aahi$, $'uhi$) *HF list***where**
hfs-valid-prefix-generic ainfo uinfo pas pre fut nxt $\equiv$
hfs-valid-prefix ainfo uinfo pas fut

declare *hfs-valid-prefix-generic-def*[*simp*]

lemma *prefix-hfs-valid-prefix-generic*:
prefix (*hfs-valid-prefix-generic ainfo uinfo pas pre fut nxt*) *fut*
apply(*simp add*: *hfs-valid-prefix-generic-def*)
by (*metis prefixI takeWhile-dropWhile-id*)

lemma *cons-hfs-valid-prefix-generic*:
⟦*hf-valid-generic ainfo uinfo hfs* (*head pas*) *hf1* (*head fut*); *hfs* = (*rev pas*)@*hf1* #*fut*⟧
$\Longrightarrow$ *hfs-valid-prefix-generic ainfo uinfo pas* (*head pas*) (*hf1* # *fut*) *None* =
hf1 # (*hfs-valid-prefix-generic ainfo uinfo* (*hf1*#*pas*) (*Some hf1*) *fut None*)
by(*auto simp add*: *TW.takeW-split-tail*)

sublocale *dataplane-2-defs - - - auth-seg0 hf-valid-generic hfs-valid-prefix-generic checkInfo extr extr-ainfo ik-auth-ainfo ik-hf*
apply *unfold-locales*
using *prefix-hfs-valid-prefix-generic cons-hfs-valid-prefix-generic* **by** *blast*+

lemma *auth-seg2-elem*: ⟦(*ainfo*, *hfs*) $\in$ *auth-seg2*; *hf* $\in$ *set hfs*⟧
$\Longrightarrow$ $\exists$ *uinfo* . *hf-valid ainfo uinfo hfs hf* $\wedge$ *checkInfo ainfo* $\wedge$ (*ainfo*, *AHIS hfs*) $\in$ *auth-seg0*
by (*auto simp add*: *auth-seg2-def TW.holds-takeW-is-identity dest*!: *TW.holds-set-list*)

end

print-locale *dataplane-3-undirected-defs*
locale *dataplane-3-undirected-ik-defs* = *dataplane-3-undirected-defs - - - - hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf* **for**
hf-valid :: $'ainfo \Rightarrow$ *msgterm* $\Rightarrow$ ($'aahi$, $'uhi$) *HF list* $\Rightarrow$ ($'aahi$, $'uhi$) *HF* $\Rightarrow$ *bool*
and *checkInfo* :: $'ainfo \Rightarrow$ *bool*
and *extr* :: *msgterm* $\Rightarrow$ $'aahi$ *ahi-scheme list*
and *extr-ainfo* :: *msgterm* $\Rightarrow$ $'ainfo$

and *ik-auth-ainfo* :: *'ainfo* ⇒ *msgterm*
and *ik-hf* :: (*'aahi*, *'uhi*) *HF* ⇒ *msgterm set*
+
— *ik-add* is Additional Intruder Knowledge, such as hop authenticators in EPIC L1.
fixes *ik-add* :: *msgterm set*
— *ik-oracle* is another type of additional Intruder Knowledge. We use it to model the attacker's ability to brute-force individual hop validation fields and segment identifiers.
and *ik-oracle* :: *msgterm set*
— As *ik-oracle* gives the attacker direct access to hop validation fields that could be used to break the property, we have to either restrict the scope of the property, or restrict the attacker such that he cannot use the oracle-obtained hop validation fields in packets whose path origin matches the path origin of the oracle query. We choose the latter approach and fix a predicate *no-oracle* that tells us if the oracle has not been queried for a path origin (ainfo, uinfo combination). This is a prophecy variable.
and *no-oracle* :: *'ainfo* ⇒ *msgterm* ⇒ *bool*
begin

print-locale *dataplane-2-ik-defs*
sublocale *dataplane-2-ik-defs - - - - hfs-valid-prefix-generic checkInfo extr extr-ainfo ik-auth-ainfo ik-hf hf-valid-generic ik-add ik-oracle no-oracle*
by *unfold-locales*
end
print-locale *dataplane-3-undirected-ik-defs*
locale *dataplane-3-undirected* = *dataplane-3-undirected-ik-defs - - - - hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf ik-add ik-oracle no-oracle*
for *hf-valid* :: *'ainfo* ⇒ *msgterm* ⇒ (*'aahi*, *'uhi*) *HF list* ⇒ (*'aahi*, *'uhi*) *HF* ⇒ *bool*
and *checkInfo* :: *'ainfo* ⇒ *bool*
and *extr* :: *msgterm* ⇒ *'aahi ahi-scheme list*
and *extr-ainfo* :: *msgterm* ⇒ *'ainfo*
and *ik-auth-ainfo* :: *'ainfo* ⇒ *msgterm*
and *ik-hf* :: (*'aahi*, *'uhi*) *HF* ⇒ *msgterm set*
and *ik-add* :: *msgterm set*
and *ik-oracle* :: *msgterm set*
and *no-oracle* :: *'ainfo* ⇒ *msgterm* ⇒ *bool* +

— A valid validation field that is contained in ik corresponds to an authorized hop field. (The notable exceptions being oracle-obtained validation fields.) This relates the result of *ik-hf* to its argument. *ik-hf* has to produce a msgterm that is either unique for each given hop field x, or it is only produced by an 'equivalence class' of hop fields such that either all of the hop fields of the class are authorized, or none are. While the extr function (constrained by assumptions below) also binds the hop information to the validation field, it does so only for AHI and AInfo, but not for UHI.
assumes *COND-ik-hf*:
⟦*hf-valid ainfo uinfo l hf*; *ik-hf hf* ⊆ *analz ik*; *ik-auth-ainfo ainfo* ∈ *analz ik*;
no-oracle ainfo uinfo; *hf* ∈ *set l*⟧
⟹ ∃ *hfs* . *hf* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
— A valid validation field that can be synthesized from the initial intruder knowledge is already contained in the initial intruder knowledge if it belongs to an honest AS. This can be combined with the previous assumption.
and *COND-honest-hf-analz*:
⟦*ASID* (*AHI hf*) ∉ *bad*; *hf-valid ainfo uinfo l hf*; *ik-hf hf* ⊆ *synth* (*analz ik*);
no-oracle ainfo uinfo; *hf* ∈ *set l*⟧
⟹ *ik-hf hf* ⊆ *analz ik*
— A valid info field that can be synthesized from the initial intruder knowledge is already contained

in the initial intruder knowlege.

and *COND-ainfo-analz*:
$[\![$*hf-valid ainfo uinfo l hf*; *ik-auth-ainfo ainfo* $\in$ *synth* (*analz ik*)$]\!]$
$\Longrightarrow$ *ik-auth-ainfo ainfo* $\in$ *analz ik*

— Each valid hop field contains the entire path.

and *COND-extr*:
$[\![$*hf-valid ainfo uinfo l hf*$]\!]$ $\Longrightarrow$ *extr* (*HVF hf*) = *AHIS l*

— A valid hop field is only valid for one specific uinfo.

and *COND-hf-valid-uinfo*:
$[\![$*hf-valid ainfo uinfo l hf*; *hf-valid ainfo′ uinfo′ l′ hf*$]\!]$
$\Longrightarrow$ *uinfo′* = *uinfo*

begin

This is the central lemma that we need to prove to show the refinement between this model and dp1. It states: If an attacker can synthesize a segment from his knowledge, and does not use a path origin that was used to query the oracle, then the valid prefix of the segment is authorized. Thus, the attacker cannot create any valid but unauthorized segments.

lemma *ik-seg-is-auth*:
assumes $\bigwedge$*hf* . *hf* $\in$ *set fut* $\Longrightarrow$ *ik-hf hf* $\subseteq$ *synth* (*analz ik*) **and**
ik-auth-ainfo ainfo $\in$ *synth* (*analz ik*) **and** *nxt* = *None* **and** *no-oracle ainfo uinfo*
shows *pfragment ainfo*
(*AHIS* (*hfs-valid-prefix ainfo uinfo pas fut*))
auth-seg0
proof−
let *?hfsvalid* = *hfs-valid-prefix ainfo uinfo pas fut*
let *?AHIS* = *AHIS ?hfsvalid*

show *?thesis*
proof(*cases* $\exists$ *hfhonest* $\in$ *set ?AHIS* . *ASID hfhonest* $\notin$ *bad*)
case *True*
then obtain *hfhonesta* **where** *hfhonesta-def*: *hfhonesta* $\in$ *set ?AHIS ASID hfhonesta* $\notin$ *bad* **by** *auto*
then obtain *hfhonestc* **where** *hfhonestc-def*:
hfhonestc $\in$ *set ?hfsvalid hfhonesta* = *AHI hfhonestc ASID* (*AHI hfhonestc*) $\notin$ *bad*
by(*auto dest*: *AHIS-set*)
then have *hfhonestc-valid*: *hf-valid ainfo uinfo* (*rev*(*pas*)@*fut*) *hfhonestc* **using** *hfhonesta-def*
by (*meson set-takeWhileD*)
have *hfhonestc-fut*: *hfhonestc* $\in$ *set fut* **using** *hfhonestc-def*(*1*) **using** *set-takeWhileD* **by** *fastforce*
from *hfhonestc-valid* **have** *ik-hf hfhonestc* $\subseteq$ *analz ik*
using *hfhonestc-def*
apply−
apply(*erule COND-honest-hf-analz*[**where** *l*=(*rev*(*pas*)@*fut*)])
using *assms hfhonesta-def set-takeWhileD*
apply *auto*
apply *force*
by *force*
then obtain *hfshonest* **where** *hfshonest-def*: *hfhonestc* $\in$ *set hfshonest* (*ainfo*, *hfshonest*) $\in$ *auth-seg2*

using *hfhonestc-valid*
apply−
apply(*drule COND-ik-hf*) **using** *assms* **apply** *auto*
using *COND-ainfo-analz hfhonestc-valid hfhonestc-fut* **by** *auto*

then obtain *uinfo'* **where** *hfhonestc-valid'*:
hf-valid ainfo uinfo' hfshonest hfhonestc **by**(*auto simp add*: *auth-seg2-def*)
then have *uinfo'-uinfo*[*simp*]:*uinfo'* = *uinfo* **using** *hfhonestc-valid COND-hf-valid-uinfo* **by** *simp*
then have *AHIS-hfshonest*[*simp*]: *AHIS hfshonest* = *AHIS* (*rev*(*pas*)@*fut*)
using *hfhonestc-valid hfhonestc-valid'* **by**(*auto dest*!: *COND-extr*)
show *?thesis*
using *hfshonest-def* [*simplified*]
apply(*auto simp add*: *auth-seg2-def pfragment-def simp del*: *AHIS-def map-append*)
using *takeWhile-dropWhile-id map-append AHIS-def* **by** *metis*
next
case *False*
then show *?thesis*
by (*auto intro*!: *pfragment-self ASM-adversary*)
qed
qed

sublocale *dataplane-2* - - - - *hfs-valid-prefix-generic* - - - - - - - - *hf-valid-generic*
apply *unfold-locales*
by (*auto simp add*: *ik-seg-is-auth strip-ifs-valid-prefix simp del*: *AHIS-def*)

end
end

Chapter 3

Instances

Here we instantiate our concrete parametrized models with a number of protocols from the literature and variants of them that we derive ourselves.

3.1 SCION

theory *SCION*
  imports
    ../Parametrized-Dataplane-3-directed
    ../infrastructure/Keys
begin

locale *scion-defs* = *network-assums-direct - - - auth-seg0*
  for *auth-seg0* :: (*msgterm* × *ahi list*) *set*
begin

3.1.1 Hop validation check and extract functions

type-synonym *SCION-HF* = (*unit*, *unit*) *HF*

The predicate *hf-valid* is given to the concrete parametrized model as a parameter. It ensures the authenticity of the hop authenticator in the hop field. The predicate takes an authenticated info field (in this model always a numeric value, hence the matching on Num ts), the hop field to be validated and in some cases the next hop field.

We distinguish if there is a next hop field (this yields the two cases below). If there is not, then the hvf simply consists of a MAC over the authenticated info field and the local routing information of the hop, using the key of the hop to which the hop field belongs. If on the other hand, there is a subsequent hop field, then the hvf of that hop field is also included in the MAC computation.

fun *hf-valid* :: *msgterm* ⇒ *msgterm*
    ⇒ *SCION-HF option*
    ⇒ *SCION-HF*
    ⇒ *SCION-HF option* ⇒ *bool* **where**
  hf-valid (*Num ts*) *uinfo* - (|*AHI* = *ahi*, *UHI* = -, *HVF* = *x*|) (*Some* (|*AHI* = *ahi2*, *UHI* = -, *HVF* = *x2*|)) ⟷
    (∃ *upif downif upif2 downif2*.
        x = *Mac*[*macKey* (*ASID ahi*)] (*L* [*Num ts*, *upif*, *downif*, *upif2*, *downif2*, *x2*]) ∧
        ASIF (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif* ∧
        ASIF (*DownIF ahi2*) *downif2* ∧ *ASIF* (*UpIF ahi2*) *upif2* ∧ *uinfo* = ε)
| *hf-valid* (*Num ts*) *uinfo* - (|*AHI* = *ahi*, *UHI* = -, *HVF* = *x*|) *None* ⟷
    (∃ *upif downif*. *x* = *Mac*[*macKey* (*ASID ahi*)] (*L* [*Num ts*, *upif*, *downif*]) ∧
        ASIF (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif* ∧ *uinfo* = ε)
| *hf-valid* - - - - - = *False*

We can extract the entire path from the hvf field, which includes the local forwarding of the current hop, the local forwarding information of the next hop (if existant) and, recursively, all upstream hvf fields and their hop information.

fun *extr* :: *msgterm* ⇒ *ahi list* **where**
  extr (*Mac*[*macKey asid*] (*L* [*ts*, *upif*, *downif*, *upif2*, *downif2*, *x2*]))
  = (|*UpIF* = *ASO upif*, *DownIF* = *ASO downif*, *ASID* = *asid*|) # *extr x2*
| *extr* (*Mac*[*macKey asid*] (*L* [*ts*, *upif*, *downif*]))
  = [(|*UpIF* = *ASO upif*, *DownIF* = *ASO downif*, *ASID* = *asid*|)]
| *extr* - = []

Extract the authenticated info field from a hop validation field.

fun *extr-ainfo* :: *msgterm* ⇒ *msgterm* **where**
  extr-ainfo (*Mac*[*macKey asid*] (*L* (*Num ts* # *xs*))) = *Num ts*
| *extr-ainfo* - = ε

abbreviation *ik-auth-ainfo* :: *msgterm* ⇒ *msgterm* **where**
  ik-auth-ainfo ≡ *id*

An authenticated info field is always a number (corresponding to a timestamp).

abbreviation *checkInfo* **where**
  checkInfo ainfo ≡ (∃ *ts*. *ainfo* = *Num ts*)

When observing a hop field, an attacker learns the HVF. UHI is empty and the AHI only contains public information that are not terms.

fun *ik-hf* :: *SCION-HF* ⇒ *msgterm set* **where**
  ik-hf hf = {*HVF hf*}

abbreviation *no-oracle* **where** *no-oracle* ≡ (λ - -. *True*)

We now define useful properties of the above definition.

lemma *hf-valid-invert*:
  hf-valid tsn uinfo prev hf mo ⟷
   ((∃ *ahi ahi2 ts upif downif asid x upif2 downif2 x2*.
     hf = (|*AHI* = *ahi*, *UHI* = (), *HVF* = *x*|) ∧
     ASID ahi = *asid* ∧ *ASIF* (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif* ∧
     mo = *Some* (|*AHI* = *ahi2*, *UHI* = (), *HVF* = *x2*|) ∧
     ASIF (*DownIF ahi2*) *downif2* ∧ *ASIF* (*UpIF ahi2*) *upif2* ∧
     x = *Mac*[*macKey asid*] (*L* [*tsn*, *upif*, *downif*, *upif2*, *downif2*, *x2*]) ∧
     tsn = *Num ts* ∧
     uinfo = ε)
  ∨ (∃ *ahi ts upif downif asid x*.
     hf = (|*AHI* = *ahi*, *UHI* = (), *HVF* = *x*|) ∧
     ASID ahi = *asid* ∧ *ASIF* (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif* ∧
     mo = *None* ∧
     x = *Mac*[*macKey asid*] (*L* [*tsn*, *upif*, *downif*]) ∧
     tsn = *Num ts* ∧
     uinfo = ε)
)
  by(*auto elim*!: *hf-valid.elims*)

lemma *hf-valid-checkInfo*[*dest*]: *hf-valid ainfo uinfo prev hf z* ⟹ *checkInfo ainfo*
  by(*auto simp add*: *hf-valid-invert*)

lemma *info-hvf*:
  assumes *hf-valid ainfo uinfo prev m z hf-valid ainfo′ uinfo′ prev′ m′ z′ HVF m* = *HVF m′*
  shows *ainfo′* = *ainfo m′* = *m*
  using *assms* **by**(*auto simp add*: *hf-valid-invert intro*: *ahi-eq*)

3.1.2 Definitions and properties of the added intruder knowledge

Here we define a *ik-add* and *ik-oracle* as being empty, as these features are not used in this instance model.

sublocale *dataplane-3-directed-defs - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf*
by *unfold-locales*

declare *TW.holds-set-list*[*dest*]
declare *TW.holds-takeW-is-identity*[*simp*]
declare *parts-singleton*[*dest*]

abbreviation *ik-add* :: *msgterm set* **where** *ik-add* ≡ {}

abbreviation *ik-oracle* :: *msgterm set* **where** *ik-oracle* ≡ {}

3.1.3 Properties of the intruder knowledge, including *ik-add* and *ik-oracle*

We now instantiate the parametrized model's definition of the intruder knowledge, using the definitions of *ik-add* and *ik-oracle* from above. We then prove the properties that we need to instantiate the *dataplane-3-directed* locale.

sublocale
dataplane-3-directed-ik-defs - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo
ik-hf ik-add ik-oracle no-oracle
by *unfold-locales*

lemma *ik-auth-hfs-form*: $t \in parts\ ik\text{-}auth\text{-}hfs \Longrightarrow \exists\ t'.\ t = Hash\ t'$
apply *auto* **apply**(*drule parts-singleton*)
by(*auto simp add*: *auth-seg2-def hf-valid-invert dest*!: *TW.holds-set-list*)

declare *ik-auth-hfs-def* [*simp del*]

lemma *parts-ik-auth-hfs*[*simp*]: $parts\ ik\text{-}auth\text{-}hfs = ik\text{-}auth\text{-}hfs$
by (*auto intro*!: *parts-Hash ik-auth-hfs-form*)

This lemma allows us not only to expand the definition of *ik-auth-hfs*, but also to obtain useful properties, such as a term being a Hash, and it being part of a valid hop field.

lemma *ik-auth-hfs-simp*:
$t \in ik\text{-}auth\text{-}hfs \longleftrightarrow (\exists\, t'.\ t = Hash\ t') \land (\exists\, hf.\ t = HVF\ hf$
$\land (\exists\, hfs.\ hf \in set\ hfs \land (\exists\, ainfo.\ (ainfo, hfs) \in auth\text{-}seg2$
$\land (\exists\ prev\ nxt\ uinfo.\ hf\text{-}valid\ ainfo\ uinfo\ prev\ hf\ nxt))))$ (**is** *?lhs* ⟷ *?rhs*)
proof
assume *asm*: *?lhs*
then obtain *ainfo hf hfs* **where**
dfs: $hf \in set\ hfs\ (ainfo, hfs) \in auth\text{-}seg2\ t = HVF\ hf$
by(*auto simp add*: *ik-auth-hfs-def*)
then obtain *uinfo* **where** *hfs-valid-None ainfo uinfo hfs* $(ainfo, AHIS\ hfs) \in auth\text{-}seg0$
by(*auto simp add*: *auth-seg2-def*)
then show *?rhs* **using** *asm dfs* **by**(*fast intro*: *ik-auth-hfs-form*)
qed(*auto simp add*: *ik-auth-hfs-def*)

Properties of Intruder Knowledge

lemma *auth-ainfo*[*dest*]: $[\![(ainfo, hfs) \in auth\text{-}seg2]\!] \Longrightarrow \exists\ ts.\ ainfo = Num\ ts$
by(*auto simp add*: *auth-seg2-def*)

lemma *Num-ik*[*intro*]: $Num\ ts \in ik$

by(*auto simp add*: *ik-def*)
(*auto simp add*: *auth-seg2-def TW.holds.simps*(*3*) *elim*!: *allE*[*of* - []])

There are no ciphertexts (or signatures) in *parts ik*. Thus, *analz ik* and *parts ik* are identical.

lemma *analz-parts-ik*[*simp*]: *analz ik* = *parts ik*
by(*rule no-crypt-analz-is-parts*)
(*auto simp add*: *ik-def auth-seg2-def ik-auth-hfs-simp*)

lemma *parts-ik*[*simp*]: *parts ik* = *ik*
by(*auto 3 4 simp add*: *ik-def auth-seg2-def*)

lemma *key-ik-bad*: *Key* (*macK asid*) ∈ *ik* ⟹ *asid* ∈ *bad*
by(*auto simp add*: *ik-def hf-valid-invert*)
(*auto 3 4 simp add*: *auth-seg2-def ik-auth-hfs-simp hf-valid-invert*)

lemma *MAC-synth-helper*:
assumes *hf-valid ainfo uinfo prev m z HVF m* = *Mac*[*Key* (*macK asid*)] *j HVF m* ∈ *ik*
shows ∃ *hfs*. *m* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
proof−
from *assms*(*2*−*3*) **obtain** *ainfo′ uinfo′ m′ hfs′ prev′ nxt′* **where** *dfs*:
m′ ∈ *set hfs′* (*ainfo′*, *hfs′*) ∈ *auth-seg2 hf-valid ainfo′ uinfo′ prev′ m′ nxt′ HVF m* = *HVF m′*
by(*auto simp add*: *ik-def ik-auth-hfs-simp*)
then have *ainfo′* = *ainfo m′* = *m* **using** *assms*(*1*) **by**(*auto elim*!: *info-hvf*)
then show *?thesis* **using** *dfs assms* **by** *auto*
qed

This definition helps with the limiting the number of cases generated. We don't require it, but it is convenient. Given a hop validation field and an asid, return if the hvf has the expected format.

definition *mac-format* :: *msgterm* ⇒ *as* ⇒ *bool* **where**
mac-format m asid ≡ ∃ *j* . *m* = *Mac*[*macKey asid*] *j*

If a valid hop field is derivable by the attacker, but does not belong to the attacker, then the hop field is already contained in the set of authorized segments.

lemma *MAC-synth*:
assumes *hf-valid ainfo uinfo prev m z HVF m* ∈ *synth ik mac-format* (*HVF m*) *asid*
asid ∉ *bad checkInfo ainfo*
shows ∃ *hfs* . *m* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
using *assms*
apply(*auto simp add*: *mac-format-def elim*!: *MAC-synth-helper dest*!: *key-ik-bad*)
by(*auto simp add*: *ik-def ik-auth-hfs-simp*)

