IsaNet: Formalization of a Verification Framework for Secure Data Plane Protocols

Tobias Klenze, Christoph Sprenger

March 17, 2025

Contents

The paper presenting this formalization is to appear in the Journal of Computer Security under the title "IsaNet: A Framework for Verifying Secure Data Plane Protocols".

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 GUI shipped with Isabelle. See the README for details.

Abstract (from JCS paper)

Today's Internet is built on decades-old networking protocols that lack scalability, reliability and security. In response, the networking community has developed *path-aware* Internet architectures that solve these issues while simultaneously empowering end hosts. In these architectures, autonomous systems authorize forwarding paths in accordance with their routing policies, and protect paths using cryptographic authenticators. For each packet, the sending end host selects an authorized path and embeds it and its authenticators in the packet header. This allows routers to efficiently determine how to forward the packet. The central security property of the data plane, i.e., of forwarding, is that packets can only travel along authorized paths. This property, which we call *path authorization*, protects the routing policies of autonomous systems from malicious senders.

The fundamental role of packet forwarding in the Internet's ecosystem and the complexity of the authentication mechanisms employed call for a formal analysis. We develop IsaNet, a parameterized verification framework for data plane protocols in Isabelle/HOL. We first formulate an abstract model without an attacker for which we prove path authorization. We then refine this model by introducing a Dolev–Yao attacker and by protecting authorized paths using (generic) cryptographic validation fields. This model is parametrized by the path authorization mechanism and assumes five simple verification conditions. We propose novel attacker models and different sets of assumptions on the underlying routing protocol. We validate our framework by instantiating it with nine concrete protocols variants and prove that they each satisfy the verification conditions (and hence path authorization). The invariants needed for the security proof are proven in the parametrized model instead of the instance models. Our framework thus supports low-effort security proofs for data plane protocols. In contrast to what could be achieved with state-of-the-art automated protocol verifiers, our results hold for arbitrary network topologies and sets of authorized paths.