3.1.4 Direct proof goals for interpretation of *dataplane-3-directed*

lemma *COND-honest-hf-analz*:
assumes *ASID* (*AHI hf*) ∉ *bad hf-valid ainfo uinfo prev hf nxt ik-hf hf* ⊆ *synth* (*analz ik*)
no-oracle ainfo uinfo
shows *ik-hf hf* ⊆ *analz ik*
proof−
let *?asid* = *ASID* (*AHI hf*)
from *assms*(*3*) **have** *hf-synth-ik*: *HVF hf* ∈ *synth ik* **by** *auto*
from *assms*(*2*) **have** *mac-format* (*HVF hf*) *?asid checkInfo ainfo*

by(*auto simp add*: *mac-format-def hf-valid-invert*)
then obtain *hfs* **where** *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2*
using *assms*(*1*,*2*,*4*) *hf-synth-ik* **by**(*auto dest*!: *MAC-synth*)
then have *HVF hf* ∈ *ik*
using *assms*(*2*)
by(*auto simp add*: *ik-auth-hfs-def intro*!: *ik-ik-auth-hfs intro*!: *exI*)
then show *?thesis* **by** *auto*
qed

lemma *COND-ainfo-analz*:
assumes *hf-valid ainfo uinfo prev hf nxt* **and** *ik-auth-ainfo ainfo* ∈ *synth* (*analz ik*)
shows *ik-auth-ainfo ainfo* ∈ *analz ik*
using *assms* **by**(*auto simp add*: *hf-valid-invert*)

lemma *COND-ik-hf*:
assumes *hf-valid ainfo uinfo prev hf z* **and** *HVF hf* ∈ *ik* **and** *no-oracle ainfo uinfo*
shows ∃ *hfs*. *hf* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
proof−
from *assms* **have** *checkInfo ainfo* **by** *auto*
then obtain *hfs ainfo* **where** *hfs-def*: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2*
using *assms* **by**(*auto 3 4 simp add*: *hf-valid-invert ik-auth-hfs-simp ik-def dest*: *ahi-eq*)
then obtain *hfs ainfo* **where** *hfs-def*: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2* **by** *auto*
show *?thesis*
using *hfs-def* **apply** (*auto simp add*: *auth-seg2-def dest*!: *TW*.*holds-set-list*)
using *hfs-def assms*(*1*) **by** (*auto simp add*: *auth-seg2-def dest*: *info-hvf*)
qed

lemma *COND-extr-prefix-path*:
⟦*hfs-valid ainfo uinfo pre l nxt*; *nxt* = *None*⟧ ⟹ *prefix* (*extr-from-hd l*) (*AHIS l*)
by(*induction pre l nxt rule*: *TW*.*holds*.*induct*)
(*auto simp add*: *TW*.*holds-split-tail TW*.*holds*.*simps*(*1*) *hf-valid-invert*,
auto split: *list*.*split-asm simp add*: *hf-valid-invert intro*!: *ahi-eq elim*: *ASIF*.*elims*)

lemma *COND-path-prefix-extr*:
prefix (*AHIS* (*hfs-valid-prefix ainfo uinfo pre l nxt*))
(*extr-from-hd l*)
apply(*induction pre l nxt rule*: *TW*.*takeW*.*induct*[**where** *?Pa*=*hf-valid ainfo uinfo*])
apply(*auto simp add*: *TW*.*takeW-split-tail TW*.*takeW*.*simps*(*1*))
apply(*auto simp add*: *hf-valid-invert intro*!: *ahi-eq*)
by(*auto elim*: *ASIF*.*elims*)

lemma *COND-hf-valid-no-prev*:
hf-valid ainfo uinfo prev hf z ⟷ *hf-valid ainfo uinfo prev′ hf z*
by(*auto simp add*: *hf-valid-invert*)

lemma *COND-hf-valid-uinfo*:
⟦*hf-valid ainfo uinfo pre hf nxt*; *hf-valid ainfo′ uinfo′ pre′ hf nxt′*⟧ ⟹ *uinfo′* = *uinfo*
by(*auto simp add*: *hf-valid-invert*)

3.1.5 Instantiation of *dataplane-3-directed* locale

sublocale
dataplane-3-directed - - - *auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf ik-add*

ik-oracle no-oracle
apply *unfold-locales*
using *COND-ik-hf COND-honest-hf-analz COND-ainfo-analz COND-extr-prefix-path COND-path-prefix-extr COND-hf-valid-no-prev COND-hf-valid-uinfo* **by** *auto*

end
end

3.2 SCION

This is a slightly variant version of SCION, in which the successor's hop information is not embedded in the MAC of a hop field. This difference shows up in the definition of *hf-valid*.

theory *SCION-variant*
  imports
    ../Parametrized-Dataplane-3-directed
    ../infrastructure/Keys
begin

locale *scion-defs* = *network-assums-direct* - - - *auth-seg0*
  for *auth-seg0* :: (*msgterm* × *ahi list*) *set*
begin

3.2.1 Hop validation check and extract functions

type-synonym *SCION-HF* = (*unit*, *unit*) *HF*

The predicate *hf-valid* is given to the concrete parametrized model as a parameter. It ensures the authenticity of the hop authenticator in the hop field. The predicate takes an authenticated info field (in this model always a numeric value, hence the matching on Num ts), the hop field to be validated and in some cases the next hop field.

We distinguish if there is a next hop field (this yields the two cases below). If there is not, then the hvf simply consists of a MAC over the authenticated info field and the local routing information of the hop, using the key of the hop to which the hop field belongs. If on the other hand, there is a subsequent hop field, then the hvf of that hop field is also included in the MAC computation.

fun *hf-valid* :: *msgterm* ⇒ *msgterm*
  ⇒ *SCION-HF option*
  ⇒ *SCION-HF*
  ⇒ *SCION-HF option* ⇒ *bool* **where**
 hf-valid (*Num ts*) *uinfo* - (|*AHI* = *ahi*, *UHI* = -, *HVF* = *x*|) (*Some* (|*AHI* = *ahi2*, *UHI* = -, *HVF* = *x2*|)) ⟷
  (∃ *upif downif*. *x* = *Mac*[*macKey* (*ASID ahi*)] (*L* [*Num ts*, *upif*, *downif*, *x2*]) ∧
    ASIF (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif* ∧ *uinfo* = ε)
| *hf-valid* (*Num ts*) *uinfo* - (|*AHI* = *ahi*, *UHI* = -, *HVF* = *x*|) *None* ⟷
  (∃ *upif downif*. *x* = *Mac*[*macKey* (*ASID ahi*)] (*L* [*Num ts*, *upif*, *downif*]) ∧
    ASIF (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif* ∧ *uinfo* = ε)
| *hf-valid* - - - - - = *False*

We can extract the entire path from the hvf field, which includes the local forwarding information as well as, recursively, all upstream hvf fields and their hop information.

fun *extr* :: *msgterm* ⇒ *ahi list* **where**
 extr (*Mac*[*macKey asid*] (*L* [*ts*, *upif*, *downif*, *x2*]))
 = (|*UpIF* = *ASO upif*, *DownIF* = *ASO downif*, *ASID* = *asid*|) # *extr x2*
| *extr* (*Mac*[*macKey asid*] (*L* [*ts*, *upif*, *downif*]))
 = [(|*UpIF* = *ASO upif*, *DownIF* = *ASO downif*, *ASID* = *asid*|)]
| *extr* - = []

Extract the authenticated info field from a hop validation field.

fun *extr-ainfo* :: *msgterm* ⇒ *msgterm* **where**
  extr-ainfo (*Mac*[*macKey asid*] (*L* (*Num ts* # *xs*))) = *Num ts*
| *extr-ainfo* - = *ε*

abbreviation *ik-auth-ainfo* :: *msgterm* ⇒ *msgterm* **where**
  ik-auth-ainfo ≡ *id*

An authenticated info field is always a number (corresponding to a timestamp).

abbreviation *checkInfo* **where**
  checkInfo ainfo ≡ (∃ *ts*. *ainfo* = *Num ts*)

When observing a hop field, an attacker learns the HVF. UHI is empty and the AHI only contains public information that are not terms.

fun *ik-hf* :: *SCION-HF* ⇒ *msgterm set* **where**
  ik-hf hf = {*HVF hf*}

abbreviation *no-oracle* **where** *no-oracle* ≡ (λ - -. *True*)

We now define useful properties of the above definition.

lemma *hf-valid-invert*:
  hf-valid tsn uinfo prev hf mo ⟷
   ((∃ *ahi ahi2 ts upif downif asid x x2*.
     hf = (|*AHI* = *ahi*, *UHI* = (), *HVF* = *x*|) ∧
     ASID ahi = *asid* ∧ *ASIF* (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif* ∧
     mo = *Some* (|*AHI* = *ahi2*, *UHI* = (), *HVF* = *x2*|) ∧
     x = *Mac*[*macKey asid*] (*L* [*tsn*, *upif*, *downif*, *x2*]) ∧
     tsn = *Num ts* ∧
     uinfo = *ε*)
  ∨ (∃ *ahi ts upif downif asid x*.
     hf = (|*AHI* = *ahi*, *UHI* = (), *HVF* = *x*|) ∧
     ASID ahi = *asid* ∧ *ASIF* (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif* ∧
     mo = *None* ∧
     x = *Mac*[*macKey asid*] (*L* [*tsn*, *upif*, *downif*]) ∧
     tsn = *Num ts* ∧
     uinfo = *ε*)
)
  by(*auto elim*!: *hf-valid.elims*)

lemma *hf-valid-checkInfo*[*dest*]: *hf-valid ainfo uinfo prev hf z* ⟹ *checkInfo ainfo*
  by(*auto simp add*: *hf-valid-invert*)

lemma *info-hvf*:
  assumes *hf-valid ainfo uinfo prev m z hf-valid ainfo′ uinfo′ prev′ m′ z′ HVF m* = *HVF m′*
  shows *ainfo′* = *ainfo m′* = *m*
  using *assms* **by**(*auto simp add*: *hf-valid-invert intro*: *ahi-eq*)

3.2.2 Definitions and properties of the added intruder knowledge

Here we define a *ik-add* and *ik-oracle* as being empty, as these features are not used in this instance model.

sublocale *dataplane-3-directed-defs* - - - *auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf*

by *unfold-locales*

declare *TW.holds-set-list*[*dest*]
declare *TW.holds-takeW-is-identity*[*simp*]
declare *parts-singleton*[*dest*]

abbreviation *ik-add* :: *msgterm set* **where** *ik-add* ≡ {}

abbreviation *ik-oracle* :: *msgterm set* **where** *ik-oracle* ≡ {}

3.2.3 Properties of the intruder knowledge, including *ik-add* and *ik-oracle*

We now instantiate the parametrized model's definition of the intruder knowledge, using the definitions of *ik-add* and *ik-oracle* from above. We then prove the properties that we need to instantiate the *dataplane-3-directed* locale.

sublocale
dataplane-3-directed-ik-defs - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf ik-add ik-oracle no-oracle
by *unfold-locales*

lemma *ik-auth-hfs-form*: $t \in parts\ ik\text{-}auth\text{-}hfs \Longrightarrow \exists\ t'.\ t = Hash\ t'$
apply *auto* **apply**(*drule parts-singleton*)
by(*auto simp add*: *auth-seg2-def hf-valid-invert dest*!: *TW.holds-set-list*)

declare *ik-auth-hfs-def*[*simp del*]

lemma *parts-ik-auth-hfs*[*simp*]: *parts ik-auth-hfs* = *ik-auth-hfs*
by (*auto intro*!: *parts-Hash ik-auth-hfs-form*)

This lemma allows us not only to expand the definition of *ik-auth-hfs*, but also to obtain useful properties, such as a term being a Hash, and it being part of a valid hop field.

lemma *ik-auth-hfs-simp*:
$t \in ik\text{-}auth\text{-}hfs \longleftrightarrow (\exists t'.\ t = Hash\ t') \wedge (\exists hf.\ t = HVF\ hf$
$\wedge (\exists hfs.\ hf \in set\ hfs \wedge (\exists ainfo.\ (ainfo, hfs) \in auth\text{-}seg2$
$\wedge (\exists\ prev\ nxt\ uinfo.\ hf\text{-}valid\ ainfo\ uinfo\ prev\ hf\ nxt))))$ (**is** *?lhs* ⟷ *?rhs*)
proof
assume *asm*: *?lhs*
then obtain *ainfo hf hfs* **where**
dfs: $hf \in set\ hfs$ $(ainfo, hfs) \in auth\text{-}seg2$ $t = HVF\ hf$
by(*auto simp add*: *ik-auth-hfs-def*)
then obtain *uinfo* **where** *hfs-valid-None ainfo uinfo hfs* $(ainfo, AHIS\ hfs) \in auth\text{-}seg0$
by(*auto simp add*: *auth-seg2-def*)
then show *?rhs* **using** *asm dfs* **by**(*fast intro*: *ik-auth-hfs-form*)
qed(*auto simp add*: *ik-auth-hfs-def*)

Properties of Intruder Knowledge

lemma *auth-ainfo*[*dest*]: $[\![(ainfo, hfs) \in auth\text{-}seg2]\!] \Longrightarrow \exists\ ts.\ ainfo = Num\ ts$
by(*auto simp add*: *auth-seg2-def*)

lemma *Num-ik*[*intro*]: $Num\ ts \in ik$
by(*auto simp add*: *ik-def*)

(*auto simp add*: *auth-seg2-def TW.holds.simps*(*3*) *elim*!: *allE*[*of* - []])

There are no ciphertexts (or signatures) in *parts ik*. Thus, *analz ik* and *parts ik* are identical.

lemma *analz-parts-ik*[*simp*]: *analz ik* = *parts ik*
by(*rule no-crypt-analz-is-parts*)
(*auto simp add*: *ik-def auth-seg2-def ik-auth-hfs-simp*)

lemma *parts-ik*[*simp*]: *parts ik* = *ik*
by(*auto 3 4 simp add*: *ik-def auth-seg2-def*)

lemma *key-ik-bad*: *Key* (*macK asid*) $\in$ *ik* $\Longrightarrow$ *asid* $\in$ *bad*
by(*auto simp add*: *ik-def hf-valid-invert*)
(*auto 3 4 simp add*: *auth-seg2-def ik-auth-hfs-simp hf-valid-invert*)

lemma *MAC-synth-helper*:
assumes *hf-valid ainfo uinfo prev m z HVF m* = *Mac*[*Key* (*macK asid*)] *j HVF m* $\in$ *ik*
shows $\exists$ *hfs*. *m* $\in$ *set hfs* $\wedge$ (*ainfo*, *hfs*) $\in$ *auth-seg2*
proof−
from *assms*(*2*−*3*) **obtain** *ainfo′ uinfo′ m′ hfs′ prev′ nxt′* **where** *dfs*:
m′ $\in$ *set hfs′* (*ainfo′*, *hfs′*) $\in$ *auth-seg2 hf-valid ainfo′ uinfo′ prev′ m′ nxt′ HVF m* = *HVF m′*
by(*auto simp add*: *ik-def ik-auth-hfs-simp*)
then have *ainfo′* = *ainfo m′* = *m* **using** *assms*(*1*) **by**(*auto elim*!: *info-hvf*)
then show *?thesis* **using** *dfs assms* **by** *auto*
qed

This definition helps with the limiting the number of cases generated. We don't require it, but it is convenient. Given a hop validation field and an asid, return if the hvf has the expected format.

definition *mac-format* :: *msgterm* $\Rightarrow$ *as* $\Rightarrow$ *bool* **where**
mac-format m asid $\equiv$ $\exists$ *j* . *m* = *Mac*[*macKey asid*] *j*

If a valid hop field is derivable by the attacker, but does not belong to the attacker, then the hop field is already contained in the set of authorized segments.

lemma *MAC-synth*:
assumes *hf-valid ainfo uinfo prev m z HVF m* $\in$ *synth ik mac-format* (*HVF m*) *asid*
asid $\notin$ *bad checkInfo ainfo*
shows $\exists$ *hfs* . *m* $\in$ *set hfs* $\wedge$ (*ainfo*, *hfs*) $\in$ *auth-seg2*
using *assms*
apply(*auto simp add*: *mac-format-def elim*!: *MAC-synth-helper dest*!: *key-ik-bad*)
by(*auto simp add*: *ik-def ik-auth-hfs-simp*)

3.2.4 Direct proof goals for interpretation of *dataplane-3-directed*

lemma *COND-honest-hf-analz*:
assumes *ASID* (*AHI hf*) $\notin$ *bad hf-valid ainfo uinfo prev hf nxt ik-hf hf* $\subseteq$ *synth* (*analz ik*)
no-oracle ainfo uinfo
shows *ik-hf hf* $\subseteq$ *analz ik*
proof−
let *?asid* = *ASID* (*AHI hf*)
from *assms*(*3*) **have** *hf-synth-ik*: *HVF hf* $\in$ *synth ik* **by** *auto*
from *assms*(*2*) **have** *mac-format* (*HVF hf*) *?asid checkInfo ainfo*
by(*auto simp add*: *mac-format-def hf-valid-invert*)

then obtain *hfs* **where** *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2*
using *assms*(*1*,*2*,*4*) *hf-synth-ik* **by**(*auto dest*!: *MAC-synth*)
then have *HVF hf* ∈ *ik*
using *assms*(*2*)
by(*auto simp add*: *ik-auth-hfs-def intro*!: *ik-ik-auth-hfs intro*!: *exI*)
then show *?thesis* **by** *auto*
qed

lemma *COND-ainfo-analz*:
assumes *hf-valid ainfo uinfo prev hf nxt* **and** *ik-auth-ainfo ainfo* ∈ *synth* (*analz ik*)
shows *ik-auth-ainfo ainfo* ∈ *analz ik*
using *assms* **by**(*auto simp add*: *hf-valid-invert*)

lemma *COND-ik-hf*:
assumes *hf-valid ainfo uinfo prev hf z* **and** *HVF hf* ∈ *ik* **and** *no-oracle ainfo uinfo*
shows ∃ *hfs*. *hf* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
proof−
from *assms* **have** *checkInfo ainfo* **by** *auto*
then obtain *hfs ainfo* **where** *hfs-def*: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2*
using *assms* **by**(*auto 3 4 simp add*: *hf-valid-invert ik-auth-hfs-simp ik-def dest*: *ahi-eq*)
then obtain *hfs ainfo* **where** *hfs-def*: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2* **by** *auto*
show *?thesis*
using *hfs-def* **apply** (*auto simp add*: *auth-seg2-def dest*!: *TW.holds-set-list*)
using *hfs-def assms*(*1*) **by** (*auto simp add*: *auth-seg2-def dest*: *info-hvf*)
qed

lemma *COND-extr-prefix-path*:
⟦*hfs-valid ainfo uinfo pre l nxt*; *nxt* = *None*⟧ ⟹ *prefix* (*extr-from-hd l*) (*AHIS l*)
by(*induction pre l nxt rule*: *TW.holds.induct*)
(*auto simp add*: *TW.holds-split-tail TW.holds.simps*(*1*) *hf-valid-invert*,
auto split: *list.split-asm simp add*: *hf-valid-invert intro*!: *ahi-eq elim*: *ASIF.elims*)

lemma *COND-path-prefix-extr*:
prefix (*AHIS* (*hfs-valid-prefix ainfo uinfo pre l nxt*))
(*extr-from-hd l*)
apply(*induction pre l nxt rule*: *TW.takeW.induct*[**where** *?Pa*=*hf-valid ainfo uinfo*])
apply(*auto simp add*: *TW.takeW-split-tail TW.takeW.simps*(*1*))
apply(*auto simp add*: *hf-valid-invert intro*!: *ahi-eq*)
by(*auto elim*: *ASIF.elims*)

lemma *COND-hf-valid-no-prev*:
hf-valid ainfo uinfo prev hf z ⟷ *hf-valid ainfo uinfo prev' hf z*
by(*auto simp add*: *hf-valid-invert*)

lemma *COND-hf-valid-uinfo*:
⟦*hf-valid ainfo uinfo pre hf nxt*; *hf-valid ainfo' uinfo' pre' hf nxt'*⟧ ⟹ *uinfo'* = *uinfo*
by(*auto simp add*: *hf-valid-invert*)

3.2.5 Instantiation of *dataplane-3-directed* locale

sublocale
dataplane-3-directed - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf ik-add ik-oracle no-oracle

apply *unfold-locales*
using *COND-ik-hf COND-honest-hf-analz COND-ainfo-analz COND-extr-prefix-path COND-path-prefix-extr COND-hf-valid-no-prev COND-hf-valid-uinfo* **by** *auto*

end
end

3.3 EPIC Level 1 in the Basic Attacker Model

theory *EPIC-L1-BA*
  imports
    ../Parametrized-Dataplane-3-directed
    ../infrastructure/Keys
begin

locale *epic-l1-defs* = *network-assums-direct* - - - *auth-seg0*
  for *auth-seg0* :: (*msgterm* × *ahi list*) *set*
begin

3.3.1 Hop validation check and extract functions

The predicate *hf-valid* is given to the concrete parametrized model as a parameter. It ensures the authenticity of the hop authenticator in the hop field. The predicate takes an authenticated info field (in this model always a numeric value, hence the matching on Num ts), an unauthenticated info field uinfo, the hop field to be validated and in some cases the next hop field.

We distinguish if there is a next hop field (this yields the two cases below). If there is not, then the hop authenticator σ simply consists of a MAC over the authenticated info field and the local routing information of the hop, using the key of the hop to which the hop field belongs. If on the other hand, there is a subsequent hop field, then the uhi field of that hop field is also included in the MAC computation.

The hop authenticator σ is used to compute both the hop validation field and the uhi field. The first is computed as a MAC over the path origin (pair of absolute timestamp ts and the relative timestamp given in uinfo), using the hop authenticator as a key to the MAC. The hop authenticator is not secret, and any end host can use it to create a valid hvf. The uhi field, according to the protocol description, is σ shortened to a few bytes. We model this as applying the hash on σ.

The predicate *hf-valid* checks if the hop authenticator, hvf and uhi field are computed correctly.

fun *hf-valid* :: *msgterm* ⇒ *msgterm*
    ⇒ (*unit*, *msgterm*) *HF option*
    ⇒ (*unit*, *msgterm*) *HF*
    ⇒ (*unit*, *msgterm*) *HF option* ⇒ *bool* **where**
  hf-valid (*Num ts*) *uinfo* - ⦇*AHI* = *ahi*, *UHI* = *uhi*, *HVF* = *x*⦈ (*Some* ⦇*AHI* = *ahi2*, *UHI* = *uhi2*, *HVF* = *x2*⦈) ⟷
    (∃ σ *upif downif*. σ = *Mac*[*macKey* (*ASID ahi*)] (*L* [*Num ts*, *upif*, *downif*, *uhi2*]) ∧
        ASIF (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif* ∧ *uhi* = *Hash* σ ∧ *x* = *Mac*[σ] ⟨*Num ts*, *uinfo*⟩)
| *hf-valid* (*Num ts*) *uinfo* - ⦇*AHI* = *ahi*, *UHI* = *uhi*, *HVF* = *x*⦈ *None* ⟷
    (∃ σ *upif downif*. σ = *Mac*[*macKey* (*ASID ahi*)] (*L* [*Num ts*, *upif*, *downif*]) ∧
        ASIF (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif* ∧ *uhi* = *Hash* σ ∧ *x* = *Mac*[σ] ⟨*Num ts*, *uinfo*⟩)
| *hf-valid* - - - - - = *False*

We can extract the entire path from the uhi field, since it includes the hop authenticator, which includes the local forwarding information as well as, recursively, all upstream hop authenticators and their hop information. However, the parametrized model defines the extract function to operate on the hop validation field, not the uhi field. We therefore define a separate function that extracts the path from a hvf. We can do so, as both hvf and uhi contain the hop authenticator. Internally, that function uses *extrUhi*.

fun *extrUhi* :: *msgterm* ⇒ *ahi list* **where**
extrUhi (*Hash* (*Mac*[*macKey asid*] (*L* [*ts*, *upif*, *downif*, *uhi2*])))
= (| *UpIF* = *ASO upif*, *DownIF* = *ASO downif*, *ASID* = *asid*|) # *extrUhi uhi2*
| *extrUhi* (*Hash* (*Mac*[*macKey asid*] (*L* [*ts*, *upif*, *downif*])))
= [(| *UpIF* = *ASO upif*, *DownIF* = *ASO downif*, *ASID* = *asid*|)]
| *extrUhi* - = []

This function extracts from a hop validation field (HVF hf) the entire path.

fun *extr* :: *msgterm* ⇒ *ahi list* **where**
extr (*Mac*[σ] -) = *extrUhi* (*Hash* σ)
| *extr* - = []

Extract the authenticated info field from a hop validation field.

fun *extr-ainfo* :: *msgterm* ⇒ *msgterm* **where**
extr-ainfo (*Mac*[*Mac*[*macKey asid*] (*L* (*Num ts* # *xs*))] -) = *Num ts*
| *extr-ainfo* - = ε

abbreviation *ik-auth-ainfo* :: *msgterm* ⇒ *msgterm* **where**
ik-auth-ainfo ≡ *id*

An authenticated info field is always a number (corresponding to a timestamp).

abbreviation *checkInfo* **where**
checkInfo ainfo ≡ (∃ *ts*. *ainfo* = *Num ts*)

When observing a hop field, an attacker learns the HVF and the UHI. The AHI only contains public information that are not terms.

fun *ik-hf* :: (*unit*, *msgterm*) *HF* ⇒ *msgterm set* **where**
ik-hf hf = {*HVF hf*, *UHI hf*}

abbreviation *no-oracle* **where** *no-oracle* ≡ (λ - -. *True*)

We now define useful properties of the above definition.

lemma *hf-valid-invert*:
hf-valid tsn uinfo prev hf mo ⟷
((∃ *ahi ahi2* σ *ts upif downif asid x upif2 downif2 asid2 uhi uhi2 x2*.
hf = (|*AHI* = *ahi*, *UHI* = *uhi*, *HVF* = *x*|) ∧
ASID ahi = *asid* ∧ *ASIF* (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif* ∧
mo = *Some* (|*AHI* = *ahi2*, *UHI* = *uhi2*, *HVF* = *x2*|) ∧
ASID ahi2 = *asid2* ∧ *ASIF* (*DownIF ahi2*) *downif2* ∧ *ASIF* (*UpIF ahi2*) *upif2* ∧
σ = *Mac*[*macKey asid*] (*L* [*tsn*, *upif*, *downif*, *uhi2*]) ∧
tsn = *Num ts* ∧
uhi = *Hash* σ ∧
x = *Mac*[σ] ⟨*tsn*, *uinfo*⟩)
∨ (∃ *ahi* σ *ts upif downif asid uhi x*.
hf = (|*AHI* = *ahi*, *UHI* = *uhi*, *HVF* = *x*|) ∧
ASID ahi = *asid* ∧ *ASIF* (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif* ∧
mo = *None* ∧
σ = *Mac*[*macKey asid*] (*L* [*tsn*, *upif*, *downif*]) ∧

$tsn = Num\ ts\ \wedge$
$uhi = Hash\ \sigma\ \wedge$
$x = Mac[\sigma]\ \langle tsn,\ uinfo\rangle)$
)
apply(*auto elim*!: *hf-valid.elims*) **using** *option.exhaust ASIF.simps* **by** *metis+*

lemma *hf-valid-checkInfo*[*dest*]: *hf-valid ainfo uinfo prev hf z* $\Longrightarrow$ *checkInfo ainfo*
by(*auto simp add*: *hf-valid-invert*)

lemma *info-hvf*:
assumes *hf-valid ainfo uinfo prev m z HVF m* $= Mac[\sigma]\ \langle ainfo',\ uinfo'\rangle \vee$ *hf-valid ainfo' uinfo' prev' m z'*
shows $uinfo = uinfo'\ ainfo' = ainfo$
using *assms* **by**(*auto simp add*: *hf-valid-invert*)

3.3.2 Definitions and properties of the added intruder knowledge

Here we define a sets which are added to the intruder knowledge: *ik-add*, which contains hop authenticators.

sublocale *dataplane-3-directed-defs - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf*
by *unfold-locales*

declare *TW.holds-set-list*[*dest*]
declare *TW.holds-takeW-is-identity*[*simp*]
declare *parts-singleton*[*dest*]

This additional Intruder Knowledge allows us to model the attacker's access not only to the hop validation fields and segment identifiers of authorized segments (which are already given in *ik-auth-hfs*), but to the underlying hop authenticators that are used to create them.

definition *ik-add* :: *msgterm set* **where**
$ik\text{-}add \equiv \{\ \sigma \mid ainfo\ uinfo\ l\ hf\ \sigma.$
$(ainfo,\ l) \in auth\text{-}seg2 \wedge hf \in set\ l \wedge HVF\ hf = Mac[\sigma]\ \langle ainfo,\ uinfo\rangle\ \}$

lemma *ik-addI*:
$[\![(ainfo,\ l) \in auth\text{-}seg2;\ hf \in set\ l;\ HVF\ hf = Mac[\sigma]\ \langle ainfo,\ uinfo\rangle]\!] \Longrightarrow \sigma \in ik\text{-}add$
by(*auto simp add*: *ik-add-def*)

lemma *ik-add-form*: $t \in ik\text{-}add \Longrightarrow \exists\ asid\ l\ .\ t = Mac[macKey\ asid]\ l$
by(*auto simp add*: *ik-add-def auth-seg2-def hf-valid-invert dest*!: *TW.holds-set-list*)

lemma *parts-ik-add*[*simp*]: $parts\ ik\text{-}add = ik\text{-}add$
by (*auto intro*!: *parts-Hash dest*: *ik-add-form*)

abbreviation *ik-oracle* :: *msgterm set* **where** $ik\text{-}oracle \equiv \{\}$

3.3.3 Properties of the intruder knowledge, including *ik-add* and *ik-oracle*

We now instantiate the parametrized model's definition of the intruder knowledge, using the definitions of *ik-add* and *ik-oracle* from above. We then prove the properties that we need to instantiate the *dataplane-3-directed* locale.

sublocale

dataplane-3-directed-ik-defs - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo
ik-hf ik-add ik-oracle no-oracle
by *unfold-locales*

lemma *ik-auth-hfs-form*: $t \in$ *parts ik-auth-hfs* $\Longrightarrow \exists\ t'\ .\ t = \mathit{Hash}\ t'$
apply *auto* **apply**(*drule parts-singleton*)
by(*auto simp add*: *auth-seg2-def hf-valid-invert dest*!: *TW.holds-set-list*)

declare *ik-auth-hfs-def* [*simp del*]

lemma *parts-ik-auth-hfs*[*simp*]: *parts ik-auth-hfs* = *ik-auth-hfs*
by (*auto intro*!: *parts-Hash ik-auth-hfs-form*)

This lemma allows us not only to expand the definition of *ik-auth-hfs*, but also to obtain useful properties, such as a term being a Hash, and it being part of a valid hop field.

lemma *ik-auth-hfs-simp*:
$t \in$ *ik-auth-hfs* $\longleftrightarrow (\exists t'\ .\ t = \mathit{Hash}\ t') \wedge (\exists \mathit{hf}\ .\ (t = \mathit{HVF}\ \mathit{hf} \vee t = \mathit{UHI}\ \mathit{hf})$
$\wedge\ (\exists \mathit{hfs}.\ \mathit{hf} \in \mathit{set}\ \mathit{hfs} \wedge (\exists \mathit{ainfo}\ .\ (\mathit{ainfo}, \mathit{hfs}) \in \mathit{auth\text{-}seg2}$
$\wedge\ (\exists\ \mathit{prev}\ \mathit{nxt}\ \mathit{uinfo}.\ \mathit{hf\text{-}valid}\ \mathit{ainfo}\ \mathit{uinfo}\ \mathit{prev}\ \mathit{hf}\ \mathit{nxt}))))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *asm*: *?lhs*
then obtain *ainfo hf hfs* **where**
dfs: $\mathit{hf} \in \mathit{set}\ \mathit{hfs}\ (\mathit{ainfo}, \mathit{hfs}) \in \mathit{auth\text{-}seg2}\ t = \mathit{HVF}\ \mathit{hf} \vee t = \mathit{UHI}\ \mathit{hf}$
by(*auto simp add*: *ik-auth-hfs-def*)
then obtain *uinfo* **where** *hfs-valid-None ainfo uinfo hfs* (*ainfo*, *AHIS hfs*) $\in$ *auth-seg0*
by(*auto simp add*: *auth-seg2-def*)
then show *?rhs* **using** *asm dfs* **by**(*fast intro*: *ik-auth-hfs-form*)
qed(*auto simp add*: *ik-auth-hfs-def*)

Properties of Intruder Knowledge

lemma *auth-ainfo*[*dest*]: $[\![(\mathit{ainfo}, \mathit{hfs}) \in \mathit{auth\text{-}seg2}]\!] \Longrightarrow \exists\ \mathit{ts}\ .\ \mathit{ainfo} = \mathit{Num}\ \mathit{ts}$
by(*auto simp add*: *auth-seg2-def*)

lemma *Num-ik*[*intro*]: *Num ts* $\in$ *ik*
by(*auto simp add*: *ik-def*)
(*auto simp add*: *auth-seg2-def TW.holds.simps*(*3*) *intro*!: *exI*[*of - []*])

There are no ciphertexts (or signatures) in *parts ik*. Thus, *analz ik* and *parts ik* are identical.

lemma *analz-parts-ik*[*simp*]: *analz ik* = *parts ik*
apply(*rule no-crypt-analz-is-parts*)
by(*auto simp add*: *ik-def auth-seg2-def ik-auth-hfs-simp*)
(*auto 3 4 simp add*: *ik-add-def auth-seg2-def hf-valid-invert*)

lemma *parts-ik*[*simp*]: *parts ik* = *ik*
by(*auto 3 4 simp add*: *ik-def auth-seg2-def*)

lemma *key-ik-bad*: *Key* (*macK asid*) $\in$ *ik* $\Longrightarrow$ *asid* $\in$ *bad*
by(*auto simp add*: *ik-def hf-valid-invert*)
(*auto 3 4 simp add*: *auth-seg2-def ik-auth-hfs-simp ik-add-def hf-valid-invert*)

Updating hop fields with different uinfo

Those hop validation fields contained in *auth-seg2* or that can be generated from the hop authenticators in *ik-add* have the property that they are agnostic about the uinfo field. If a hop validation field is contained in *auth-seg2* (resp. derivable from *ik-add*), then a field with a different uinfo is also contained (resp. derivable). To show this, we first define a function that updates uinfo in a hop validation field.

fun *uinfo-upd-hf* :: *msgterm* ⇒ (*unit*, *msgterm*) *HF* ⇒ (*unit*, *msgterm*) *HF* **where**
  uinfo-upd-hf new-uinfo hf =
    (*case HVF hf of Mac*[σ] ⟨*ainfo*, *uinfo*⟩ ⇒ *hf*⦇*HVF* := *Mac*[σ] ⟨*ainfo*, *new-uinfo*⟩⦈ | - ⇒ *hf*)

fun *uinfo-upd* :: *msgterm* ⇒ (*unit*, *msgterm*) *HF list* ⇒ (*unit*, *msgterm*) *HF list* **where**
  uinfo-upd new-uinfo hfs = *map* (*uinfo-upd-hf new-uinfo*) *hfs*

lemma *uinfo-upd-valid*:
  hfs-valid ainfo uinfo pre l nxt ⟹ *hfs-valid ainfo new-uinfo pre* (*uinfo-upd new-uinfo l*) *nxt*
  apply(*induction pre l nxt rule*: *TW.holds.induct*)
  apply *auto*
  subgoal for *prev x y ys z*
    by(*cases map* (*uinfo-upd-hf new-uinfo*) *ys*)
      (*auto simp add*: *TW.holds-split-tail hf-valid-invert*)
  by(*auto 3 4 simp add*: *TW.holds-split-tail hf-valid-invert TW.holds.simps*(*3*))

lemma *uinfo-upd-hf-AHI*: *AHI* (*uinfo-upd-hf new-uinfo hf*) = *AHI hf*
  apply(*cases HVF hf*) **apply** *auto*
  subgoal for *x* **apply**(*cases x*) **apply** *auto*
    subgoal for *x1 x2* **apply**(*cases x2*) **by** *auto*
    done
  done

lemma *uinfo-upd-hf-AHIS*[*simp*]: *AHIS* (*map* (*uinfo-upd-hf new-uinfo*) *l*) = *AHIS l*
  apply(*induction l*) **using** *uinfo-upd-hf-AHI* **by** *auto*

lemma *uinfo-upd-auth-seg2*:
  assumes *hf-valid ainfo uinfo prev m z* σ = *Mac*[*Key* (*macK asid*)] *j*
      HVF m = *Mac*[σ] ⟨*ainfo*, *uinfo'*⟩ σ ∈ *ik-add*
  shows ∃ *hfs*. *m* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
proof−
  from *assms*(*4*) **obtain** *ainfo-add uinfo-add l-add hf-add* **where**
    (*ainfo-add*, *l-add*) ∈ *auth-seg2 hf-add* ∈ *set l-add HVF hf-add* = *Mac*[σ] ⟨*ainfo-add*, *uinfo-add*⟩
    by(*auto simp add*: *ik-add-def*)
  then have *add*: *m* ∈ *set* (*uinfo-upd uinfo l-add*) (*ainfo-add*, (*uinfo-upd uinfo l-add*)) ∈ *auth-seg2*
    using *assms*(*1*−*3*) **apply**(*auto simp add*: *auth-seg2-def simp del*: *AHIS-def*)
    apply(*auto simp add*: *hf-valid-invert intro*!: *image-eqI dest*!: *TW.holds-set-list*)[*1*]
    by(*auto intro*!: *exI elim*: *ahi-eq dest*: *uinfo-upd-valid simp del*: *AHIS-def*)
  then have *ainfo-add* = *ainfo*
    using *assms*(*1*) **by**(*auto simp add*: *auth-seg2-def dest*!: *TW.holds-set-list dest*: *info-hvf*)
  then show *?thesis* **using** *add* **by** *fastforce*
qed

lemma *MAC-synth-helper*:
⟦*hf-valid ainfo uinfo prev m z*;

HVF m = *Mac*[σ] ⟨*ainfo*, *uinfo*⟩; σ = *Mac*[*Key* (*macK asid*)] *j*; σ ∈ *ik* ∨ *HVF m* ∈ *ik*⟧
⟹ ∃ *hfs*. *m* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
apply(*auto simp add*: *ik-def ik-auth-hfs-simp dest*: *ik-add-form*)
prefer *3* **subgoal by**(*auto elim*!: *uinfo-upd-auth-seg2*)
prefer *3* **subgoal by**(*auto elim*!: *uinfo-upd-auth-seg2 intro*: *ik-addI dest*: *info-hvf HOL.sym*)
by(*auto simp add*: *hf-valid-invert*)

This definition helps with the limiting the number of cases generated. We don't require it, but it is convenient. Given a hop validation field and an asid, return if the hvf has the expected format.

definition *mac-format* :: *msgterm* ⇒ *as* ⇒ *bool* **where**
mac-format m asid ≡ ∃ *j ts uinfo* . *m* = *Mac*[*Mac*[*macKey asid*] *j*] ⟨*Num ts*, *uinfo*⟩

If a valid hop field is derivable by the attacker, but does not belong to the attacker, then the hop field is already contained in the set of authorized segments.

lemma *MAC-synth*:
assumes *hf-valid ainfo uinfo prev m z HVF m* ∈ *synth ik mac-format* (*HVF m*) *asid*
asid ∉ *bad checkInfo ainfo*
shows ∃ *hfs* . *m* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
using *assms*
apply(*auto simp add*: *mac-format-def elim*!: *MAC-synth-helper dest*!: *key-ik-bad*)
apply(*auto simp add*: *ik-def ik-auth-hfs-simp dest*: *ik-add-form*)
using *assms*(*1*) **by**(*auto dest*: *info-hvf simp add*: *hf-valid-invert*)

3.3.4 Direct proof goals for interpretation of *dataplane-3-directed*

lemma *COND-honest-hf-analz*:
assumes *ASID* (*AHI hf*) ∉ *bad hf-valid ainfo uinfo prev hf nxt ik-hf hf* ⊆ *synth* (*analz ik*)
no-oracle ainfo uinfo
shows *ik-hf hf* ⊆ *analz ik*
proof−
let *?asid* = *ASID* (*AHI hf*)
from *assms*(*3*) **have** *hf-synth-ik*: *HVF hf* ∈ *synth ik UHI hf* ∈ *synth ik* **by** *auto*
from *assms*(*2*) **have** *mac-format* (*HVF hf*) *?asid checkInfo ainfo*
by(*auto simp add*: *mac-format-def hf-valid-invert*)
then obtain *hfs* **where** *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2*
using *assms*(*1*,*2*,*4*) *hf-synth-ik* **by**(*auto dest*!: *MAC-synth*)
then have *HVF hf* ∈ *ik UHI hf* ∈ *ik*
using *assms*(*2*)
by(*auto simp add*: *ik-auth-hfs-def intro*!: *ik-ik-auth-hfs intro*!: *exI*)
then show *?thesis* **by** *auto*
qed

lemma *COND-ainfo-analz*:
assumes *hf-valid ainfo uinfo prev hf nxt* **and** *ik-auth-ainfo ainfo* ∈ *synth* (*analz ik*)
shows *ik-auth-ainfo ainfo* ∈ *analz ik*
using *assms* **by**(*auto simp add*: *hf-valid-invert*)

lemma *COND-ik-hf*:
assumes *hf-valid ainfo uinfo prev hf z* **and** *HVF hf* ∈ *ik* **and** *no-oracle ainfo uinfo*
shows ∃ *hfs*. *hf* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
proof−

from *assms* **have** *checkInfo ainfo* **by** *auto*
then obtain *hfs ainfo* **where** *hfs-def*: $hf \in set\ hfs\ (ainfo, hfs) \in auth\text{-}seg2$
using *assms* **by**(*auto 3 4 simp add*: *hf-valid-invert ik-auth-hfs-simp ik-def dest*: *ahi-eq dest*!: *ik-add-form*)
then obtain *hfs ainfo* **where** *hfs-def*: $hf \in set\ hfs\ (ainfo, hfs) \in auth\text{-}seg2$ **by** *auto*
show *?thesis*
using *hfs-def* **apply** (*auto simp add*: *auth-seg2-def dest*!: *TW.holds-set-list*)
using *hfs-def assms*(*1*) **by** (*auto simp add*: *auth-seg2-def dest*: *info-hvf*)
qed

lemma *COND-extr-prefix-path*:
⟦*hfs-valid ainfo uinfo pre l nxt*; *nxt* = *None*⟧ ⟹ *prefix* (*extr-from-hd l*) (*AHIS l*)
by(*induction pre l nxt rule*: *TW.holds.induct*)
(*auto simp add*: *TW.holds-split-tail TW.holds.simps*(*1*) *hf-valid-invert*,
auto split: *list.split-asm simp add*: *hf-valid-invert intro*!: *ahi-eq elim*: *ASIF.elims*)

lemma *COND-path-prefix-extr*:
prefix (*AHIS* (*hfs-valid-prefix ainfo uinfo pre l nxt*))
(*extr-from-hd l*)
apply(*induction pre l nxt rule*: *TW.takeW.induct*[**where** *?Pa=hf-valid ainfo uinfo*])
apply(*auto simp add*: *TW.takeW-split-tail TW.takeW.simps*(*1*))
apply(*auto simp add*: *hf-valid-invert intro*!: *ahi-eq*)
by(*auto elim*: *ASIF.elims*)

lemma *COND-hf-valid-no-prev*:
hf-valid ainfo uinfo prev hf z ⟷ *hf-valid ainfo uinfo prev′ hf z*
by(*auto simp add*: *hf-valid-invert*)

lemma *COND-hf-valid-uinfo*:
⟦*hf-valid ainfo uinfo pre hf nxt*; *hf-valid ainfo′ uinfo′ pre′ hf nxt′*⟧ ⟹ *uinfo′* = *uinfo*
by(*auto dest*: *info-hvf*)

3.3.5 Instantiation of *dataplane-3-directed* locale

sublocale
dataplane-3-directed - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf ik-add ik-oracle no-oracle
apply *unfold-locales*
using *COND-ik-hf COND-honest-hf-analz COND-ainfo-analz COND-extr-prefix-path COND-path-prefix-extr COND-hf-valid-no-prev COND-hf-valid-uinfo* **by** *auto*

end
end

3.4 EPIC Level 1 in the Strong Attacker Model

theory *EPIC-L1-SA*
  imports
    ../Parametrized-Dataplane-3-directed
    ../infrastructure/Keys
begin

locale *epic-l1-defs* = *network-assums-direct* - - - *auth-seg0*
  for *auth-seg0* :: (*msgterm* × *ahi list*) *set* +
  fixes *no-oracle* :: *msgterm* ⇒ *msgterm* ⇒ *bool*
begin

3.4.1 Hop validation check and extract functions

The predicate *hf-valid* is given to the concrete parametrized model as a parameter. It ensures the authenticity of the hop authenticator in the hop field. The predicate takes an authenticated info field (in this model always a numeric value, hence the matching on Num ts), an unauthenticated info field uinfo, the hop field to be validated and in some cases the next hop field.

We distinguish if there is a next hop field (this yields the two cases below). If there is not, then the hop authenticator σ simply consists of a MAC over the authenticated info field and the local routing information of the hop, using the key of the hop to which the hop field belongs. If on the other hand, there is a subsequent hop field, then the uhi field of that hop field is also included in the MAC computation.

The hop authenticator σ is used to compute both the hop validation field and the uhi field. The first is computed as a MAC over the path origin (pair of absolute timestamp ts and the relative timestamp given in uinfo), using the hop authenticator as a key to the MAC. The hop authenticator is not secret, and any end host can use it to create a valid hvf. The uhi field, according to the protocol description, is σ shortened to a few bytes. We model this as applying the hash on σ.

The predicate *hf-valid* checks if the hop authenticator, hvf and uhi field are computed correctly.

fun *hf-valid* :: *msgterm* ⇒ *msgterm*
    ⇒ (*unit*, *msgterm*) *HF option*
    ⇒ (*unit*, *msgterm*) *HF*
    ⇒ (*unit*, *msgterm*) *HF option* ⇒ *bool* **where**
  hf-valid (*Num ts*) *uinfo* - ⦇*AHI* = *ahi*, *UHI* = *uhi*, *HVF* = *x*⦈ (*Some* ⦇*AHI* = *ahi2*, *UHI* = *uhi2*, *HVF* = *x2*⦈) ⟷
    (∃ σ *upif downif*. σ = *Mac*[*macKey* (*ASID ahi*)] (*L* [*Num ts*, *upif*, *downif*, *uhi2*]) ∧
      ASIF (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif* ∧ *uhi* = *Hash* σ ∧ *x* = *Mac*[σ] ⟨*Num ts*, *uinfo*⟩)
| *hf-valid* (*Num ts*) *uinfo* - ⦇*AHI* = *ahi*, *UHI* = *uhi*, *HVF* = *x*⦈ *None* ⟷
    (∃ σ *upif downif*. σ = *Mac*[*macKey* (*ASID ahi*)] (*L* [*Num ts*, *upif*, *downif*]) ∧
      ASIF (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif* ∧ *uhi* = *Hash* σ ∧ *x* = *Mac*[σ] ⟨*Num ts*, *uinfo*⟩)
| *hf-valid* - - - - - = *False*

We can extract the entire path from the uhi field, since it includes the hop authenticator, which includes the local forwarding information as well as, recursively, all upstream hop authenticators and their hop information. However, the parametrized model defines the extract function to operate on the hop validation field, not the uhi field. We therefore define a separate function that extracts the path from a hvf. We can do so, as both hvf and uhi contain the hop authenticator. Internally, that function uses *extrUhi*.

fun *extrUhi* :: *msgterm* ⇒ *ahi list* **where**
  extrUhi (*Hash* (*Mac*[*macKey asid*] (*L* [*ts*, *upif*, *downif*, *uhi2*])))
 = (| *UpIF* = *ASO upif*, *DownIF* = *ASO downif*, *ASID* = *asid*|) # *extrUhi uhi2*
| *extrUhi* (*Hash* (*Mac*[*macKey asid*] (*L* [*ts*, *upif*, *downif*])))
 = [(| *UpIF* = *ASO upif*, *DownIF* = *ASO downif*, *ASID* = *asid*|)]
| *extrUhi* - = []

This function extracts from a hop validation field (HVF hf) the entire path.

fun *extr* :: *msgterm* ⇒ *ahi list* **where**
  extr (*Mac*[σ] -) = *extrUhi* (*Hash* σ)
  | *extr* - = []

Extract the authenticated info field from a hop validation field.

fun *extr-ainfo* :: *msgterm* ⇒ *msgterm* **where**
  extr-ainfo (*Mac*[-] ⟨*Num ts*, -⟩) = *Num ts*
| *extr-ainfo* - = ε

abbreviation *ik-auth-ainfo* :: *msgterm* ⇒ *msgterm* **where**
  ik-auth-ainfo ≡ *id*

An authenticated info field is always a number (corresponding to a timestamp).

abbreviation *checkInfo* **where**
  checkInfo ainfo ≡ (∃ *ts*. *ainfo* = *Num ts*)

When observing a hop field, an attacker learns the HVF and the UHI. The AHI only contains public information that are not terms.

fun *ik-hf* :: (*unit*, *msgterm*) *HF* ⇒ *msgterm set* **where**
  ik-hf hf = {*HVF hf*, *UHI hf*}

We now define useful properties of the above definition.

lemma *hf-valid-invert*:
  hf-valid tsn uinfo prev hf mo ⟷
   ((∃ *ahi ahi2* σ *ts upif downif asid x upif2 downif2 asid2 uhi uhi2 x2*.
     hf = (|*AHI* = *ahi*, *UHI* = *uhi*, *HVF* = *x*|) ∧
     ASID ahi = *asid* ∧ *ASIF* (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif* ∧
     mo = *Some* (|*AHI* = *ahi2*, *UHI* = *uhi2*, *HVF* = *x2*|) ∧
     ASID ahi2 = *asid2* ∧ *ASIF* (*DownIF ahi2*) *downif2* ∧ *ASIF* (*UpIF ahi2*) *upif2* ∧
     σ = *Mac*[*macKey asid*] (*L* [*tsn*, *upif*, *downif*, *uhi2*]) ∧
     tsn = *Num ts* ∧
     uhi = *Hash* σ ∧
     x = *Mac*[σ] ⟨*tsn*, *uinfo*⟩)
 ∨ (∃ *ahi* σ *ts upif downif asid uhi x*.
     hf = (|*AHI* = *ahi*, *UHI* = *uhi*, *HVF* = *x*|) ∧
     ASID ahi = *asid* ∧ *ASIF* (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif* ∧
     mo = *None* ∧
     σ = *Mac*[*macKey asid*] (*L* [*tsn*, *upif*, *downif*]) ∧

$tsn = Num\ ts \wedge$
$uhi = Hash\ \sigma \wedge$
$x = Mac[\sigma]\ \langle tsn,\ uinfo \rangle)$
)
apply(*auto elim*!: *hf-valid.elims*) **using** *option.exhaust ASIF.simps* **by** *metis+*

lemma *hf-valid-checkInfo*[*dest*]: *hf-valid ainfo uinfo prev hf z* $\Longrightarrow$ *checkInfo ainfo*
by(*auto simp add*: *hf-valid-invert*)

lemma *info-hvf*:
assumes *hf-valid ainfo uinfo prev m z HVF m* $=$ *Mac*[σ] $\langle$*ainfo'*, *uinfo'*$\rangle$ $\vee$ *hf-valid ainfo' uinfo' prev' m z'*
shows *uinfo* $=$ *uinfo' ainfo'* $=$ *ainfo*
using *assms* **by**(*auto simp add*: *hf-valid-invert*)

sublocale *dataplane-3-directed-defs - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf*
by *unfold-locales*

3.4.2 Definitions and properties of the added intruder knowledge

Here we define two sets which are added to the intruder knowledge: *ik-add*, which contains hop authenticators. And *ik-oracle*, which contains the oracle's output to the strong attacker.

abbreviation *is-oracle* **where** *is-oracle ainfo t* $\equiv \neg$ *no-oracle ainfo t*

declare *TW.holds-set-list*[*dest*]
declare *TW.holds-takeW-is-identity*[*simp*]
declare *parts-singleton*[*dest*]

This additional Intruder Knowledge allows us to model the attacker's access not only to the hop validation fields and segment identifiers of authorized segments (which are already given in *ik-auth-hfs*), but to the underlying hop authenticators that are used to create them.

definition *ik-add* :: *msgterm set* **where**
ik-add $\equiv$ { σ | *ainfo uinfo l hf* σ.
(*ainfo*::*msgterm*, *l*::((*unit*, *msgterm*) *HF list*)) $\in$
(*local.auth-seg2*::((*msgterm* $\times$ (*unit*, *msgterm*) *HF list*) *set*))
$\wedge$ *hf* $\in$ *set l* $\wedge$ *HVF hf* $=$ *Mac*[σ] $\langle$*ainfo*, *uinfo*$\rangle$ }

lemma *ik-addI*:
$[\![$(*ainfo*, *l*) $\in$ *local.auth-seg2*; *hf* $\in$ *set l*; *HVF hf* $=$ *Mac*[σ] $\langle$*ainfo*, *uinfo*$\rangle]\!]$ $\Longrightarrow \sigma \in$ *ik-add*
by(*auto simp add*: *ik-add-def*)

lemma *ik-add-form*: *t* $\in$ *local.ik-add* $\Longrightarrow \exists$ *asid l* . *t* $=$ *Mac*[*macKey asid*] *l*

by(*auto simp add*: *ik-add-def auth-seg2-def dest*!: *TW.holds-set-list*)
(*auto simp add*: *hf-valid-invert*)

lemma *parts-ik-add*[*simp*]: *parts ik-add* $=$ *ik-add*
by (*auto intro*!: *parts-Hash dest*: *ik-add-form*)

This is the oracle output provided to the adversary. Only those hop validation fields and segment identifiers whose path origin (combination of ainfo uinfo) is not contained in *no-oracle* appears here.

definition *ik-oracle* :: *msgterm set* **where**
ik-oracle = $\{t \mid t$ *ainfo hf l uinfo* . *hf* $\in$ *set l* $\wedge$ *hfs-valid-None ainfo uinfo l* $\wedge$
is-oracle ainfo uinfo $\wedge$ (*ainfo*, *l*) $\notin$ *auth-seg2* $\wedge$ (t = *HVF hf* $\vee$ t = *UHI hf*) $\}$

lemma *ik-oracle-parts-form*:
$t \in$ *ik-oracle* $\Longrightarrow$
($\exists$ *asid l ainfo uinfo* . t = *Mac*[*Mac*[*macKey asid*] *l*] $\langle$*ainfo*, *uinfo*$\rangle$) $\vee$
($\exists$ *asid l* . t = *Hash* (*Mac*[*macKey asid*] *l*))
by(*auto simp add*: *ik-oracle-def hf-valid-invert dest*!: *TW.holds-set-list*)

lemma *parts-ik-oracle*[*simp*]: *parts ik-oracle* = *ik-oracle*
by (*auto intro*!: *parts-Hash dest*: *ik-oracle-parts-form*)

lemma *ik-oracle-simp*: $t \in$ *ik-oracle* $\longleftrightarrow$
($\exists$ *ainfo hf l uinfo*. *hf* $\in$ *set l* $\wedge$ *hfs-valid-None ainfo uinfo l* $\wedge$ *is-oracle ainfo uinfo*
$\wedge$ (*ainfo*, *l*) $\notin$ *auth-seg2* $\wedge$ (t = *HVF hf* $\vee$ t = *UHI hf*))
by(*rule iffI*, *frule ik-oracle-parts-form*)
(*auto simp add*: *ik-oracle-def hf-valid-invert*)

3.4.3 Properties of the intruder knowledge, including *ik-add* and *ik-oracle*

We now instantiate the parametrized model's definition of the intruder knowledge, using the definitions of *ik-add* and *ik-oracle* from above. We then prove the properties that we need to instantiate the *dataplane-3-directed* locale.

sublocale
dataplane-3-directed-ik-defs - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo
ik-hf ik-add ik-oracle no-oracle
by *unfold-locales*

lemma *ik-auth-hfs-form*: $t \in$ *parts ik-auth-hfs* $\Longrightarrow \exists\ t'$. t = *Hash* t'
apply *auto* **apply**(*drule parts-singleton*)
by(*auto simp add*: *auth-seg2-def hf-valid-invert dest*!: *TW.holds-set-list*)

declare *ik-auth-hfs-def* [*simp del*]

lemma *parts-ik-auth-hfs*[*simp*]: *parts ik-auth-hfs* = *ik-auth-hfs*
by (*auto intro*!: *parts-Hash ik-auth-hfs-form*)

This lemma allows us not only to expand the definition of *ik-auth-hfs*, but also to obtain useful properties, such as a term being a Hash, and it being part of a valid hop field.

lemma *ik-auth-hfs-simp*:
$t \in$ *ik-auth-hfs* $\longleftrightarrow$ ($\exists\, t'$. t = *Hash* t') $\wedge$ ($\exists$ *hf* . (t = *HVF hf* $\vee$ t = *UHI hf*)
$\wedge$ ($\exists$ *hfs*. *hf* $\in$ *set hfs* $\wedge$ ($\exists$ *ainfo* . (*ainfo*, *hfs*) $\in$ *auth-seg2*
$\wedge$ ($\exists$ *prev nxt uinfo*. *hf-valid ainfo uinfo prev hf nxt*)))) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *asm*: *?lhs*
then obtain *ainfo hf hfs* **where**
dfs: *hf* $\in$ *set hfs* (*ainfo*, *hfs*) $\in$ *auth-seg2* t = *HVF hf* $\vee$ t = *UHI hf*
by(*auto simp add*: *ik-auth-hfs-def*)
then obtain *uinfo* **where** *hfs-valid-None ainfo uinfo hfs* (*ainfo*, *AHIS hfs*) $\in$ *auth-seg0*
by(*auto simp add*: *auth-seg2-def*)
then show *?rhs* **using** *asm dfs* **by**(*fast intro*: *ik-auth-hfs-form*)

qed(*auto simp add*: *ik-auth-hfs-def*)

Properties of Intruder Knowledge

lemma *auth-ainfo*[*dest*]: $[\![(ainfo,\ hfs) \in auth\text{-}seg2]\!] \Longrightarrow \exists\ ts\ .\ ainfo = Num\ ts$
  by(*auto simp add*: *auth-seg2-def*)

lemma *Num-ik*[*intro*]: $Num\ ts \in ik$
  by(*auto simp add*: *ik-def*)
    (*auto simp add*: *auth-seg2-def TW.holds.simps*(*3*) *elim*: *allE*[*of* - []])

There are no ciphertexts (or signatures) in *parts ik*. Thus, *analz ik* and *parts ik* are identical.

lemma *analz-parts-ik*[*simp*]: *analz ik* = *parts ik*
  apply(*rule no-crypt-analz-is-parts*)
  by(*auto simp add*: *ik-def auth-seg2-def ik-auth-hfs-simp*)
    (*auto simp add*: *ik-add-def ik-oracle-def auth-seg2-def hf-valid-invert hfs-valid-prefix-generic-def*
      dest!: *TW.holds-set-list*)

lemma *parts-ik*[*simp*]: *parts ik* = *ik*
  by(*auto 3 4 simp add*: *ik-def auth-seg2-def*)

lemma *key-ik-bad*: $Key\ (macK\ asid) \in ik \Longrightarrow asid \in bad$
  by(*auto simp add*: *ik-def hf-valid-invert ik-oracle-simp*)
    (*auto 3 4 simp add*: *auth-seg2-def ik-auth-hfs-simp ik-add-def hf-valid-invert*)

Updating hop fields with different uinfo

Those hop validation fields contained in *auth-seg2* or that can be generated from the hop authenticators in *ik-add* have the property that they are agnostic about the uinfo field. If a hop validation field is contained in *auth-seg2* (resp. derivable from *ik-add*), then a field with a different uinfo is also contained (resp. derivable). To show this, we first define a function that updates uinfo in a hop validation field.

fun *uinfo-upd-hf* :: $msgterm \Rightarrow (unit,\ msgterm)\ HF \Rightarrow (unit,\ msgterm)\ HF$ **where**
  uinfo-upd-hf new-uinfo hf =
    $(case\ HVF\ hf\ of\ Mac[\sigma]\ \langle ainfo,\ uinfo\rangle \Rightarrow hf(\!|HVF := Mac[\sigma]\ \langle ainfo,\ new\text{-}uinfo\rangle|\!)\ |\ \text{-} \Rightarrow hf)$

fun *uinfo-upd* :: $msgterm \Rightarrow (unit,\ msgterm)\ HF\ list \Rightarrow (unit,\ msgterm)\ HF\ list$ **where**
  uinfo-upd new-uinfo hfs = *map* (*uinfo-upd-hf new-uinfo*) *hfs*

lemma *uinfo-upd-valid*:
  hfs-valid ainfo uinfo pre l nxt $\Longrightarrow$ *hfs-valid ainfo new-uinfo pre* (*uinfo-upd new-uinfo l*) *nxt*
  apply(*induction pre l nxt rule*: *TW.holds.induct*)
  apply *auto*
  subgoal for *prev x y ys z*
    by(*cases map* (*uinfo-upd-hf new-uinfo*) *ys*)
      (*auto simp add*: *TW.holds-split-tail hf-valid-invert*)
  by(*auto 3 4 simp add*: *TW.holds-split-tail hf-valid-invert TW.holds.simps*(*3*))

lemma *uinfo-upd-hf-AHI*: *AHI* (*uinfo-upd-hf new-uinfo hf*) = *AHI hf*
  apply(*cases HVF hf*) **apply** *auto*
  subgoal for *x* **apply**(*cases x*) **apply** *auto*
    subgoal for *x1 x2* **apply**(*cases x2*) **by** *auto*

done
done

lemma *uinfo-upd-hf-AHIS*[*simp*]: $AHIS\ (map\ (uinfo\text{-}upd\text{-}hf\ new\text{-}uinfo)\ l) = AHIS\ l$
apply(*induction l*) **using** *uinfo-upd-hf-AHI* **by** *auto*

lemma *uinfo-upd-auth-seg2*:
assumes $hf\text{-}valid\ ainfo\ uinfo\ prev\ m\ z\ \sigma = Mac[Key\ (macK\ asid)]\ j$
$HVF\ m = Mac[\sigma]\ \langle ainfo,\ uinfo'\rangle\ \sigma \in ik\text{-}add$
shows $\exists hfs.\ m \in set\ hfs \wedge (ainfo,\ hfs) \in auth\text{-}seg2$
proof−
from *assms*(*4*) **obtain** *ainfo-add uinfo-add l-add hf-add* **where**
$(ainfo\text{-}add,\ l\text{-}add) \in auth\text{-}seg2\ hf\text{-}add \in set\ l\text{-}add\ HVF\ hf\text{-}add = Mac[\sigma]\ \langle ainfo\text{-}add,\ uinfo\text{-}add\rangle$
by(*auto simp add*: *ik-add-def*)
then have *add*: $m \in set\ (uinfo\text{-}upd\ uinfo\ l\text{-}add)\ (ainfo\text{-}add,\ (uinfo\text{-}upd\ uinfo\ l\text{-}add)) \in auth\text{-}seg2$
using *assms*(*1*−*3*) **apply**(*auto simp add*: *auth-seg2-def simp del*: *AHIS-def*)
apply(*auto simp add*: *hf-valid-invert intro*!: *image-eqI dest*!: *TW.holds-set-list*)[*1*]
by(*auto intro*!: *exI elim*: *ahi-eq dest*: *uinfo-upd-valid simp del*: *AHIS-def*)
then have $ainfo\text{-}add = ainfo$
using *assms*(*1*) **by**(*auto simp add*: *auth-seg2-def dest*!: *TW.holds-set-list dest*: *info-hvf*)
then show *?thesis* **using** *add* **by** *fastforce*
qed

lemma *MAC-synth-oracle*:
assumes $hf\text{-}valid\ ainfo\ uinfo\ prev\ m\ z\ HVF\ m \in ik\text{-}oracle$
shows *is-oracle ainfo uinfo*
using *assms* **by**(*auto simp add*: *ik-oracle-def assms*(*1*) *hf-valid-invert dest*!: *TW.holds-set-list*)

lemma *ik-oracle-is-oracle*:
$[\![Mac[\sigma]\ \langle ainfo,\ uinfo\rangle \in ik\text{-}oracle]\!] \Longrightarrow is\text{-}oracle\ ainfo\ uinfo$
by (*auto simp add*: *ik-oracle-def dest*: *info-hvf*)
(*auto dest*!: *TW.holds-set-list simp add*: *hf-valid-invert*)

lemma *MAC-synth-helper*:
$[\![hf\text{-}valid\ ainfo\ uinfo\ prev\ m\ z;\ no\text{-}oracle\ ainfo\ uinfo;$
$HVF\ m = Mac[\sigma]\ \langle ainfo,\ uinfo\rangle;\ \sigma = Mac[Key\ (macK\ asid)]\ j;\ \sigma \in ik \vee HVF\ m \in ik]\!]$
$\Longrightarrow \exists hfs.\ m \in set\ hfs \wedge (ainfo,\ hfs) \in auth\text{-}seg2$
apply(*auto simp add*: *ik-def ik-auth-hfs-simp*
dest: *MAC-synth-oracle ik-add-form ik-oracle-parts-form*[*simplified*])
prefer *3* **subgoal by**(*auto elim*!: *uinfo-upd-auth-seg2*)
prefer *3* **subgoal by**(*auto elim*!: *uinfo-upd-auth-seg2 intro*: *ik-addI dest*: *info-hvf HOL.sym*)
by(*auto simp add*: *hf-valid-invert*)

This definition helps with the limiting the number of cases generated. We don't require it, but it is convenient. Given a hop validation field and an asid, return if the hvf has the expected format.

definition $mac\text{-}format :: msgterm \Rightarrow as \Rightarrow bool$ **where**
$mac\text{-}format\ m\ asid \equiv \exists\ j\ ts\ uinfo\ .\ m = Mac[Mac[macKey\ asid]\ j]\ \langle Num\ ts,\ uinfo\rangle$

If a valid hop field is derivable by the attacker, but does not belong to the attacker, and is over a path origin that does not belong to an oracle query, then the hop field is already contained in the set of authorized segments.

lemma *MAC-synth*:
  assumes *hf-valid ainfo uinfo prev m z HVF m* ∈ *synth ik mac-format* (*HVF m*) *asid*
    asid ∉ *bad checkInfo ainfo no-oracle ainfo uinfo*
  shows ∃ *hfs* . *m* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
  using *assms*
  apply(*auto simp add*: *mac-format-def elim*!: *MAC-synth-helper dest*!: *key-ik-bad*)
  apply(*auto simp add*: *ik-def ik-auth-hfs-simp dest*: *ik-add-form dest*!: *ik-oracle-parts-form*)
  using *assms*(*1*) **by**(*auto dest*: *info-hvf simp add*: *hf-valid-invert*)

3.4.4 Direct proof goals for interpretation of *dataplane-3-directed*

lemma *COND-honest-hf-analz*:
  assumes *ASID* (*AHI hf*) ∉ *bad hf-valid ainfo uinfo prev hf nxt ik-hf hf* ⊆ *synth* (*analz ik*)
    no-oracle ainfo uinfo
    shows *ik-hf hf* ⊆ *analz ik*
proof−
  let *?asid* = *ASID* (*AHI hf*)
  from *assms*(*3*) **have** *hf-synth-ik*: *HVF hf* ∈ *synth ik UHI hf* ∈ *synth ik* **by** *auto*
  from *assms*(*2*) **have** *mac-format* (*HVF hf*) *?asid checkInfo ainfo*
    by(*auto simp add*: *mac-format-def hf-valid-invert*)
  then obtain *hfs* **where** *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2*
    using *assms*(*1*,*2*,*4*) *hf-synth-ik* **by**(*auto dest*!: *MAC-synth*)
  then have *HVF hf* ∈ *ik UHI hf* ∈ *ik*
    using *assms*(*2*)
    by(*auto simp add*: *ik-auth-hfs-def intro*!: *ik-ik-auth-hfs intro*!: *exI*)
  then show *?thesis* **by** *auto*
qed

lemma *COND-ainfo-analz*:
  assumes *hf-valid ainfo uinfo prev hf nxt* **and** *ik-auth-ainfo ainfo* ∈ *synth* (*analz ik*)
  shows *ik-auth-ainfo ainfo* ∈ *analz ik*
  using *assms* **by**(*auto simp add*: *hf-valid-invert*)

lemma *COND-ik-hf*:
  assumes *hf-valid ainfo uinfo prev hf z* **and** *HVF hf* ∈ *ik* **and** *no-oracle ainfo uinfo*
  shows ∃ *hfs*. *hf* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
proof−
  from *assms* **have** *checkInfo ainfo* **by** *auto*
  then obtain *hfs ainfo* **where** *hfs-def*: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2*
  using *assms* **by**(*auto 3 4 simp add*: *hf-valid-invert ik-auth-hfs-simp ik-def dest*: *ahi-eq*
                  dest!: *ik-oracle-is-oracle ik-add-form*)
  then obtain *hfs ainfo* **where** *hfs-def*: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2* **by** *auto*
  show *?thesis*
    using *hfs-def* **apply** (*auto simp add*: *auth-seg2-def dest*!: *TW.holds-set-list*)
    using *hfs-def assms*(*1*) **by** (*auto simp add*: *auth-seg2-def dest*: *info-hvf*)
  qed

lemma *COND-extr-prefix-path*:
  ⟦*hfs-valid ainfo uinfo pre l nxt*; *nxt* = *None*⟧ ⟹ *prefix* (*extr-from-hd l*) (*AHIS l*)
  by(*induction pre l nxt rule*: *TW.holds.induct*)
    (*auto simp add*: *TW.holds-split-tail TW.holds.simps*(*1*) *hf-valid-invert*,
     auto split: *list.split-asm simp add*: *hf-valid-invert intro*!: *ahi-eq elim*: *ASIF.elims*)

lemma *COND-path-prefix-extr*:
prefix (*AHIS* (*hfs-valid-prefix ainfo uinfo pre l nxt*))
(*extr-from-hd l*)
apply(*induction pre l nxt rule*: *TW.takeW.induct*[**where** *?Pa=hf-valid ainfo uinfo*])
apply(*auto simp add*: *TW.takeW-split-tail TW.takeW.simps*(*1*))
apply(*auto simp add*: *hf-valid-invert intro*!: *ahi-eq*)
by(*auto elim*: *ASIF.elims*)

lemma *COND-hf-valid-no-prev*:
hf-valid ainfo uinfo prev hf z $\longleftrightarrow$ *hf-valid ainfo uinfo prev' hf z*
by(*auto simp add*: *hf-valid-invert*)

lemma *COND-hf-valid-uinfo*:
$[\![$*hf-valid ainfo uinfo pre hf nxt*; *hf-valid ainfo' uinfo' pre' hf nxt'*$]\!]$ $\Longrightarrow$ *uinfo'* = *uinfo*
by(*auto dest*: *info-hvf*)

3.4.5 Instantiation of *dataplane-3-directed* locale

sublocale
dataplane-3-directed - - - *auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf ik-add ik-oracle no-oracle*
apply *unfold-locales*
using *COND-ik-hf COND-honest-hf-analz COND-ainfo-analz COND-extr-prefix-path COND-path-prefix-extr COND-hf-valid-no-prev COND-hf-valid-uinfo* **by** *auto*

end
end

3.5 EPIC Level 1 Example instantiation of locale

In this theory we instantiate the locale *dataplane0* and thus show that its assumptions are satisfiable. In particular, this involves the assumptions concerning the network. We also instantiate the locale *epic-l1-defs*.

theory *EPIC-L1-SA-Example*
 imports
  EPIC-L1-SA
begin

The network topology that we define is the same as in Fig. 2 of the paper.

abbreviation *nA* :: *as* **where** *nA* ≡ *3*
abbreviation *nB* :: *as* **where** *nB* ≡ *4*
abbreviation *nC* :: *as* **where** *nC* ≡ *5*
abbreviation *nD* :: *as* **where** *nD* ≡ *6*
abbreviation *nE* :: *as* **where** *nE* ≡ *7*
abbreviation *nF* :: *as* **where** *nF* ≡ *8*
abbreviation *nG* :: *as* **where** *nG* ≡ *9*

abbreviation *bad* :: *as set* **where** *bad* ≡ {*nF*}

We assume a complete graph, in which interfaces contain the name of the adjacent AS

fun *tgtas* :: *as* ⇒ *ifs* ⇒ *as option* **where**
 tgtas a i = *Some i*
fun *tgtif* :: *as* ⇒ *ifs* ⇒ *ifs option* **where**
 tgtif a i = *Some a*

3.5.1 Left segment

abbreviation *hiAl* :: *ahi* **where** *hiAl* ≡ (|*UpIF* = *None*, *DownIF* = *Some nB*, *ASID* = *nA*|)
abbreviation *hiBl* :: *ahi* **where** *hiBl* ≡ (|*UpIF* = *Some nA*, *DownIF* = *Some nD*, *ASID* = *nB*|)
abbreviation *hiDl* :: *ahi* **where** *hiDl* ≡ (|*UpIF* = *Some nB*, *DownIF* = *Some nE*, *ASID* = *nD*|)
abbreviation *hiEl* :: *ahi* **where** *hiEl* ≡ (|*UpIF* = *Some nD*, *DownIF* = *Some nF*, *ASID* = *nE*|)
abbreviation *hiFl* :: *ahi* **where** *hiFl* ≡ (|*UpIF* = *Some nE*, *DownIF* = *None*, *ASID* = *nF*|)

3.5.2 Right segment

abbreviation *hiAr* :: *ahi* **where** *hiAr* ≡ (|*UpIF* = *None*, *DownIF* = *Some nB*, *ASID* = *nA*|)
abbreviation *hiBr* :: *ahi* **where** *hiBr* ≡ (|*UpIF* = *Some nA*, *DownIF* = *Some nD*, *ASID* = *nB*|)
abbreviation *hiDr* :: *ahi* **where** *hiDr* ≡ (|*UpIF* = *Some nB*, *DownIF* = *Some nE*, *ASID* = *nD*|)
abbreviation *hiEr* :: *ahi* **where** *hiEr* ≡ (|*UpIF* = *Some nD*, *DownIF* = *Some nG*, *ASID* = *nE*|)
abbreviation *hiGr* :: *ahi* **where** *hiGr* ≡ (|*UpIF* = *Some nE*, *DownIF* = *None*, *ASID* = *nG*|)

abbreviation *hfF-attr-E* :: *ahi set* **where** *hfF-attr-E* ≡ {*hi* . *ASID hi* = *nF* ∧ *UpIF hi* = *Some nE*}
abbreviation *hfF-attr* :: *ahi set* **where** *hfF-attr* ≡ {*hi* . *ASID hi* = *nF*}

abbreviation *leftpath* :: *ahi list* **where**
 leftpath ≡ [*hiFl*, *hiEl*, *hiDl*, *hiBl*, *hiAl*]
abbreviation *rightpath* :: *ahi list* **where**
 rightpath ≡ [*hiGr*, *hiEr*, *hiDr*, *hiBr*, *hiAr*]
abbreviation *rightsegment* **where** *rightsegment* ≡ (*Num 0*, *rightpath*)

abbreviation *leftpath-wormholed* :: *ahi list set* **where**
leftpath-wormholed ≡
{ *xs*@[*hf*, *hiEl*, *hiDl*, *hiBl*, *hiAl*] | *hf xs* . *hf* ∈ *hfF-attr-E* ∧ *set xs* ⊆ *hfF-attr*}

definition *leftsegment-wormholed* :: (*msgterm* × *ahi list*) *set* **where**
leftsegment-wormholed = { (*Num 0*, *leftpath*) | *leftpath* . *leftpath* ∈ *leftpath-wormholed*}

definition *attr-segment* :: (*msgterm* × *ahi list*) *set* **where**
attr-segment = { (*ainfo*, *path*) | *ainfo path* . *set path* ⊆ *hfF-attr*}

definition *auth-seg0* :: (*msgterm* × *ahi list*) *set* **where**
auth-seg0 = *leftsegment-wormholed* ∪ {*rightsegment*} ∪ *attr-segment*

lemma *tgtasif-inv*:
⟦*tgtas u i* = *Some v*; *tgtif u i* = *Some j*⟧ ⟹ *tgtas v j* = *Some u*
⟦*tgtas u i* = *Some v*; *tgtif u i* = *Some j*⟧ ⟹ *tgtif v j* = *Some i*
by *simp+*

locale *no-assumptions-left*
begin

sublocale *d0*: *network-model bad auth-seg0 tgtas tgtif*
apply *unfold-locales*

done

lemma *attr-ifs-valid*: ⟦*ASID y* = *nF*; *set ys* ⊆ *hfF-attr*⟧ ⟹ *d0.ifs-valid* (*Some y*) *ys nxt*

apply(*induction ys arbitrary*: *y*)
apply(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def attr-segment-def TW.holds-split-tail TW.holds.simps*)
subgoal for *a ys y*
by(*cases ys*, *auto*)
done

lemma *attr-ifs-valid'*: ⟦*set ys* ⊆ *hfF-attr*; *pre* = *None*⟧ ⟹ *d0.ifs-valid pre ys nxt*
apply(*induction pre ys nxt rule*: *TW.holds.induct*)
apply(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def attr-segment-def TW.holds-split-tail TW.holds.simps*)
subgoal for *x y ys nxt*
apply(*cases ys*, *auto*)
using *attr-ifs-valid* **by** *auto*
done

lemma *leftpath-ifs-valid*: ⟦*pre* = *None*; *ASID hf* = *nF*; *UpIF hf* = *Some nE*; *set xs* ⊆ *hfF-attr*⟧
⟹ *d0.ifs-valid pre* (*xs* @ [*hf*, *hiEl*, *hiDl*, *hiBl*, *hiAl*]) *nxt*
apply(*auto simp add*: *TW.holds-append*)
using *attr-ifs-valid'* **apply** *blast*
apply(*cases xs*)
apply *auto*

apply(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def attr-segment-def TW.holds-split-tail TW.holds.simps*)
apply *force*+
done

lemma *ASM-if-valid*: $\llbracket$(*info*, *l*) $\in$ *auth-seg0*; *pre* = *None*$\rrbracket$ $\Longrightarrow$ *d0.ifs-valid pre l nxt*
apply(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def attr-segment-def*)
using *leftpath-ifs-valid attr-ifs-valid'* **apply** *simp-all*
by(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def attr-segment-def TW.holds-split-tail TW.holds.simps*)

lemma *rooted-app*[*simp*]: *d0.rooted* (*xs*@*y*#*ys*) $\longleftrightarrow$ *d0.rooted* (*y*#*ys*)
by(*induction xs arbitrary*: *y ys*, *auto*)
(*metis Nil-is-append-conv d0.rooted.simps*(*2*) *d0.terminated.cases*)+

lemma *ASM-rooted*: (*info*, *l*) $\in$ *auth-seg0* $\Longrightarrow$ *d0.rooted l*
apply(*induction l*)
apply(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def attr-segment-def TW.holds-split-tail*)
by (*metis d0.rooted.simps*(*1*) *d0.rooted.simps*(*2*) *d0.terminated.cases insert-iff*)

lemma *ASM-terminated*: (*info*, *l*) $\in$ *auth-seg0* $\Longrightarrow$ *d0.terminated l*
apply(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def TW.holds-split-tail attr-segment-def*)
subgoal for *hf xs*
by(*induction xs*, *auto*)
by(*induction l*, *auto*)

lemma *ASM-empty*: (*info*, []) $\in$ *auth-seg0*
by(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def attr-segment-def*)

lemma *ASM-singleton*: $\llbracket$*ASID hf* $\in$ *bad*$\rrbracket$ $\Longrightarrow$ (*info*, [*hf*]) $\in$ *auth-seg0*
by(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def attr-segment-def*)

lemma *ASM-extension*:
$\llbracket$(*info*, *hf2*#*ys*) $\in$ *auth-seg0*; *ASID hf2* $\in$ *bad*; *ASID hf1* $\in$ *bad*$\rrbracket$
$\Longrightarrow$ (*info*, *hf1*#*hf2*#*ys*) $\in$ *auth-seg0*
by(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def TW.holds-split-tail attr-segment-def*)

lemma *ASM-modify*: $\llbracket$(*info*, *hf*#*ys*) $\in$ *auth-seg0*; *ASID hf* = *a*;
ASID hf' = *a*; *UpIF hf'* = *UpIF hf*; *a* $\in$ *bad*$\rrbracket$ $\Longrightarrow$ (*info*, *hf'*#*ys*) $\in$ *auth-seg0*
apply(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def attr-segment-def*)
subgoal for *y hfa l*
apply(*induction l*)
by *auto*
subgoal for *y hfa l*
apply(*induction l*)
by *auto*
done

lemma *rightpath-no-nF*: $\llbracket$*ASID hf* = *nF*; *zs* @ *hf* # *ys* = *rightpath*$\rrbracket$ $\Longrightarrow$ *False*
apply(*cases ys rule*: *rev-cases*, *auto*)
subgoal for *ys'* **apply**(*cases ys' rule*: *rev-cases*, *auto*)

subgoal for *ys''* **apply**(*cases ys'' rule*: *rev-cases*, *auto*)
subgoal for *ys'''* **apply**(*cases ys''' rule*: *rev-cases*, *auto*)
subgoal for *ys'''* **apply**(*cases ys''' rule*: *rev-cases*, *auto*)
done
done
done
done
done

lemma *ASM-cutoff-leftpath*:
$[\![ASID\ hf = nF;$
$\forall hfa.\ UpIF\ hfa = Some\ nE \longrightarrow ASID\ hfa = nF \longrightarrow (\forall xs.\ hf\ \#\ ys = xs\ @\ [hfa,\ hiEl,\ hiDr,\ hiBr,\ hiAr] \longrightarrow$
$\neg\ set\ xs \subseteq hfF\text{-}attr);\ x \in set\ ys;\ info = Num\ 0;$
$zs\ @\ hf\ \#\ ys = xs\ @\ [hfa,\ hiEl,\ hiDr,\ hiBr,\ hiAr];\ ASID\ hfa = nF;\ UpIF\ hfa = Some\ nE;$
$set\ xs \subseteq hfF\text{-}attr]\!]$
$\Longrightarrow ASID\ x = nF$
apply(*cases ys rule*: *rev-cases*) **apply** *simp*
subgoal for *ys' b*
apply(*cases ys' rule*: *rev-cases*) **apply** *simp*
subgoal for *ys'' c*
apply(*cases ys'' rule*: *rev-cases*) **apply** *simp*
subgoal for *ys''' d*
apply(*cases ys'' rule*: *rev-cases*) **apply** *simp*
subgoal for *ys''' e*
apply(*cases ys''' rule*: *rev-cases*) **apply** *simp*
subgoal for *ys'''' f*
apply(*cases ys'''' rule*: *rev-cases*) **apply** *simp*
apply *auto*
by *blast+*
done
done
done
done
done

lemma *ASM-cutoff*: $[\![(info,\ zs@hf\#ys) \in auth\text{-}seg0;\ ASID\ hf \in bad]\!] \Longrightarrow (info,\ hf\#ys) \in auth\text{-}seg0$
apply(*simp add*: *auth-seg0-def*, *auto dest*: *rightpath-no-nF*)
by(*auto simp add*: *leftsegment-wormholed-def TW.holds-split-tail attr-segment-def intro*: *ASM-cutoff-leftpath*)

definition *no-oracle* :: *msgterm* ⇒ *msgterm* ⇒ *bool* **where**
no-oracle ainfo uinfo = *True*

sublocale *e1*: *epic-l1-defs bad tgtas tgtif auth-seg0 no-oracle*
apply *unfold-locales*
using *ASM-if-valid ASM-rooted ASM-terminated ASM-empty ASM-singleton ASM-extension ASM-modify ASM-cutoff*
apply *simp-all*
done

sublocale *e1-int*: *epic-l1-defs bad tgtas tgtif auth-seg0 no-oracle*
using *e1.epic-l1-defs-axioms* **by** *auto*

3.5.3 Executability

Honest sender's packet forwarding

abbreviation *ainfo* **where** $ainfo \equiv Num\ 0$
abbreviation *uinfo* **where** $uinfo \equiv Num\ 1$
abbreviation σA **where** $\sigma A \equiv Mac[macKey\ nA]\ (L\ [ainfo,\ \varepsilon,\ AS\ nB])$
abbreviation σB **where** $\sigma B \equiv Mac[macKey\ nB]\ (L\ [ainfo,\ AS\ nA,\ AS\ nD,\ Hash\ \sigma A])$
abbreviation σD **where** $\sigma D \equiv Mac[macKey\ nD]\ (L\ [ainfo,\ AS\ nB,\ AS\ nE,\ Hash\ \sigma B])$
abbreviation σE **where** $\sigma E \equiv Mac[macKey\ nE]\ (L\ [ainfo,\ AS\ nD,\ AS\ nF,\ Hash\ \sigma D])$
abbreviation σF **where** $\sigma F \equiv Mac[macKey\ nF]\ (L\ [ainfo,\ AS\ nE,\ \varepsilon,\ Hash\ \sigma E])$

definition *hfAl* **where** $hfAl \equiv (\!|AHI = hiAl,\ UHI = Hash\ \sigma A,\ HVF = Mac[\sigma A]\ \langle ainfo,\ uinfo\rangle|\!)$
definition *hfBl* **where** $hfBl \equiv (\!|AHI = hiBl,\ UHI = Hash\ \sigma B,\ HVF = Mac[\sigma B]\ \langle ainfo,\ uinfo\rangle|\!)$
definition *hfDl* **where** $hfDl \equiv (\!|AHI = hiDl,\ UHI = Hash\ \sigma D,\ HVF = Mac[\sigma D]\ \langle ainfo,\ uinfo\rangle|\!)$
definition *hfEl* **where** $hfEl \equiv (\!|AHI = hiEl,\ UHI = Hash\ \sigma E,\ HVF = Mac[\sigma E]\ \langle ainfo,\ uinfo\rangle|\!)$
definition *hfFl* **where** $hfFl \equiv (\!|AHI = hiFl,\ UHI = Hash\ \sigma F,\ HVF = Mac[\sigma F]\ \langle ainfo,\ uinfo\rangle|\!)$

lemmas *hfl-defs = hfAl-def hfBl-def hfDl-def hfEl-def hfFl-def*

lemma *e1.hf-valid ainfo uinfo None hfAl None*
  by (*simp add*: *e1.hf-valid-invert hfAl-def*)
lemma *e1.hf-valid ainfo uinfo None hfBl* (*Some hfAl*)
  apply (*auto simp add*: *e1.hf-valid-invert hfAl-def hfBl-def*)
  using *d0.ASIF.simps* **by** *blast+*

lemma *e1.hf-valid ainfo uinfo None hfFl* (*Some hfEl*)
  apply (*auto intro*!: *exI simp add*: *e1.hf-valid-invert hfl-defs*)
  using *d0.ASIF.simps* **by** *blast+*

abbreviation *forwardingpath* **where**
  forwardingpath ≡ [*hfFl*, *hfEl*, *hfDl*, *hfBl*, *hfAl*]

definition *pkt0* **where** *pkt0* ≡ (|
    AInfo = *ainfo*,
    UInfo = *uinfo*,
    past = [],
    future = *forwardingpath*,
    history = []
  |)
definition *pkt1* **where** *pkt1* ≡ (|
    AInfo = *ainfo*,
    UInfo = *uinfo*,
    past = [*hfFl*],
    future = [*hfEl*, *hfDl*, *hfBl*, *hfAl*],
    history = [*hiFl*]
  |)
definition *pkt2* **where** *pkt2* ≡ (|
    AInfo = *ainfo*,
    UInfo = *uinfo*,
    past = [*hfEl*, *hfFl*],
    future = [*hfDl*, *hfBl*, *hfAl*],
    history = [*hiEl*, *hiFl*]

$\|$)
definition *pkt3* **where** *pkt3* ≡ (|
AInfo = *ainfo*,
UInfo = *uinfo*,
past = [*hfDl*, *hfEl*, *hfFl*],
future = [*hfBl*, *hfAl*],
history = [*hiDl*, *hiEl*, *hiFl*]
|)
definition *pkt4* **where** *pkt4* ≡ (|
AInfo = *ainfo*,
UInfo = *uinfo*,
past = [*hfBl*, *hfDl*, *hfEl*, *hfFl*],
future = [*hfAl*],
history = [*hiBl*, *hiDl*, *hiEl*, *hiFl*]
|)
definition *pkt5* **where** *pkt5* ≡ (|
AInfo = *ainfo*,
UInfo = *uinfo*,
past = [*hfAl*, *hfBl*, *hfDl*, *hfEl*, *hfFl*],
future = [],
history = [*hiAl*, *hiBl*, *hiDl*, *hiEl*, *hiFl*]
|)

definition *s0* **where** *s0* ≡ *e1.dp2-init*
definition *s1* **where** *s1* ≡ *s0*(|*loc2* := (*loc2 s0*)(*nF* := {*pkt0*})|)
definition *s2* **where**
s2 ≡ *s1*(|*chan2* := (*chan2 s1*)((*nF*, *nE*, *nE*, *nF*) := *chan2 s1* (*nF*, *nE*, *nE*, *nF*) ∪ {*pkt1*})|)
definition *s3* **where** *s3* ≡ *s2*(|*loc2* := (*loc2 s2*)(*nE* := {*pkt1*})|)
definition *s4* **where**
s4 ≡ *s3*(|*chan2* := (*chan2 s3*)((*nE*, *nD*, *nD*, *nE*) := *chan2 s3* (*nE*, *nD*, *nD*, *nE*) ∪ {*pkt2*})|)
definition *s5* **where** *s5* ≡ *s4*(|*loc2* := (*loc2 s4*)(*nD* := {*pkt2*})|)
definition *s6* **where**
s6 ≡ *s5*(|*chan2* := (*chan2 s5*)((*nD*, *nB*, *nB*, *nD*) := *chan2 s5* (*nD*, *nB*, *nB*, *nD*) ∪ {*pkt3*})|)
definition *s7* **where** *s7* ≡ *s6*(|*loc2* := (*loc2 s6*)(*nB* := {*pkt3*})|)
definition *s8* **where**
s8 ≡ *s7*(|*chan2* := (*chan2 s7*)((*nB*, *nA*, *nA*, *nB*) := *chan2 s7* (*nB*, *nA*, *nA*, *nB*) ∪ {*pkt4*})|)
definition *s9* **where** *s9* ≡ *s8*(|*loc2* := (*loc2 s8*)(*nA* := {*pkt4*})|)
definition *s10* **where** *s10* ≡ *s9*(|*loc2* := (*loc2 s9*)(*nA* := {*pkt4*, *pkt5*})|)

lemmas *forwading-states* =
s0-def s1-def s2-def s3-def s4-def s5-def s6-def s7-def s8-def s9-def s10-def

lemma *forwardingpath-valid*: *e1.hfs-valid-None ainfo uinfo forwardingpath*
by(*auto simp add*: *TW.holds-split-tail hfl-defs*)

lemma *forwardingpath-auth*: *pfragment ainfo forwardingpath e1.auth-seg2*
apply(*auto simp add*: *e1.auth-seg2-def pfragment-def*)
apply(*auto intro*!: *exI*[*of* - []])
apply(*rule exI*[*of* - *uinfo*])
apply(*simp only*: *forwardingpath-valid*)
by(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def hfl-defs*)

lemma *reach-s0*: *reach e1.dp2 s0* **by**(*auto simp add*: *s0-def e1.dp2-def*)

lemma *s0-s1*: *e1.dp2*: *s0* $-$*evt-dispatch-int2 nF pkt0*$\rightarrow$ *s1*
  using *forwardingpath-auth*
  apply(*auto dest*!: *e1.dp2-dispatch-int-also-works-for-honest*)
  by (*auto simp add*: *e1.dp2-def forwading-states e1.dp2-defs e1.dp2-msgs pkt0-def*)
    (*auto simp add*: *hfl-defs no-oracle-def*)

lemma *s1-s2*: *e1.dp2*: *s1* $-$*evt-send2 nF nE pkt0*$\rightarrow$ *s2*
  by (*auto simp add*: *e1.dp2-def forwading-states e1.dp2-defs e1.dp2-msgs pkt0-def pkt1-def*)
    (*auto simp add*: *hfl-defs*)

lemma *s2-s3*: *e1.dp2*: *s2* $-$*evt-recv2 nE nF pkt1*$\rightarrow$ *s3*
  by (*auto simp add*: *e1.dp2-def forwading-states e1.dp2-defs e1.dp2-msgs pkt0-def pkt1-def*)
    (*auto simp add*: *hfl-defs*)

lemma *s3-s4*: *e1.dp2*: *s3* $-$*evt-send2 nE nD pkt1*$\rightarrow$ *s4*
  by (*auto simp add*: *e1.dp2-def forwading-states e1.dp2-defs e1.dp2-msgs pkt1-def pkt2-def*)
    (*auto simp add*: *hfl-defs*)

lemma *s4-s5*: *e1.dp2*: *s4* $-$*evt-recv2 nD nE pkt2*$\rightarrow$ *s5*
  by (*auto simp add*: *e1.dp2-def forwading-states e1.dp2-defs e1.dp2-msgs pkt1-def pkt2-def*)
    (*auto simp add*: *hfl-defs*)

lemma *s5-s6*: *e1.dp2*: *s5* $-$*evt-send2 nD nB pkt2*$\rightarrow$ *s6*
  by (*auto simp add*: *e1.dp2-def forwading-states e1.dp2-defs e1.dp2-msgs pkt3-def pkt2-def*)
    (*auto simp add*: *hfl-defs*)

lemma *s6-s7*: *e1.dp2*: *s6* $-$*evt-recv2 nB nD pkt3*$\rightarrow$ *s7*
  by (*auto simp add*: *e1.dp2-def forwading-states e1.dp2-defs e1.dp2-msgs pkt3-def pkt2-def*)
    (*auto simp add*: *hfl-defs*)

lemma *s7-s8*: *e1.dp2*: *s7* $-$*evt-send2 nB nA pkt3*$\rightarrow$ *s8*
  by (*auto simp add*: *e1.dp2-def forwading-states e1.dp2-defs e1.dp2-msgs pkt4-def pkt3-def*)
    (*auto simp add*: *hfl-defs*)

lemma *s8-s9*: *e1.dp2*: *s8* $-$*evt-recv2 nA nB pkt4*$\rightarrow$ *s9*
  by (*auto simp add*: *e1.dp2-def forwading-states e1.dp2-defs e1.dp2-msgs pkt4-def pkt3-def*)
    (*auto simp add*: *hfl-defs*)

lemma *s9-s10*: *e1.dp2*: *s9* $-$*evt-deliver2 nA pkt4*$\rightarrow$ *s10*
  by (*auto simp add*: *e1.dp2-def forwading-states e1.dp2-defs e1.dp2-msgs pkt5-def pkt4-def*)
    (*auto simp add*: *hfl-defs*)

The state in which the packet is received is reachable

lemma *executability*: *reach e1.dp2 s10*
  using *reach-s0 s0-s1 s1-s2 s2-s3 s3-s4 s4-s5 s5-s6 s6-s7 s7-s8 s8-s9 s9-s10*
  by(*auto elim*!: *reach-trans*)

Attacker event executability

We also show that the attacker event can be executed.

definition *pkt-attr* **where** *pkt-attr* ≡ (|
AInfo = *ainfo*,
UInfo = *uinfo*,
past = [],
future = [*hfEl*],
history = []
|)

definition *s-attr* **where**
s-attr ≡ *s0*(|*chan2* := (*chan2 s0*)((*nF*, *nE*, *nE*, *nF*) := *chan2 s0* (*nF*, *nE*, *nE*, *nF*) ∪ {*pkt-attr*})|)

lemma *ik-auth-hfs-in-ik*: *t* ∈ *e1.ik-auth-hfs* ⟹ *t* ∈ *synth* (*analz* (*e1.ik-dyn s*))
by(*auto simp add*: *e1.ik-dyn-def e1.ik-def*)

lemma *hvf-e-auth*: *HVF hfEl* ∈ *e1.ik-auth-hfs*
apply(*auto simp add*: *e1.ik-auth-hfs-def*
intro!: *exI*[*of* - *hfEl*] *exI*[*of* - [*hfFl*, *hfEl*, *hfDl*, *hfBl*, *hfAl*]] *exI*[*of* - *ainfo*])
apply(*auto simp add*: *e1.auth-seg2-def*)
using *forwardingpath-valid* **by**(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def hfl-defs*)

lemma *uhi-e-auth*: *UHI hfEl* ∈ *e1.ik-auth-hfs*
apply(*auto simp add*: *e1.ik-auth-hfs-def*
intro!: *exI*[*of* - *hfEl*] *exI*[*of* - [*hfFl*, *hfEl*, *hfDl*, *hfBl*, *hfAl*]] *exI*[*of* - *ainfo*])
apply(*auto simp add*: *e1.auth-seg2-def*)
using *forwardingpath-valid* **by**(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def hfl-defs*)

The attacker can also execute her event.

lemma *attr-executability*: *reach e1.dp2 s-attr*
proof−
have *e1.dp2*: *s0* −*evt-dispatch-ext2 nF nE pkt-attr*→ *s-attr*
apply (*auto simp add*: *forwading-states e1.dp2-defs e1.dp2-msgs pkt-attr-def*)
using *ik-auth-hfs-in-ik hvf-e-auth uhi-e-auth* **apply** *blast*+
by(*auto simp add*: *no-oracle-def s-attr-def s0-def e1.dp2-init-def pkt-attr-def*)
then show *?thesis* **using** *reach-s0* **by** *auto*
qed

end
end

3.6 EPIC Level 2 in the Strong Attacker Model

theory *EPIC-L2-SA*
  imports
    ../Parametrized-Dataplane-3-directed
    ../infrastructure/Keys
begin

locale *epic-l2-defs* = *network-assums-direct - - - auth-seg0*
  for *auth-seg0* :: (*msgterm* × *ahi list*) *set* +
  fixes *no-oracle* :: *msgterm* ⇒ *msgterm* ⇒ *bool*
begin

3.6.1 Hop validation check and extract functions

We model the host key, i.e., the DRKey shared between an AS and an end host as a pair of AS identifier and source identifier. Note that this "key" is not necessarily secret. Because the source identifier is not directly embedded, we extract it from the uinfo field. The uinfo (i.e., the token) is derived from the source address. We thus assume that there is some function that extracts the source identifier from the uinfo field.

definition *source-extract* :: *msgterm* ⇒ *msgterm* **where** *source-extract* = *undefined*

definition *K-i* :: *as* ⇒ *msgterm* ⇒ *msgterm* **where**
  K-i asid uinfo = ⟨*AS asid*, *source-extract uinfo*⟩

The predicate *hf-valid* is given to the concrete parametrized model as a parameter. It ensures the authenticity of the hop authenticator in the hop field. The predicate takes an authenticated info field (in this model always a numeric value, hence the matching on Num ts), an unauthenticated info field uinfo, the hop field to be validated and in some cases the next hop field.

We distinguish if there is a next hop field (this yields the two cases below). If there is not, then the hop authenticator σ simply consists of a MAC over the authenticated info field and the local routing information of the hop, using the key of the hop to which the hop field belongs. If on the other hand, there is a subsequent hop field, then the uhi field of that hop field is also included in the MAC computation.

The hop authenticator σ is used to compute both the hop validation field and the uhi field. The first is computed as a MAC over the path origin (pair of absolute timestamp ts and the relative timestamp given in uinfo), using the hop authenticator as a key to the MAC. The hop authenticator is not secret, and any end host can use it to create a valid hvf. The uhi field, according to the protocol description, is σ shortened to a few bytes. We model this as applying the hash on σ.

The predicate *hf-valid* checks if the hop authenticator, hvf and uhi field are computed correctly.

fun *hf-valid* :: *msgterm* ⇒ *msgterm*
    ⇒ (*unit*, *msgterm*) *HF option*
    ⇒ (*unit*, *msgterm*) *HF*
    ⇒ (*unit*, *msgterm*) *HF option* ⇒ *bool* **where**
  hf-valid (*Num ts*) *uinfo* - (|*AHI* = *ahi*, *UHI* = *uhi*, *HVF* = *x*|) (*Some* (|*AHI* = *ahi2*, *UHI* = *uhi2*, *HVF* = *x2*|)) ⟷

$(\exists\, \sigma\ upif\ downif.\ \sigma = Mac[macKey\ (ASID\ ahi)]\ (L\ [Num\ ts,\ upif,\ downif,\ uhi2]) \wedge$
$ASIF\ (DownIF\ ahi)\ downif \wedge ASIF\ (UpIF\ ahi)\ upif \wedge uhi = Hash\ \sigma \wedge$
$x = Mac[K\text{-}i\ (ASID\ ahi)\ uinfo]\ \langle Num\ ts,\ uinfo,\ \sigma\rangle)$
$|\ hf\text{-}valid\ (Num\ ts)\ uinfo\ \text{-}\ (\!|AHI = ahi,\ UHI = uhi,\ HVF = x|\!)\ None \longleftrightarrow$
$(\exists\, \sigma\ upif\ downif.\ \sigma = Mac[macKey\ (ASID\ ahi)]\ (L\ [Num\ ts,\ upif,\ downif]) \wedge$
$ASIF\ (DownIF\ ahi)\ downif \wedge ASIF\ (UpIF\ ahi)\ upif \wedge uhi = Hash\ \sigma \wedge$
$x = Mac[K\text{-}i\ (ASID\ ahi)\ uinfo]\ \langle Num\ ts,\ uinfo,\ \sigma\rangle)$
$|\ hf\text{-}valid\ \text{-}\ \text{-}\ \text{-}\ \text{-}\ \text{-} = False$

We can extract the entire path from the uhi field, since it includes the hop authenticator, which includes the local forwarding information as well as, recursively, all upstream hop authenticators and their hop information. However, the parametrized model defines the extract function to operate on the hop validation field, not the uhi field. We therefore define a separate function that extracts the path from a hvf. We can do so, as both hvf and uhi contain the hop authenticator. Internally, that function uses *extrUhi*.

fun $extrUhi :: msgterm \Rightarrow ahi\ list$ **where**
$extrUhi\ (Hash\ (Mac[macKey\ asid]\ (L\ [ts,\ upif,\ downif,\ uhi2])))$
$= (\!|UpIF = ASO\ upif,\ DownIF = ASO\ downif,\ ASID = asid|\!)\ \#\ extrUhi\ uhi2$
$|\ extrUhi\ (Hash\ (Mac[macKey\ asid]\ (L\ [ts,\ upif,\ downif])))$
$= [(\!|UpIF = ASO\ upif,\ DownIF = ASO\ downif,\ ASID = asid|\!)]$
$|\ extrUhi\ \text{-} = []$

This function extracts from a hop validation field (HVF hf) the entire path.

fun $extr :: msgterm \Rightarrow ahi\ list$ **where**
$extr\ (Mac[\text{-}]\ \langle \text{-},\ \text{-},\ \sigma\rangle) = extrUhi\ (Hash\ \sigma)$
$|\ extr\ \text{-} = []$

Extract the authenticated info field from a hop validation field.

fun $extr\text{-}ainfo :: msgterm \Rightarrow msgterm$ **where**
$extr\text{-}ainfo\ (Mac[\text{-}]\ \langle Num\ ts,\ \text{-},\ \text{-}\rangle) = Num\ ts$
$|\ extr\text{-}ainfo\ \text{-} = \varepsilon$

abbreviation $ik\text{-}auth\text{-}ainfo :: msgterm \Rightarrow msgterm$ **where**
$ik\text{-}auth\text{-}ainfo \equiv id$

An authenticated info field is always a number (corresponding to a timestamp).

abbreviation *checkInfo* **where**
$checkInfo\ ainfo \equiv (\exists\, ts.\ ainfo = Num\ ts)$

When observing a hop field, an attacker learns the HVF and the UHI. The AHI only contains public information that are not terms.

fun $ik\text{-}hf :: (unit,\ msgterm)\ HF \Rightarrow msgterm\ set$ **where**
$ik\text{-}hf\ hf = \{HVF\ hf,\ UHI\ hf\}$

We now define useful properties of the above definition.

lemma *hf-valid-invert*:
$hf\text{-}valid\ tsn\ uinfo\ prev\ hf\ mo \longleftrightarrow$
$((\exists\, ahi\ ahi2\ \sigma\ ts\ upif\ downif\ asid\ x\ upif2\ downif2\ asid2\ uhi\ uhi2\ x2.$
$hf = (\!|AHI = ahi,\ UHI = uhi,\ HVF = x|\!) \wedge$
$ASID\ ahi = asid \wedge ASIF\ (DownIF\ ahi)\ downif \wedge ASIF\ (UpIF\ ahi)\ upif \wedge$

$mo = Some\ (\!|AHI = ahi2,\ UHI = uhi2,\ HVF = x2|\!) \wedge$
$ASID\ ahi2 = asid2 \wedge ASIF\ (DownIF\ ahi2)\ downif2 \wedge ASIF\ (UpIF\ ahi2)\ upif2 \wedge$
$\sigma = Mac[macKey\ asid]\ (L\ [tsn,\ upif,\ downif,\ uhi2]) \wedge$
$tsn = Num\ ts \wedge$
$uhi = Hash\ \sigma \wedge$
$x = Mac[K\text{-}i\ (ASID\ ahi)\ uinfo]\ \langle tsn,\ uinfo,\ \sigma\rangle)$
$\vee\ (\exists\, ahi\ \sigma\ ts\ upif\ downif\ asid\ uhi\ x.$
$hf = (\!|AHI = ahi,\ UHI = uhi,\ HVF = x|\!) \wedge$
$ASID\ ahi = asid \wedge ASIF\ (DownIF\ ahi)\ downif \wedge ASIF\ (UpIF\ ahi)\ upif \wedge$
$mo = None \wedge$
$\sigma = Mac[macKey\ asid]\ (L\ [tsn,\ upif,\ downif]) \wedge$
$tsn = Num\ ts \wedge$
$uhi = Hash\ \sigma \wedge$
$x = Mac[K\text{-}i\ (ASID\ ahi)\ uinfo]\ \langle tsn,\ uinfo,\ \sigma\rangle)$
$)$
apply(*auto elim*!: *hf-valid.elims*) **using** *option.exhaust ASIF.simps* **by** *metis+*

lemma *hf-valid-checkInfo*[*dest*]: *hf-valid ainfo uinfo prev hf z* $\Longrightarrow$ *checkInfo ainfo*
by(*auto simp add*: *hf-valid-invert*)

lemma *info-hvf*:
assumes *hf-valid ainfo uinfo prev m z HVF m* $=$ *Mac*[*k-i*] $\langle ainfo', uinfo', \sigma\rangle \vee$ *hf-valid ainfo' uinfo' prev' m z'*
shows *uinfo* $=$ *uinfo' ainfo'* $=$ *ainfo*
using *assms* **by**(*auto simp add*: *hf-valid-invert*)

sublocale *dataplane-3-directed-defs - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf*
by *unfold-locales*

3.6.2 Definitions and properties of the added intruder knowledge

Here we define two sets which are added to the intruder knowledge: *ik-add*, which contains hop authenticators. And *ik-oracle*, which contains the oracle's output to the strong attacker.

abbreviation *is-oracle* **where** *is-oracle ainfo t* $\equiv \neg$ *no-oracle ainfo t*

declare *TW.holds-set-list*[*dest*]
declare *TW.holds-takeW-is-identity*[*simp*]
declare *parts-singleton*[*dest*]

This additional Intruder Knowledge allows us to model the attacker's access not only to the hop validation fields and segment identifiers of authorized segments (which are already given in *ik-auth-hfs*), but to the underlying hop authenticators that are used to create them.

definition *ik-add* :: *msgterm set* **where**
ik-add $\equiv \{\ \sigma \mid ainfo\ uinfo\ l\ hf\ \sigma\ k\text{-}i.$
$(ainfo,\ l) \in auth\text{-}seg2$
$\wedge\ hf \in set\ l \wedge HVF\ hf = Mac[k\text{-}i]\ \langle ainfo,\ uinfo,\ \sigma\rangle\ \}$

lemma *ik-addI*:
$[\![(ainfo,\ l) \in auth\text{-}seg2;\ hf \in set\ l;\ HVF\ hf = Mac[k\text{-}i]\ \langle ainfo,\ uinfo,\ \sigma\rangle]\!] \Longrightarrow \sigma \in ik\text{-}add$
apply(*auto simp add*: *ik-add-def*)
by *blast*

lemma *ik-add-form*: $t \in$ *ik-add* $\Longrightarrow \exists$ *asid l* . $t =$ *Mac*[*macKey asid*] *l*

by(*auto simp add*: *ik-add-def auth-seg2-def dest*!: *TW.holds-set-list*)
(*auto simp add*: *hf-valid-invert*)

lemma *parts-ik-add*[*simp*]: *parts ik-add* = *ik-add*
by (*auto intro*!: *parts-Hash dest*: *ik-add-form*)

This is the oracle output provided to the adversary. Only those hop validation fields and segment identifiers whose path origin (combination of ainfo uinfo) is not contained in *no-oracle* appears here.

definition *ik-oracle* :: *msgterm set* **where**
ik-oracle = $\{t \mid$ *t ainfo hf l uinfo* . *hf* $\in$ *set l* $\wedge$ *hfs-valid-None ainfo uinfo l* $\wedge$
is-oracle ainfo uinfo $\wedge$ (*ainfo*, *l*) $\notin$ *auth-seg2* $\wedge$ (t = *HVF hf* $\vee$ t = *UHI hf*) $\}$

lemma *ik-oracle-parts-form*:
$t \in$ *ik-oracle* $\Longrightarrow$
($\exists$ *asid l ainfo uinfo k-i* . t = *Mac*[*k-i*] $\langle$*ainfo*, *uinfo*, *Mac*[*macKey asid*] *l*$\rangle$) $\vee$
($\exists$ *asid l* . t = *Hash* (*Mac*[*macKey asid*] *l*))
by(*auto simp add*: *ik-oracle-def hf-valid-invert dest*!: *TW.holds-set-list*)

lemma *parts-ik-oracle*[*simp*]: *parts ik-oracle* = *ik-oracle*
by (*auto intro*!: *parts-Hash dest*: *ik-oracle-parts-form*)

lemma *ik-oracle-simp*: $t \in$ *ik-oracle* $\longleftrightarrow$
($\exists$ *ainfo hf l uinfo*. *hf* $\in$ *set l* $\wedge$ *hfs-valid-None ainfo uinfo l* $\wedge$ *is-oracle ainfo uinfo*
$\wedge$ (*ainfo*, *l*) $\notin$ *auth-seg2* $\wedge$ (t = *HVF hf* $\vee$ t = *UHI hf*))
by(*rule iffI*, *frule ik-oracle-parts-form*)
(*auto simp add*: *ik-oracle-def hf-valid-invert*)

3.6.3 Properties of the intruder knowledge, including *ik-add* and *ik-oracle*

We now instantiate the parametrized model's definition of the intruder knowledge, using the definitions of *ik-add* and *ik-oracle* from above. We then prove the properties that we need to instantiate the *dataplane-3-directed* locale.

sublocale
dataplane-3-directed-ik-defs - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo
ik-hf ik-add ik-oracle no-oracle
by *unfold-locales*

lemma *ik-auth-hfs-form*: $t \in$ *parts ik-auth-hfs* $\Longrightarrow \exists\ t'$. t = *Hash* t'
apply *auto* **apply**(*drule parts-singleton*)
by(*auto simp add*: *auth-seg2-def hf-valid-invert dest*!: *TW.holds-set-list*)

declare *ik-auth-hfs-def*[*simp del*]

lemma *parts-ik-auth-hfs*[*simp*]: *parts ik-auth-hfs* = *ik-auth-hfs*
by (*auto intro*!: *parts-Hash ik-auth-hfs-form*)

This lemma allows us not only to expand the definition of *ik-auth-hfs*, but also to obtain useful properties, such as a term being a Hash, and it being part of a valid hop field.

lemma *ik-auth-hfs-simp*:
  $t \in$ *ik-auth-hfs* $\longleftrightarrow (\exists\, t'\, .\ t = Hash\ t') \wedge (\exists\, hf\ .\ (t = HVF\ hf \vee t = UHI\ hf)$
  $\wedge (\exists\, hfs.\ hf \in set\ hfs \wedge (\exists\, ainfo\ .\ (ainfo,\ hfs) \in auth\text{-}seg2$
  $\wedge (\exists\ prev\ nxt\ uinfo.\ hf\text{-}valid\ ainfo\ uinfo\ prev\ hf\ nxt))))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
  assume *asm*: *?lhs*
  then obtain *ainfo hf hfs* **where**
    dfs: $hf \in set\ hfs\ (ainfo,\ hfs) \in auth\text{-}seg2\ t = HVF\ hf \vee t = UHI\ hf$
    by(*auto simp add*: *ik-auth-hfs-def*)
  then obtain *uinfo* **where** *hfs-valid-None ainfo uinfo hfs* $(ainfo,\ AHIS\ hfs) \in auth\text{-}seg0$
    by(*auto simp add*: *auth-seg2-def*)
  then show *?rhs* **using** *asm dfs* **by**(*fast intro*: *ik-auth-hfs-form*)
qed(*auto simp add*: *ik-auth-hfs-def*)

Properties of Intruder Knowledge

lemma *auth-ainfo*[*dest*]: $[\![(ainfo,\ hfs) \in auth\text{-}seg2]\!] \Longrightarrow \exists\ ts\ .\ ainfo = Num\ ts$
  by(*auto simp add*: *auth-seg2-def*)

lemma *Num-ik*[*intro*]: $Num\ ts \in ik$
  by(*auto simp add*: *ik-def*)
    (*auto simp add*: *auth-seg2-def TW.holds.simps*(*3*) *elim*: *allE*[*of* - []])

There are no ciphertexts (or signatures) in *parts ik*. Thus, *analz ik* and *parts ik* are identical.

lemma *analz-parts-ik*[*simp*]: *analz ik* = *parts ik*
  apply(*rule no-crypt-analz-is-parts*)
  by(*auto simp add*: *ik-def auth-seg2-def ik-auth-hfs-simp*)
    (*auto simp add*: *ik-add-def ik-oracle-def auth-seg2-def hf-valid-invert hfs-valid-prefix-generic-def*
      dest!: *TW.holds-set-list*)

lemma *parts-ik*[*simp*]: *parts ik* = *ik*
  by(*auto 3 4 simp add*: *ik-def auth-seg2-def*)

lemma *key-ik-bad*: $Key\ (macK\ asid) \in ik \Longrightarrow asid \in bad$
  by(*auto simp add*: *ik-def hf-valid-invert ik-oracle-simp*)
    (*auto 3 4 simp add*: *auth-seg2-def ik-auth-hfs-simp ik-add-def hf-valid-invert*)

Updating hop fields with different uinfo

fun *K-i-upd* :: $msgterm \Rightarrow msgterm \Rightarrow msgterm$ **where**
  $K\text{-}i\text{-}upd\ \langle AS\ asid,\ \text{-}\rangle\ uinfo' = \langle AS\ asid,\ source\text{-}extract\ uinfo'\rangle$
  $|\ K\text{-}i\text{-}upd\ \text{-}\ \text{-} = \varepsilon$

Those hop validation fields contained in *auth-seg2* or that can be generated from the hop authenticators in *ik-add* have the property that they are agnostic about the uinfo field. If a hop validation field is contained in *auth-seg2* (resp. derivable from *ik-add*), then a field with a different uinfo is also contained (resp. derivable). To show this, we first define a function that updates uinfo in a hop validation field.

fun *uinfo-upd-hf* :: $msgterm \Rightarrow (unit,\ msgterm)\ HF \Rightarrow (unit,\ msgterm)\ HF$ **where**
  uinfo-upd-hf new-uinfo hf =
    (*case HVF hf of* $Mac[k\text{-}i]\ \langle ainfo,\ uinfo,\ \sigma\rangle$
  $\Rightarrow hf(\!|HVF := Mac[K\text{-}i\text{-}upd\ k\text{-}i\ new\text{-}uinfo]\ \langle ainfo,\ new\text{-}uinfo,\ \sigma\rangle|\!)\ |\ \text{-} \Rightarrow hf)$

fun *uinfo-upd* :: *msgterm* ⇒ (*unit*, *msgterm*) *HF list* ⇒ (*unit*, *msgterm*) *HF list* **where**
uinfo-upd new-uinfo hfs = *map* (*uinfo-upd-hf new-uinfo*) *hfs*

lemma *uinfo-upd-valid*:
hfs-valid ainfo uinfo pre l nxt ⟹ *hfs-valid ainfo new-uinfo pre* (*uinfo-upd new-uinfo l*) *nxt*
apply(*induction pre l nxt rule*: *TW.holds.induct*)
apply *auto*
subgoal for *prev x y ys z*
by(*cases map* (*uinfo-upd-hf new-uinfo*) *ys*)
(*auto simp add*: *TW.holds-split-tail hf-valid-invert K-i-def*)
by(*auto 3 4 simp add*: *TW.holds-split-tail hf-valid-invert TW.holds.simps*(*3*) *K-i-def*)

lemma *uinfo-upd-hf-AHI*: *AHI* (*uinfo-upd-hf new-uinfo hf*) = *AHI hf*
apply(*cases HVF hf*) **apply** *auto*
subgoal for *k-i* **apply**(*cases k-i*) **apply** *auto*
subgoal for *as uinfo* **apply**(*cases uinfo*) **apply** *auto*
subgoal for *x1 x2* **apply**(*cases x2*) **by** *auto*
done
done
done

lemma *uinfo-upd-hf-AHIS*[*simp*]: *AHIS* (*map* (*uinfo-upd-hf new-uinfo*) *l*) = *AHIS l*
apply(*induction l*) **using** *uinfo-upd-hf-AHI* **by** *auto*

lemma *uinfo-upd-auth-seg2*:
assumes *hf-valid ainfo uinfo prev m z σ* = *Mac*[*Key* (*macK asid*)] *j*
HVF m = *Mac*[*k-i*] ⟨*ainfo*, *uinfo′*, *σ*⟩ *σ* ∈ *ik-add*
shows ∃ *hfs*. *m* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
proof−
from *assms*(*4*) **obtain** *ainfo-add uinfo-add l-add hf-add k-i-add* **where**
(*ainfo-add*, *l-add*) ∈ *auth-seg2 hf-add* ∈ *set l-add*
HVF hf-add = *Mac*[*k-i-add*] ⟨*ainfo-add*, *uinfo-add*, *σ*⟩
by(*auto simp add*: *ik-add-def*)
then have *add*: *m* ∈ *set* (*uinfo-upd uinfo l-add*) (*ainfo-add*, (*uinfo-upd uinfo l-add*)) ∈ *auth-seg2*
using *assms*(*1*−*3*) **apply**(*auto simp add*: *auth-seg2-def simp del*: *AHIS-def*)
apply(*auto simp add*: *hf-valid-invert intro*!: *image-eqI dest*!: *TW.holds-set-list*)[*1*]
by(*auto intro*!: *exI elim*: *ahi-eq dest*: *uinfo-upd-valid simp del*: *AHIS-def simp add*: *K-i-def*)
then have *ainfo-add* = *ainfo*
using *assms*(*1*) **by**(*auto simp add*: *auth-seg2-def dest*!: *TW.holds-set-list dest*: *info-hvf*)
then show *?thesis* **using** *add* **by** *fastforce*
qed

lemma *MAC-synth-oracle*:
assumes *hf-valid ainfo uinfo prev m z HVF m* ∈ *ik-oracle*
shows *is-oracle ainfo uinfo*
using *assms* **by**(*auto simp add*: *ik-oracle-def assms*(*1*) *hf-valid-invert dest*!: *TW.holds-set-list*)

lemma *ik-oracle-is-oracle*:
⟦*Mac*[*k-i*] ⟨*ainfo*, *uinfo*, *σ*⟩ ∈ *ik-oracle*⟧ ⟹ *is-oracle ainfo uinfo*
by (*auto simp add*: *ik-oracle-def dest*: *info-hvf*)
(*auto dest*!: *TW.holds-set-list simp add*: *hf-valid-invert*)

lemma *MAC-synth-helper*:
$[\![$*hf-valid ainfo uinfo prev m z*; *no-oracle ainfo uinfo*;
HVF m = *Mac*[*k-i*] ⟨*ainfo*, *uinfo*, σ⟩; σ = *Mac*[*Key* (*macK asid*)] *j*; σ ∈ *ik* ∨ *HVF m* ∈ *ik*$]\!]$
⟹ ∃ *hfs*. *m* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
apply(*auto simp add*: *ik-def ik-auth-hfs-simp*
dest: *MAC-synth-oracle ik-add-form ik-oracle-parts-form*[*simplified*])
subgoal by(*auto simp add*: *hf-valid-invert simp add*: *K-i-def*)
subgoal by(*auto simp add*: *hf-valid-invert simp add*: *K-i-def*)
subgoal by(*auto elim*!: *uinfo-upd-auth-seg2 simp add*: *K-i-def*)
subgoal apply(*auto simp add*: *hf-valid-invert simp add*: *K-i-def*)
using *ik-oracle-parts-form* **by** *blast*+
subgoal apply(*auto simp add*: *hf-valid-invert simp add*: *K-i-def*)
using *ahi-eq* **by** *blast*+
subgoal by(*auto simp add*: *hf-valid-invert simp add*: *K-i-def*)
subgoal apply(*auto simp add*: *hf-valid-invert simp add*: *K-i-def*)
using *ik-add-form* **by** *blast*+
done

This definition helps with the limiting the number of cases generated. We don't require it, but it is convenient. Given a hop validation field and an asid, return if the hvf has the expected format.

definition *mac-format* :: *msgterm* ⇒ *as* ⇒ *bool* **where**
mac-format m asid ≡ ∃ *j ts uinfo k-i* . *m* = *Mac*[*k-i*] ⟨*Num ts*, *uinfo*, *Mac*[*macKey asid*] *j*⟩

If a valid hop field is derivable by the attacker, but does not belong to the attacker, and is over a path origin that does not belong to an oracle query, then the hop field is already contained in the set of authorized segments.

lemma *MAC-synth*:
assumes *hf-valid ainfo uinfo prev m z HVF m* ∈ *synth ik mac-format* (*HVF m*) *asid*
asid ∉ *bad checkInfo ainfo no-oracle ainfo uinfo*
shows ∃ *hfs* . *m* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
using *assms*
apply(*auto simp add*: *mac-format-def elim*!: *MAC-synth-helper dest*!: *key-ik-bad*)
apply(*auto simp add*: *ik-def ik-auth-hfs-simp dest*: *ik-add-form dest*!: *ik-oracle-parts-form*)
using *assms*(*1*) **by**(*auto dest*: *info-hvf simp add*: *hf-valid-invert*)

3.6.4 Direct proof goals for interpretation of *dataplane-3-directed*

lemma *COND-honest-hf-analz*:
assumes *ASID* (*AHI hf*) ∉ *bad hf-valid ainfo uinfo prev hf nxt ik-hf hf* ⊆ *synth* (*analz ik*)
no-oracle ainfo uinfo
shows *ik-hf hf* ⊆ *analz ik*
proof−
let *?asid* = *ASID* (*AHI hf*)
from *assms*(*3*) **have** *hf-synth-ik*: *HVF hf* ∈ *synth ik UHI hf* ∈ *synth ik* **by** *auto*
from *assms*(*2*) **have** *mac-format* (*HVF hf*) *?asid checkInfo ainfo*
by(*auto simp add*: *mac-format-def hf-valid-invert*)
then obtain *hfs* **where** *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2*
using *assms*(*1*,*2*,*4*) *hf-synth-ik* **by**(*auto dest*!: *MAC-synth*)
then have *HVF hf* ∈ *ik UHI hf* ∈ *ik*
using *assms*(*2*)
by(*auto simp add*: *ik-auth-hfs-def intro*!: *ik-ik-auth-hfs intro*!: *exI*)

then show *?thesis* **by** *auto*
qed

lemma *COND-ainfo-analz*:
assumes *hf-valid ainfo uinfo prev hf nxt* **and** *ik-auth-ainfo ainfo* ∈ *synth* (*analz ik*)
shows *ik-auth-ainfo ainfo* ∈ *analz ik*
using *assms* **by**(*auto simp add*: *hf-valid-invert*)

lemma *COND-ik-hf*:
assumes *hf-valid ainfo uinfo prev hf z* **and** *HVF hf* ∈ *ik* **and** *no-oracle ainfo uinfo*
shows ∃ *hfs*. *hf* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
proof−
from *assms* **have** *checkInfo ainfo* **by** *auto*
then obtain *hfs ainfo* **where** *hfs-def*: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2*
using *assms* **by**(*auto 3 4 simp add*: *K-i-def hf-valid-invert ik-auth-hfs-simp ik-def dest*: *ahi-eq dest*!: *ik-oracle-is-oracle ik-add-form*)
then obtain *hfs ainfo* **where** *hfs-def*: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2* **by** *auto*
show *?thesis*
using *hfs-def* **apply** (*auto simp add*: *auth-seg2-def dest*!: *TW*.*holds-set-list*)
using *hfs-def assms*(*1*) **by** (*auto simp add*: *auth-seg2-def dest*: *info-hvf*)
qed

lemma *COND-extr-prefix-path*:
⟦*hfs-valid ainfo uinfo pre l nxt*; *nxt* = *None*⟧ ⟹ *prefix* (*extr-from-hd l*) (*AHIS l*)
by(*induction pre l nxt rule*: *TW*.*holds*.*induct*)
(*auto simp add*: *TW*.*holds-split-tail TW*.*holds*.*simps*(*1*) *hf-valid-invert*,
auto split: *list*.*split-asm simp add*: *hf-valid-invert K-i-def intro*!: *ahi-eq elim*: *ASIF*.*elims*)

lemma *COND-path-prefix-extr*:
prefix (*AHIS* (*hfs-valid-prefix ainfo uinfo pre l nxt*))
(*extr-from-hd l*)
apply(*induction pre l nxt rule*: *TW*.*takeW*.*induct*[**where** *?Pa*=*hf-valid ainfo uinfo*])
apply(*auto simp add*: *TW*.*takeW-split-tail TW*.*takeW*.*simps*(*1*))
apply(*auto simp add*: *hf-valid-invert K-i-def intro*!: *ahi-eq*)
by(*auto elim*: *ASIF*.*elims*)

lemma *COND-hf-valid-no-prev*:
hf-valid ainfo uinfo prev hf z ⟷ *hf-valid ainfo uinfo prev*′ *hf z*
by(*auto simp add*: *hf-valid-invert*)

lemma *COND-hf-valid-uinfo*:
⟦*hf-valid ainfo uinfo pre hf nxt*; *hf-valid ainfo*′ *uinfo*′ *pre*′ *hf nxt*′⟧ ⟹ *uinfo*′ = *uinfo*
by(*auto dest*: *info-hvf*)

3.6.5 Instantiation of *dataplane-3-directed* locale

sublocale
dataplane-3-directed - - - *auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf ik-add ik-oracle no-oracle*
apply *unfold-locales*
using *COND-ik-hf COND-honest-hf-analz COND-ainfo-analz COND-extr-prefix-path COND-path-prefix-extr COND-hf-valid-no-prev COND-hf-valid-uinfo* **by** *auto*

end
end

3.7 ICING

We abstract and simplify from the protocol ICING in several ways. First, we only consider Proofs of Consent (PoC), not Proofs of Provenance (PoP). Our framework does not support proving the path validation properties that PoPs provide, and it also currently does not support XOR, and dynamically changing hop fields. Thus, instead of embedding $A_i \oplus PoP_{0,1}$, we embed A_i directly. We also remove the payload from the Hash that is included in each packet.

We offer three versions of this protocol:

- *ICING*, which contains our best effort at modeling the protocol as accurately as possible.
- *ICING-variant*, in which we strip down the protocol to what is required to obtain the security guarantees and remove unnecessary fields.
- *ICING-variant2*, in which we furthermore simplify the protocol. The key of the MAC in this protocol is only the key of the AS, as opposed to a key derived specifically for this hop field. In order to prove that this scheme is secure, we have to assume that ASes only occur once on an authorized path, since otherwise the MAC for two different hop fields (by the same AS) would be the same, and the AS could not distinguish the hop fields based on the MAC.

theory *ICING*
  imports
    ../Parametrized-Dataplane-3-undirected
begin

locale *icing-defs* = *network-assums-undirect* - - - *auth-seg0*
  for *auth-seg0* :: (*msgterm* × *nat ahi-scheme list*) *set*
begin

3.7.1 Hop validation check and extract functions

type-synonym *ICING-HF* = (*nat*, *unit*) *HF*

The term *sntag* is a key that is derived from the key of an AS and a specific hop field. We use it in the computation of *hf-valid*. The "tag" field is a opaque numeric value which is used to encode further routing information of a node.

fun *sntag* :: *nat ahi-scheme* ⇒ *msgterm* **where**
  sntag (|*UpIF* = *upif*, *DownIF* = *downif*, *ASID* = *asid*, ... = *tag*|)
    = ⟨*macKey asid*, *if2term upif*, *if2term downif*, *Num tag*⟩

lemma *sntag-eq*: *sntag ahi2* = *sntag ahi1* ⟹ *ahi2* = *ahi1*
  by(*cases ahi1*,*cases ahi2*) *auto*

fun *hf2term* :: *nat ahi-scheme* ⇒ *msgterm* **where**
  hf2term (|*UpIF* = *upif*, *DownIF* = *downif*, *ASID* = *asid*, ... = *tag*|)
  = *L* [*if2term upif*, *if2term downif*, *Num asid*, *Num tag*]

fun *term2hf* :: *msgterm* ⇒ *nat ahi-scheme* **where**

term2hf (*L* [*upif*, *downif*, *Num asid*, *Num tag*])
= (| *UpIF* = *term2if upif*, *DownIF* = *term2if downif*, *ASID* = *asid*, ... = *tag* |)

lemma *term2hf-hf2term*[*simp*]: *term2hf* (*hf2term hf*) = *hf* **apply**(*cases hf*) **by** *auto*

We make some useful definitions that will be used to define the predicate *hf-valid*. Having them as separate definitions is useful to prevent unfolding in proofs that don't require it.

definition *fullpath* :: *ICING-HF list* ⇒ *msgterm* **where**
fullpath hfs = *L* (*map* (*hf2term o AHI*) *hfs*)

definition *maccontents* **where**
maccontents ahi hfs PoC-i-expire
= ⟨*Mac*[*sntag ahi*] ⟨*fullpath hfs*, *Num PoC-i-expire*⟩, ⟨*Num 0*, *Hash* (*fullpath hfs*)⟩⟩

The predicate *hf-valid* is given to the concrete parametrized model as a parameter. It ensures the authenticity of the hop authenticator in the hop field. The predicate takes an expiration timestamp (in this model always a numeric value, hence the matching on *Num PoC-i-expire*), the entire segment and the hop field to be validated.

fun *hf-valid* :: *msgterm* ⇒ *msgterm*
⇒ *ICING-HF list*
⇒ *ICING-HF*
⇒ *bool* **where**
hf-valid (*Num PoC-i-expire*) *uinfo hfs* (|*AHI* = *ahi*, *UHI* = *uhi*, *HVF* = *A-i*|) ⟷
uhi = () ∧ *uinfo* = ε ∧ *A-i* = *Hash* (*maccontents ahi hfs PoC-i-expire*)
| *hf-valid* - - - - = *False*

We can extract the entire path (past and future) from the hvf field.

fun *extr* :: *msgterm* ⇒ *nat ahi-scheme list* **where**
extr (*Mac*[*Mac*[-] ⟨*L fullpathhfs*, *Num PoC-i-expire*⟩] -)
= *map term2hf fullpathhfs*
| *extr* - = []

Extract the authenticated info field from a hop validation field.

fun *extr-ainfo* :: *msgterm* ⇒ *msgterm* **where**
extr-ainfo (*Mac*[-] (*L* (*Num ts* # *xs*))) = *Num ts*
| *extr-ainfo* - = ε

abbreviation *ik-auth-ainfo* :: *msgterm* ⇒ *msgterm* **where**
ik-auth-ainfo ≡ *id*

An authenticated info field is always a number (corresponding to a timestamp).

abbreviation *checkInfo* **where**
checkInfo ainfo ≡ (∃ *ts*. *ainfo* = *Num ts*)

When observing a hop field, an attacker learns the HVF. UHI is empty and the AHI only contains public information that are not terms.

fun *ik-hf* :: *ICING-HF* ⇒ *msgterm set* **where**
ik-hf hf = {*HVF hf*}

abbreviation *no-oracle* **where** *no-oracle* ≡ (λ - -. *True*)

We now define useful properties of the above definition.

lemma *hf-valid-invert*:
hf-valid tsn uinfo hfs hf $\longleftrightarrow$
$(\exists$ *PoC-i-expire ahi A-i* . *tsn* $=$ *Num PoC-i-expire* $\wedge$ *ahi* $=$ *AHI hf* $\wedge$
UHI hf $= () \wedge$ *uinfo* $= \varepsilon \wedge$
HVF hf $=$ *A-i* $\wedge$
A-i $=$ *Hash* (*maccontents ahi hfs PoC-i-expire*))
apply(*cases hf*) **by**(*auto elim*!: *hf-valid.elims*)

lemma *hf-valid-checkInfo*[*dest*]: *hf-valid ainfo uinfo hfs hf* $\Longrightarrow$ *checkInfo ainfo*
by(*auto simp add*: *hf-valid-invert*)

lemma *info-hvf*:
assumes *hf-valid ainfo uinfo hfs m hf-valid ainfo′ uinfo′ hfs′ m′*
HVF m $=$ *HVF m′ m* $\in$ *set hfs m′* $\in$ *set hfs′*
shows *ainfo′* $=$ *ainfo m′* $=$ *m*
using *assms*
apply(*auto simp add*: *hf-valid-invert maccontents-def intro*: *ahi-eq*)
apply(*cases m*,*cases m′*)
by(*auto intro*: *sntag-eq*)

3.7.2 Definitions and properties of the added intruder knowledge

Here we define a *ik-add* and *ik-oracle* as being empty, as these features are not used in this instance model.

sublocale *dataplane-3-undirected-defs - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf*
by *unfold-locales*

declare *parts-singleton*[*dest*]

definition *ik-add* :: *msgterm set* **where**
ik-add $\equiv$ { *PoC* | *ainfo l hf PoC pkthash*.
(*ainfo*, *l*) $\in$ *auth-seg2*
$\wedge$ *hf* $\in$ *set l* $\wedge$ *HVF hf* $=$ *Mac*[*PoC*] *pkthash* }

lemma *ik-addI*:
$[\![$(*ainfo*, *l*) $\in$ *local.auth-seg2*; *hf* $\in$ *set l*; *HVF hf* $=$ *Mac*[*PoC*] *pkthash*$]\!]$ $\Longrightarrow$ *PoC* $\in$ *ik-add*
by(*auto simp add*: *ik-add-def*)

lemma *ik-add-form*:
t $\in$ *ik-add* $\Longrightarrow \exists$ *asid upif downif tag l* . *t* $=$ *Mac*[⟨*macKey asid*, *if2term upif*, *if2term downif*, *Num tag*⟩] *l*
apply(*auto simp add*: *ik-add-def auth-seg2-def maccontents-def dest*!: *TW.holds-set-list*)
apply(*auto simp add*: *hf-valid-invert maccontents-def*)
by (*meson sntag.elims*)

lemma *parts-ik-add*[*simp*]: *parts ik-add* $=$ *ik-add*
by (*auto intro*!: *parts-Hash dest*: *ik-add-form*)

abbreviation *ik-oracle* :: *msgterm set* **where** *ik-oracle* $\equiv$ {}

3.7.3 Properties of the intruder knowledge, including *ik-add* and *ik-oracle*

We now instantiate the parametrized model's definition of the intruder knowledge, using the definitions of *ik-add* and *ik-oracle* from above. We then prove the properties that we need to instantiate the *dataplane-3-undirected* locale.

sublocale
  dataplane-3-undirected-ik-defs - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo
      ik-hf ik-add ik-oracle no-oracle
  by *unfold-locales*

lemma *ik-auth-hfs-form*: $t \in$ *parts ik-auth-hfs* $\Longrightarrow \exists\ t'\ .\ t =$ *Hash* t'
  apply *auto* **apply**(*drule parts-singleton*)
  by(*auto simp add*: *auth-seg2-def hf-valid-invert*)

declare *ik-auth-hfs-def*[*simp del*]

lemma *parts-ik-auth-hfs*[*simp*]: *parts ik-auth-hfs* = *ik-auth-hfs*
  by (*auto intro*!: *parts-Hash ik-auth-hfs-form*)

This lemma allows us not only to expand the definition of *ik-auth-hfs*, but also to obtain useful properties, such as a term being a Hash, and it being part of a valid hop field.

lemma *ik-auth-hfs-simp*:
  $t \in$ *ik-auth-hfs* $\longleftrightarrow$ ($\exists\, t'\ .\ t =$ *Hash* t') $\wedge$ ($\exists$ *hf* . t = *HVF hf*
      $\wedge$ ($\exists$ *hfs*. *hf* $\in$ *set hfs* $\wedge$ ($\exists$ *ainfo* . (*ainfo*, *hfs*) $\in$ *auth-seg2*
      $\wedge$ ($\exists$ *uinfo*. *hf-valid ainfo uinfo hfs hf*)))) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
  assume *asm*: *?lhs*
  then obtain *ainfo hf hfs* **where**
    dfs: *hf* $\in$ *set hfs* (*ainfo*, *hfs*) $\in$ *auth-seg2* t = *HVF hf*
    by(*auto simp add*: *ik-auth-hfs-def*)
  then obtain *uinfo* **where** *hfs-valid-prefix ainfo uinfo* [] *hfs* = *hfs* (*ainfo*, *AHIS hfs*) $\in$ *auth-seg0*
    by(*auto simp add*: *auth-seg2-def*)
  then show *?rhs* **using** *asm dfs* **by** *auto*(*fast intro*!: *ik-auth-hfs-form*)+
qed(*auto simp add*: *ik-auth-hfs-def*)

Properties of Intruder Knowledge

lemma *auth-ainfo*[*dest*]: $[\![$(*ainfo*, *hfs*) $\in$ *auth-seg2*$]\!] \Longrightarrow \exists$ *ts* . *ainfo* = *Num ts*
  by(*auto simp add*: *auth-seg2-def*)

lemma *Num-ik*[*intro*]: *Num ts* $\in$ *ik*
  by(*auto simp add*: *ik-def auth-seg2-def intro*!: *exI*[*of* - []])

There are no ciphertexts (or signatures) in *parts ik*. Thus, *analz ik* and *parts ik* are identical.

lemma *analz-parts-ik*[*simp*]: *analz ik* = *parts ik*
  apply(*rule no-crypt-analz-is-parts*)
  by(*auto simp add*: *ik-def auth-seg2-def ik-auth-hfs-simp dest*: *ik-add-form*)

lemma *parts-ik*[*simp*]: *parts ik* = *ik*
  by(*auto 3 4 simp add*: *ik-def auth-seg2-def dest*!: *parts-singleton-set*)

lemma *sntag-synth-bad*: *sntag ahi* $\in$ *synth ik* $\Longrightarrow$ *ASID ahi* $\in$ *bad*

apply(*cases ahi*)
by(*auto simp add*: *ik-def ik-auth-hfs-simp dest*: *ik-add-form*)

lemma *HF-eq*:
$[\![AHI\ hf' = AHI\ hf;\ UHI\ hf' = UHI\ hf;\ HVF\ hf' = HVF\ hf]\!] \Longrightarrow hf' = (hf::('x, 'y)HF)$
apply(*cases hf'*, *cases hf*)
by(*auto elim*: *HF.cases*)

3.7.4 Direct proof goals for interpretation of *dataplane-3-undirected*

lemma *ik-add-auth*: $[\![Mac[sntag\ (AHI\ hf)]\ \langle fullpath\ hfs,\ Num\ PoC\text{-}i\text{-}expire\rangle \in ik\text{-}add;$
$ASID\ (AHI\ hf) \notin bad;\ hf \in set\ hfs;\ uinfo = \varepsilon;$
$HVF\ hf = Mac[Mac[sntag\ (AHI\ hf)]\ \langle fullpath\ hfs,\ Num\ PoC\text{-}i\text{-}expire\rangle]\ \langle Num\ 0,\ Hash\ (fullpath\ hfs)\rangle]\!]$
$\Longrightarrow \exists hfs'.\ hf \in set\ hfs' \land (Num\ PoC\text{-}i\text{-}expire,\ hfs') \in auth\text{-}seg2$
apply(*auto simp add*: *ik-add-def*)
subgoal for *ainfo l hfa pkthash*
apply (*auto intro*!: *exI*[*of - l*])
subgoal
apply(*rule back-subst*[**where** *?a=hfa*, **where** *?b=hf*])
by(*auto intro*!: *HF-eq dest*!: *auth-seg2-elem simp add*: *hf-valid-invert maccontents-def sntag-eq*)
apply (*frule auth-seg2-elem*)
apply(*auto simp add*: *hf-valid-invert*)
by (*simp add*: *maccontents-def*)
done

lemma *COND-honest-hf-analz*:
assumes $ASID\ (AHI\ hf) \notin bad$ *hf-valid ainfo uinfo hfs hf* $ik\text{-}hf\ hf \subseteq synth\ (analz\ ik)$
no-oracle ainfo uinfo $hf \in set\ hfs$
shows $ik\text{-}hf\ hf \subseteq analz\ ik$
proof−
from *assms*(*3*) **have** *hf-synth-ik*: $HVF\ hf \in synth\ ik$ **by** *auto*
then have $\exists hfs.\ hf \in set\ hfs \land (ainfo,\ hfs) \in auth\text{-}seg2$
using *assms*(*1*,*2*,*4*,*5*)
apply(*auto simp add*: *ik-def hf-valid-invert ik-auth-hfs-simp*)
subgoal for *PoC-i-expire hf' hfs' PoC-i-expire'*
by(*auto intro*!: *exI*[*of - hfs'*] *elim*!: *back-subst*[**where** *?a=hf'*, **where** *?b=hf*]
simp add: *maccontents-def sntag-eq*)
by(*auto simp add*: *ik-auth-hfs-simp ik-def hf-valid-invert maccontents-def*
intro: *sntag-synth-bad dest*: *ik-add-form elim*: *ik-add-auth*)
then have $HVF\ hf \in ik$
using *assms*(*2*)
by(*auto simp add*: *ik-auth-hfs-def intro*!: *ik-ik-auth-hfs intro*!: *exI*)
then show *?thesis* **by** *auto*
qed

lemma *COND-ainfo-analz*:
assumes *hf-valid ainfo uinfo hfs hf* **and** $ik\text{-}auth\text{-}ainfo\ ainfo \in synth\ (analz\ ik)$
shows $ik\text{-}auth\text{-}ainfo\ ainfo \in analz\ ik$
using *assms* **by**(*auto simp add*: *hf-valid-invert*)

lemma *COND-ik-hf*:
assumes *hf-valid ainfo uinfo hfs hf* **and** $HVF\ hf \in ik$ **and** *no-oracle ainfo uinfo* **and** $hf \in set\ hfs$

shows $\exists$ *hfs*. *hf* $\in$ *set hfs* $\wedge$ (*ainfo*, *hfs*) $\in$ *auth-seg2*
using *assms* **apply**(*auto 3 4 simp add*: *hf-valid-invert ik-auth-hfs-simp ik-def dest*: *ahi-eq*)
using *assms*(*1*) *assms*(*2*) **apply**(*auto simp add*: *maccontents-def*)
apply(*frule sntag-eq*)
apply(*auto simp add*: *ik-def ik-auth-hfs-simp dest*: *ik-add-form*)
by (*metis info-hvf*(*1*) *info-hvf*(*2*))

lemma *COND-extr*:
$[\![$*hf-valid ainfo uinfo l hf*$]\!] \Longrightarrow$ *extr* (*HVF hf*) = *AHIS l*
by(*auto simp add*: *hf-valid-invert maccontents-def fullpath-def*)

lemma *COND-hf-valid-uinfo*:
$[\![$*hf-valid ainfo uinfo l hf*; *hf-valid ainfo' uinfo' l' hf*$]\!]$
$\Longrightarrow$ *uinfo'* = *uinfo*
by(*auto simp add*: *hf-valid-invert*)

3.7.5 Instantiation of *dataplane-3-undirected* locale

sublocale
dataplane-3-undirected - - - *auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf ik-add ik-oracle no-oracle*
apply *unfold-locales*
using *COND-ik-hf COND-honest-hf-analz COND-ainfo-analz COND-extr COND-hf-valid-uinfo* **by** *auto*

end
end

3.8 ICING variant

We abstract and simplify from the protocol ICING in several ways. First, we only consider Proofs of Consent (PoC), not Proofs of Provenance (PoP). Our framework does not support proving the path validation properties that PoPs provide, and it also currently does not support XOR, and dynamically changing hop fields. Thus, instead of embedding $A_i \oplus PoP_{0,1}$, we embed A_i directly. We also remove the payload from the Hash that is included in each packet.

We offer three versions of this protocol:

- *ICING*, which contains our best effort at modeling the protocol as accurately as possible.
- *ICING-variant*, in which we strip down the protocol to what is required to obtain the security guarantees and remove unnecessary fields.
- *ICING-variant2*, in which we furthermore simplify the protocol. The key of the MAC in this protocol is only the key of the AS, as opposed to a key derived specifically for this hop field. In order to prove that this scheme is secure, we have to assume that ASes only occur once on an authorized path, since otherwise the MAC for two different hop fields (by the same AS) would be the same, and the AS could not distinguish the hop fields based on the MAC.

theory *ICING-variant*
  imports
    ../Parametrized-Dataplane-3-undirected
begin

locale *icing-defs* = *network-assums-undirect - - - auth-seg0*
  for *auth-seg0* :: (*msgterm* × *ahi list*) *set*
begin

3.8.1 Hop validation check and extract functions

type-synonym *ICING-HF* = (*unit*, *unit*) *HF*

The term *sntag* is a key that is derived from the key of an AS and a specific hop field. We use it in the computation of *hf-valid*.

fun *sntag* :: *ahi* ⇒ *msgterm* **where**
  sntag (|*UpIF* = *upif*, *DownIF* = *downif*, *ASID* = *asid*|) = ⟨*macKey asid*,⟨*if2term upif*,*if2term downif*⟩⟩

lemma *sntag-eq*: *sntag ahi2* = *sntag ahi1* ⟹ *ahi2* = *ahi1*
  by(*cases ahi1*,*cases ahi2*) *auto*

The predicate *hf-valid* is given to the concrete parametrized model as a parameter. It ensures the authenticity of the hop authenticator in the hop field. The predicate takes an expiration timestamp (in this model always a numeric value, hence the matching on *Num PoC-i-expire*), the entire segment and the hop field to be validated.

fun *hf-valid* :: *msgterm* ⇒ *msgterm*
  ⇒ *ICING-HF list*

$\Rightarrow$ *ICING-HF*
$\Rightarrow$ *bool* **where**
hf-valid (*Num PoC-i-expire*) *uinfo hfs* (|*AHI* = *ahi*, *UHI* = *uhi*, *HVF* = *x*|) $\longleftrightarrow$ *uhi* = () $\wedge$
x = *Mac*[*sntag ahi*] (*L* ((*Num PoC-i-expire*)#(*map* (*hf2term o AHI*) *hfs*))) $\wedge$ *uinfo* = ε
| *hf-valid* - - - - = *False*

We can extract the entire path (past and future) from the hvf field.

fun *extr* :: *msgterm* $\Rightarrow$ *ahi list* **where**
extr (*Mac*[-] (*L hfs*))
= *map term2hf* (*tl hfs*)
| *extr* - = []

Extract the authenticated info field from a hop validation field.

fun *extr-ainfo* :: *msgterm* $\Rightarrow$ *msgterm* **where**
extr-ainfo (*Mac*[-] (*L* (*Num ts* # *xs*))) = *Num ts*
| *extr-ainfo* - = ε

abbreviation *ik-auth-ainfo* :: *msgterm* $\Rightarrow$ *msgterm* **where**
ik-auth-ainfo $\equiv$ *id*

An authenticated info field is always a number (corresponding to a timestamp).

abbreviation *checkInfo* **where**
checkInfo ainfo $\equiv$ ($\exists$ *ts*. *ainfo* = *Num ts*)

When observing a hop field, an attacker learns the HVF. UHI is empty and the AHI only contains public information that are not terms.

fun *ik-hf* :: *ICING-HF* $\Rightarrow$ *msgterm set* **where**
ik-hf hf = {*HVF hf*}

abbreviation *no-oracle* **where** *no-oracle* $\equiv$ (λ - -. *True*)

We now define useful properties of the above definition.

lemma *hf-valid-invert*:
hf-valid tsn uinfo hfs hf $\longleftrightarrow$
($\exists$ *ts ahi*. *tsn* = *Num ts* $\wedge$ *ahi* = *AHI hf* $\wedge$
UHI hf = () $\wedge$
HVF hf = *Mac*[*sntag ahi*] (*L* ((*Num ts*)#(*map* (*hf2term o AHI*) *hfs*))) $\wedge$ *uinfo* = ε)
apply(*cases hf*) **by**(*auto elim*!: *hf-valid.elims*)

lemma *hf-valid-checkInfo*[*dest*]: *hf-valid ainfo uinfo hfs hf* $\Longrightarrow$ *checkInfo ainfo*
by(*auto simp add*: *hf-valid-invert*)

lemma *info-hvf*:
assumes *hf-valid ainfo uinfo hfs m hf-valid ainfo' uinfo' hfs' m'*
HVF m = *HVF m' m* $\in$ *set hfs m'* $\in$ *set hfs'*
shows *ainfo'* = *ainfo m'* = *m*
using *assms*
apply(*auto simp add*: *hf-valid-invert intro*: *ahi-eq*)
apply(*cases m*,*cases m'*)
by(*auto intro*: *sntag-eq*)

3.8.2 Definitions and properties of the added intruder knowledge

Here we define a *ik-add* and *ik-oracle* as being empty, as these features are not used in this instance model.

sublocale *dataplane-3-undirected-defs - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf*
  by *unfold-locales*

declare *parts-singleton*[*dest*]

abbreviation *ik-add* :: *msgterm set* **where** *ik-add* ≡ {}

abbreviation *ik-oracle* :: *msgterm set* **where** *ik-oracle* ≡ {}

3.8.3 Properties of the intruder knowledge, including *ik-add* and *ik-oracle*

We now instantiate the parametrized model's definition of the intruder knowledge, using the definitions of *ik-add* and *ik-oracle* from above. We then prove the properties that we need to instantiate the *dataplane-3-undirected* locale.

sublocale
  dataplane-3-undirected-ik-defs - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo
    ik-hf ik-add ik-oracle no-oracle
  by *unfold-locales*

lemma *ik-auth-hfs-form*: $t \in$ *parts ik-auth-hfs* $\Longrightarrow \exists\ t'\ .\ t =$ *Hash* t'
  apply *auto* **apply**(*drule parts-singleton*)
  by(*auto simp add*: *auth-seg2-def hf-valid-invert*)

declare *ik-auth-hfs-def*[*simp del*]

lemma *parts-ik-auth-hfs*[*simp*]: *parts ik-auth-hfs* = *ik-auth-hfs*
  by (*auto intro*!: *parts-Hash ik-auth-hfs-form*)

This lemma allows us not only to expand the definition of *ik-auth-hfs*, but also to obtain useful properties, such as a term being a Hash, and it being part of a valid hop field.

lemma *ik-auth-hfs-simp*:
  $t \in$ *ik-auth-hfs* $\longleftrightarrow (\exists\, t'\ .\ t =$ *Hash* $t') \wedge (\exists\, hf\ .\ t =$ *HVF hf*
      $\wedge\ (\exists\, hfs.\ hf \in$ *set hfs* $\wedge\ (\exists\, ainfo\ .\ (ainfo,\ hfs) \in$ *auth-seg2*
      $\wedge\ (\exists\, uinfo.$ *hf-valid ainfo uinfo hfs hf*$))))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
  assume *asm*: *?lhs*
  then obtain *ainfo hf hfs* **where**
    dfs: $hf \in$ *set hfs* $(ainfo,\ hfs) \in$ *auth-seg2* $t =$ *HVF hf*
    by(*auto simp add*: *ik-auth-hfs-def*)
  then obtain *uinfo* **where** *hfs-valid-prefix ainfo uinfo* [] *hfs* = *hfs* (*ainfo*, *AHIS hfs*) ∈ *auth-seg0*
    by(*auto simp add*: *auth-seg2-def*)
  then show *?rhs* **using** *asm dfs* **by** *auto*(*fast intro*!: *ik-auth-hfs-form*)+
qed(*auto simp add*: *ik-auth-hfs-def*)

Properties of Intruder Knowledge

lemma *auth-ainfo*[*dest*]: $[\![(ainfo,\ hfs) \in$ *auth-seg2*$]\!] \Longrightarrow \exists\ ts\ .\ ainfo =$ *Num ts*

by(*auto simp add*: *auth-seg2-def*)

lemma *Num-ik*[*intro*]: *Num ts* $\in$ *ik*
by(*auto simp add*: *ik-def*)
(*auto simp add*: *auth-seg2-def elim*!: *allE*[*of* - []])

There are no ciphertexts (or signatures) in *parts ik*. Thus, *analz ik* and *parts ik* are identical.

lemma *analz-parts-ik*[*simp*]: *analz ik* = *parts ik*
apply(*rule no-crypt-analz-is-parts*)
by(*auto simp add*: *ik-def auth-seg2-def ik-auth-hfs-simp*)

lemma *parts-ik*[*simp*]: *parts ik* = *ik*
by(*auto 3 4 simp add*: *ik-def auth-seg2-def dest*!: *parts-singleton-set*)

lemma *sntag-synth-bad*: *sntag ahi* $\in$ *synth ik* $\Longrightarrow$ *ASID ahi* $\in$ *bad*
apply(*cases ahi*)
by(*auto simp add*: *ik-def ik-auth-hfs-simp*)

3.8.4 Direct proof goals for interpretation of *dataplane-3-undirected*

lemma *COND-honest-hf-analz*:
assumes *ASID* (*AHI hf*) $\notin$ *bad hf-valid ainfo uinfo hfs hf ik-hf hf* $\subseteq$ *synth* (*analz ik*)
no-oracle ainfo uinfo hf $\in$ *set hfs*
shows *ik-hf hf* $\subseteq$ *analz ik*
proof−
from *assms*(*3*) **have** *hf-synth-ik*: *HVF hf* $\in$ *synth ik* **by** *auto*
then have $\exists$*hfs*. *hf* $\in$ *set hfs* $\wedge$ (*ainfo*, *hfs*) $\in$ *auth-seg2*
using *assms*(*1*,*2*,*4*,*5*)
apply(*auto simp add*: *ik-def hf-valid-invert ik-auth-hfs-simp*)
subgoal for *hf′ hfs′ ts′*
using *HF.equality* **by** (*fastforce dest*!: *sntag-eq intro*: *exI*[*of* - *hfs′*])
by(*auto simp add*: *ik-auth-hfs-simp ik-def hf-valid-invert sntag-synth-bad*)
then have *HVF hf* $\in$ *ik*
using *assms*(*2*)
by(*auto simp add*: *ik-auth-hfs-def intro*!: *ik-ik-auth-hfs intro*!: *exI*)
then show *?thesis* **by** *auto*
qed

lemma *COND-ainfo-analz*:
assumes *hf-valid ainfo uinfo hfs hf* **and** *ik-auth-ainfo ainfo* $\in$ *synth* (*analz ik*)
shows *ik-auth-ainfo ainfo* $\in$ *analz ik*
using *assms* **by**(*auto simp add*: *hf-valid-invert*)

lemma *COND-ik-hf*:
assumes *hf-valid ainfo uinfo hfs hf* **and** *HVF hf* $\in$ *ik* **and** *no-oracle ainfo uinfo* **and** *hf* $\in$ *set hfs*
shows $\exists$*hfs*. *hf* $\in$ *set hfs* $\wedge$ (*ainfo*, *hfs*) $\in$ *auth-seg2*
using *assms* **apply**(*auto 3 4 simp add*: *hf-valid-invert ik-auth-hfs-simp ik-def dest*: *ahi-eq*)
apply(*frule sntag-eq*)
apply(*auto simp add*: *ik-def ik-auth-hfs-simp*)
by (*metis* (*mono-tags*, *lifting*) *HF.surjective old.unit.exhaust*)

lemma *COND-extr*:
$[\![$*hf-valid ainfo uinfo l hf*$]\!]$ $\Longrightarrow$ *extr* (*HVF hf*) = *AHIS l*

by(*auto simp add*: *hf-valid-invert*)

lemma *COND-hf-valid-uinfo*:
$[\![$*hf-valid ainfo uinfo l hf*; *hf-valid ainfo' uinfo' l' hf*$]\!]$
$\Longrightarrow$ *uinfo'* = *uinfo*
by(*auto simp add*: *hf-valid-invert*)

3.8.5 Instantiation of *dataplane-3-undirected* locale

sublocale
dataplane-3-undirected - - - *auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf ik-add ik-oracle no-oracle*
apply *unfold-locales*
using *COND-ik-hf COND-honest-hf-analz COND-ainfo-analz COND-extr COND-hf-valid-uinfo* **by** *auto*

end
end

3.9 ICING variant

We abstract and simplify from the protocol ICING in several ways. First, we only consider Proofs of Consent (PoC), not Proofs of Provenance (PoP). Our framework does not support proving the path validation properties that PoPs provide, and it also currently does not support XOR, and dynamically changing hop fields. Thus, instead of embedding $A_i \oplus PoP_{0,1}$, we embed A_i directly. We also remove the payload from the Hash that is included in each packet.

We offer three versions of this protocol:

- *ICING*, which contains our best effort at modeling the protocol as accurately as possible.
- *ICING-variant*, in which we strip down the protocol to what is required to obtain the security guarantees and remove unnecessary fields.
- *ICING-variant2*, in which we furthermore simplify the protocol. The key of the MAC in this protocol is only the key of the AS, as opposed to a key derived specifically for this hop field. In order to prove that this scheme is secure, we have to assume that ASes only occur once on an authorized path, since otherwise the MAC for two different hop fields (by the same AS) would be the same, and the AS could not distinguish the hop fields based on the MAC.

theory *ICING-variant2*
  imports
    ../Parametrized-Dataplane-3-undirected
begin

locale *icing-defs* = *network-assums-undirect* - - - *auth-seg0*
  for *auth-seg0* :: (*msgterm* × *ahi list*) *set*
+ **assumes** *auth-seg0-no-dups*:
  ⟦(*ainfo*, *hfs*) ∈ *auth-seg0*; *hf* ∈ *set hfs*; *hf'* ∈ *set hfs*; *ASID hf'* = *ASID hf*⟧ ⟹ *hf'* = *hf*
begin

3.9.1 Hop validation check and extract functions

type-synonym *ICING-HF* = (*unit*, *unit*) *HF*

The term *sntag* simply is the AS key. We use it in the computation of *hf-valid*.

fun *sntag* :: *ahi* ⇒ *msgterm* **where**
  sntag (|*UpIF* = *upif*, *DownIF* = *downif*, *ASID* = *asid*|) = *macKey asid*

The predicate *hf-valid* is given to the concrete parametrized model as a parameter. It ensures the authenticity of the hop authenticator in the hop field. The predicate takes an expiration timestamp (in this model always a numeric value, hence the matching on *Num PoC-i-expire*), the entire segment and the hop field to be validated.

fun *hf-valid* :: *msgterm* ⇒ *msgterm*
    ⇒ *ICING-HF list*
    ⇒ *ICING-HF*
    ⇒ *bool* **where**
  hf-valid (*Num PoC-i-expire*) *uinfo hfs* (|*AHI* = *ahi*, *UHI* = *uhi*, *HVF* = *x*|) ⟷ *uhi* = () ∧

$x = Mac[sntag\ ahi]\ (L\ ((Num\ PoC\text{-}i\text{-}expire)\#(map\ (hf2term\ o\ AHI)\ hfs))) \wedge uinfo = \varepsilon$
| *hf-valid - - - - = False*

We can extract the entire path (past and future) from the hvf field.

fun *extr* :: *msgterm* ⇒ *ahi list* **where**
extr (*Mac*[-] (*L hfs*))
= *map term2hf* (*tl hfs*)
| *extr* - = []

Extract the authenticated info field from a hop validation field.

fun *extr-ainfo* :: *msgterm* ⇒ *msgterm* **where**
extr-ainfo (*Mac*[-] (*L* (*Num ts* # *xs*))) = *Num ts*
| *extr-ainfo* - = ε

abbreviation *ik-auth-ainfo* :: *msgterm* ⇒ *msgterm* **where**
ik-auth-ainfo ≡ *id*

An authenticated info field is always a number (corresponding to a timestamp).

abbreviation *checkInfo* **where**
checkInfo ainfo ≡ (∃ *ts*. *ainfo* = *Num ts*)

When observing a hop field, an attacker learns the HVF. UHI is empty and the AHI only contains public information that are not terms.

fun *ik-hf* :: *ICING-HF* ⇒ *msgterm set* **where**
ik-hf hf = {*HVF hf*}

abbreviation *no-oracle* **where** *no-oracle* ≡ (λ - -. *True*)

We now define useful properties of the above definition.

lemma *hf-valid-invert*:
hf-valid tsn uinfo hfs hf ⟷
(∃ *ts ahi*. *tsn* = *Num ts* ∧ *ahi* = *AHI hf* ∧
UHI hf = () ∧
HVF hf = *Mac*[*sntag ahi*] (*L* ((*Num ts*)#(*map* (*hf2term o AHI*) *hfs*))) ∧ *uinfo* = ε)
apply(*cases hf*) **by**(*auto elim*!: *hf-valid.elims*)

lemma *hf-valid-checkInfo*[*dest*]: *hf-valid ainfo uinfo hfs hf* ⟹ *checkInfo ainfo*
by(*auto simp add*: *hf-valid-invert*)

3.9.2 Definitions and properties of the added intruder knowledge

Here we define a *ik-add* and *ik-oracle* as being empty, as these features are not used in this instance model.

sublocale *dataplane-3-undirected-defs - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf*
by *unfold-locales*

declare *parts-singleton*[*dest*]

abbreviation *ik-add* :: *msgterm set* **where** *ik-add* ≡ {}

abbreviation *ik-oracle* :: *msgterm set* **where** *ik-oracle* ≡ {}

3.9.3 Properties of the intruder knowledge, including *ik-add* and *ik-oracle*

We now instantiate the parametrized model's definition of the intruder knowledge, using the definitions of *ik-add* and *ik-oracle* from above. We then prove the properties that we need to instantiate the *dataplane-3-undirected* locale.

sublocale
  dataplane-3-undirected-ik-defs - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo
        ik-hf ik-add ik-oracle no-oracle
  by *unfold-locales*

lemma *ik-auth-hfs-form*: $t \in$ *parts ik-auth-hfs* $\Longrightarrow \exists\ t'$. t = *Hash* t'
  apply *auto* **apply**(*drule parts-singleton*)
  by(*auto simp add*: *auth-seg2-def hf-valid-invert*)

declare *ik-auth-hfs-def*[*simp del*]

lemma *parts-ik-auth-hfs*[*simp*]: *parts ik-auth-hfs* = *ik-auth-hfs*
  by (*auto intro*!: *parts-Hash ik-auth-hfs-form*)

This lemma allows us not only to expand the definition of *ik-auth-hfs*, but also to obtain useful properties, such as a term being a Hash, and it being part of a valid hop field.

lemma *ik-auth-hfs-simp*:
  $t \in$ *ik-auth-hfs* $\longleftrightarrow$ ($\exists\, t'$. t = *Hash* t') $\land$ ($\exists$ *hf* . t = *HVF hf*
      $\land$ ($\exists$ *hfs*. *hf* $\in$ *set hfs* $\land$ ($\exists$ *ainfo* . (*ainfo*, *hfs*) $\in$ *auth-seg2*
      $\land$ ($\exists$ *uinfo*. *hf-valid ainfo uinfo hfs hf*)))) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
  assume *asm*: *?lhs*
  then obtain *ainfo hf hfs* **where**
    dfs: *hf* $\in$ *set hfs* (*ainfo*, *hfs*) $\in$ *auth-seg2* t = *HVF hf*
    by(*auto simp add*: *ik-auth-hfs-def*)
  then obtain *uinfo* **where** *hfs-valid-prefix ainfo uinfo* [] *hfs* = *hfs* (*ainfo*, *AHIS hfs*) $\in$ *auth-seg0*
    by(*auto simp add*: *auth-seg2-def*)
  then show *?rhs* **using** *asm dfs* **by** *auto*(*fast intro*!: *ik-auth-hfs-form*)+
qed(*auto simp add*: *ik-auth-hfs-def*)

Properties of Intruder Knowledge

lemma *auth-ainfo*[*dest*]: $[\![$(*ainfo*, *hfs*) $\in$ *auth-seg2*$]\!]$ $\Longrightarrow \exists$ *ts* . *ainfo* = *Num ts*
  by(*auto simp add*: *auth-seg2-def*)

lemma *Num-ik*[*intro*]: *Num ts* $\in$ *ik*
  by(*auto simp add*: *ik-def*)
    (*auto simp add*: *auth-seg2-def elim*!: *allE*[*of* - []])

There are no ciphertexts (or signatures) in *parts ik*. Thus, *analz ik* and *parts ik* are identical.

lemma *analz-parts-ik*[*simp*]: *analz ik* = *parts ik*
  apply(*rule no-crypt-analz-is-parts*)
  by(*auto simp add*: *ik-def auth-seg2-def ik-auth-hfs-simp*)

lemma *parts-ik*[*simp*]: *parts ik* = *ik*
  by(*auto 3 4 simp add*: *ik-def auth-seg2-def dest*!: *parts-singleton-set*)

lemma *sntag-synth-bad*: *sntag ahi* ∈ *synth ik* ⟹ *ASID ahi* ∈ *bad*
  apply(*cases ahi*)
  by(*auto simp add*: *ik-def ik-auth-hfs-simp*)

lemma *back-subst-set-member*: ⟦*hf* ′ ∈ *set hfs*; *hf* ′ = *hf*⟧ ⟹ *hf* ∈ *set hfs* **by** *simp*

lemma *sntag-asid*: *sntag hf* = *sntag hf* ′ ⟹ *ASID hf* ′ = *ASID hf* **apply**(*cases hf*, *cases hf* ′) **by** *auto*

lemma *map-hf2term-eq*: *map* (λ*x*. *hf2term* (*AHI x*)) *hfs* = *map* (λ*x*. *hf2term* (*AHI x*)) *hfs*′
⟹ *AHIS hfs*′ = *AHIS hfs* **apply**(*induction hfs hfs*′ *rule*: *list-induct2*′, *auto*)
  using *term2hf-hf2term* **by** (*metis*)

3.9.4 Direct proof goals for interpretation of *dataplane-3-undirected*

lemma *COND-honest-hf-analz*:
  assumes *ASID* (*AHI hf*) ∉ *bad hf-valid ainfo uinfo hfs hf ik-hf hf* ⊆ *synth* (*analz ik*)
    no-oracle ainfo uinfo hf ∈ *set hfs*
    shows *ik-hf hf* ⊆ *analz ik*
proof−
  from *assms*(*3*) **have** *hf-synth-ik*: *HVF hf* ∈ *synth ik* **by** *auto*
  then have ∃ *hfs*. *hf* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
    using *assms*(*1*,*2*,*4*,*5*)
    apply(*auto simp add*: *ik-def hf-valid-invert ik-auth-hfs-simp*)
    subgoal for *hf* ′ *hfs*′ *ts*′
      apply (*auto intro*!: *exI*[*of - hfs*′])
      apply(*frule back-subst-set-member*[**where** *hfs*=*hfs*′])
      apply(*rule HF*.*equality*)
        apply *auto*
      apply(*drule sntag-asid*)
      apply(*drule map-hf2term-eq*)
      using *auth-seg0-no-dups*
      by (*metis* (*mono-tags*, *lifting*) *AHIS-set-rev HF*.*surjective auth-seg20 old*.*unit*.*exhaust*)
    by(*auto simp add*: *ik-auth-hfs-simp ik-def hf-valid-invert sntag-synth-bad*)
  then have *HVF hf* ∈ *ik*
    using *assms*(*2*)
    by(*auto simp add*: *ik-auth-hfs-def intro*!: *ik-ik-auth-hfs intro*!: *exI*)
  then show *?thesis* **by** *auto*
qed

lemma *COND-ainfo-analz*:
  assumes *hf-valid ainfo uinfo hfs hf* **and** *ik-auth-ainfo ainfo* ∈ *synth* (*analz ik*)
  shows *ik-auth-ainfo ainfo* ∈ *analz ik*
  using *assms* **by**(*auto simp add*: *hf-valid-invert*)

lemma *COND-ik-hf*:
  assumes *hf-valid ainfo uinfo hfs hf* **and** *HVF hf* ∈ *ik* **and** *no-oracle ainfo uinfo* **and** *hf* ∈ *set hfs*
  shows ∃ *hfs*. *hf* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2*
  using *assms* **apply**(*auto 3 4 simp add*: *hf-valid-invert ik-auth-hfs-simp ik-def dest*: *ahi-eq*)
  subgoal for *ts*′ *hf* ′ *hfs*′
      apply (*auto intro*!: *exI*[*of - hfs*′])
    apply(*frule back-subst-set-member*[**where** *hfs*=*hfs*′])
    apply *auto*

apply(*rule HF.equality*)
apply *auto*
apply(*drule sntag-asid*)
apply(*drule map-hf2term-eq*)
using *auth-seg0-no-dups*
by (*metis* (*mono-tags*, *lifting*) *AHIS-set-rev HF.surjective auth-seg20 old.unit.exhaust*)
done

lemma *COND-extr*:
$[\![$*hf-valid ainfo uinfo l hf*$]\!] \Longrightarrow$ *extr* (*HVF hf*) = *AHIS l*
by(*auto simp add*: *hf-valid-invert*)

lemma *COND-hf-valid-uinfo*:
$[\![$*hf-valid ainfo uinfo l hf*; *hf-valid ainfo' uinfo' l' hf*$]\!]$
$\Longrightarrow$ *uinfo'* = *uinfo*
by(*auto simp add*: *hf-valid-invert*)

3.9.5 Instantiation of *dataplane-3-undirected* locale

sublocale
dataplane-3-undirected - - - auth-seg0 hf-valid checkInfo extr extr-ainfo ik-auth-ainfo ik-hf ik-add ik-oracle no-oracle
apply *unfold-locales*
using *COND-ik-hf COND-honest-hf-analz COND-ainfo-analz COND-extr COND-hf-valid-uinfo* **by** *auto*

end
end

3.10 All Protocols

We import all protocols.

theory *All-Protocols*
imports
instances/SCION
instances/SCION-variant
instances/EPIC-L1-BA
instances/EPIC-L1-SA
instances/EPIC-L1-SA-Example
instances/EPIC-L2-SA
instances/ICING
instances/ICING-variant
instances/ICING-variant2
begin

end