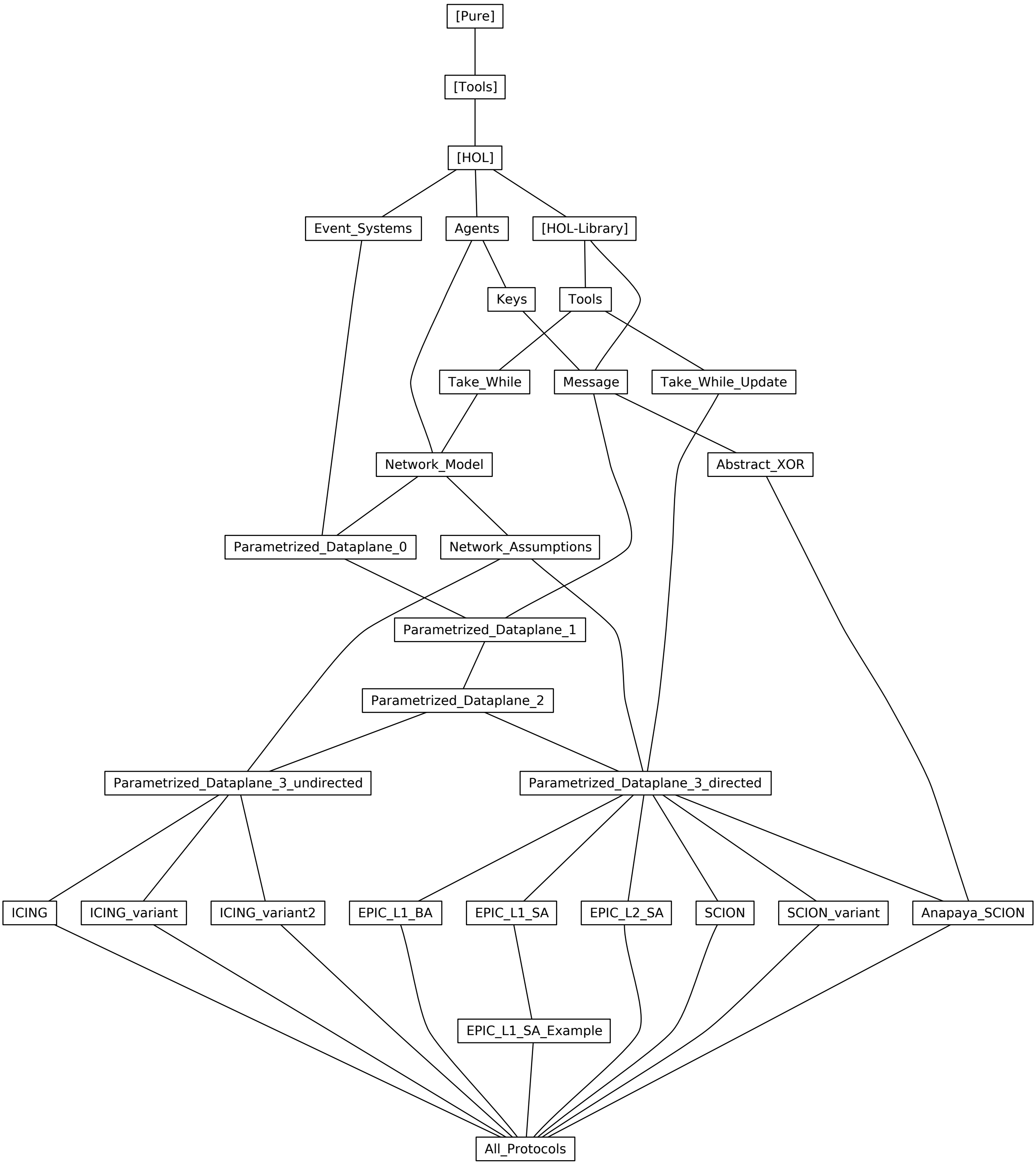

Figure 1: Theory dependencies

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-: - −⟨-⟩→ -)› [50, 50, 50] 90)
for $E\ s$
where
trace-nil [*simp*,*intro*!]:
$E: s\ -\langle[]\rangle\!\rightarrow s$
| *trace-snoc* [*intro*]:
$[\![\ E: s\ -\langle\tau\rangle\!\rightarrow s';\ E: s'\ -e\!\rightarrow s''\ ]\!] \Longrightarrow E: s\ -\langle\tau\ @\ [e]\rangle\!\rightarrow s''$

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*!]: $[\![\ E: s\ -e\!\rightarrow s'\ ]\!] \Longrightarrow E: s\ -\langle[e]\rangle\!\rightarrow s'$
by (*auto intro*: *trace-snoc*[**where** τ = [], *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
$E: s\ -\langle\tau\rangle\!\rightarrow s'$
$[\![\ \tau = [];\ s' = s\ ]\!] \Longrightarrow P$
$\bigwedge e\ \tau'\ s''.\ [\![\ \tau = e\ \#\ \tau';\ E: s\ -e\!\rightarrow s'';\ E: s''\ -\langle\tau'\rangle\!\rightarrow s'\ ]\!] \Longrightarrow P$
shows P
using *assms*
by (*induction rule*: *trace.induct*) *fastforce+*

lemma *trace-consD*: $(E: s\ -\langle e\ \#\ \tau\rangle\!\rightarrow s') \Longrightarrow \exists\ s''.\ (E: s\ -e\!\rightarrow s'') \wedge (E: s''\ -\langle\tau\rangle\!\rightarrow s')$
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*]:
  $[\![E\colon s -\langle\tau\rangle\rightarrow s''; F\colon t -\langle\sigma\rangle\rightarrow t''; \textit{length}\ \tau = \textit{length}\ \sigma;$
    $P\ []\ s\ []\ t;$
    $\bigwedge\tau\ s'\ e\ s''\ \sigma\ t'\ f\ t''.$
    $[\![E\colon s -\langle\tau\rangle\rightarrow s'; E\colon s' -e\rightarrow s''; F\colon t -\langle\sigma\rangle\rightarrow t'; F\colon t' -f\rightarrow t''; P\ \tau\ s'\ \sigma\ t']\!]$
      $\Longrightarrow P\ (\tau\ @\ [e])\ s''\ (\sigma\ @\ [f])\ t'']\!]$
  $\Longrightarrow P\ \tau\ s''\ \sigma\ t''$
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 ‹$\textit{length}\ (\tau\ @\ [e]) = \textit{length}\ \sigma$› **and** ‹$F\colon t -\langle\sigma\rangle\rightarrow t''$›
  obtain $f\ \sigma'\ t'$
    where $\sigma = \sigma'\ @\ [f]$ $\textit{length}\ \tau = \textit{length}\ \sigma'$ $F\colon t -\langle\sigma'\rangle\rightarrow t'$ $F\colon t' -f\rightarrow t''$
    by (*auto elim*: *trace.cases*)
  then show *?case* **using** *trace-snoc* **by** *blast*
qed

Relate traces to reachability and invariants

lemma *reach-trace-equiv*: $\textit{reach}\ E\ s \longleftrightarrow (\exists s0\ \tau.\ \textit{init}\ E\ s0 \wedge E\colon s0 -\langle\tau\rangle\rightarrow s)$ (**is** *?A* $\longleftrightarrow$ *?B*)
proof
  assume *?A* **then show** *?B*
    by (*induction s rule*: *reach.induct*) *auto*
next
  assume *?B*
  then obtain *s0* τ **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*: $[\![\textit{init}\ E\ s0; E\colon s0 -\langle\tau\rangle\rightarrow s]\!] \Longrightarrow \textit{reach}\ E\ s$
  by(*auto simp add*: *reach-trace-equiv*)

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

lemma *Inv-trace*: $[\![\textit{Inv}\ E\ I; \textit{init}\ E\ s0; E\colon s0 -\langle\tau\rangle\rightarrow s']\!] \Longrightarrow I\ s'$
  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*: $s\ -\langle\tau\rangle\rightarrow\ s'\}$

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

lemma *tracesE* [*elim*]: $[\![\ \tau \in$ *traces E*; $\bigwedge s\ s'.\ [\![$ *init E s*; *E*: $s\ -\langle\tau\rangle\rightarrow\ s'$ $]\!] \Longrightarrow P\ ]\!] \Longrightarrow P$
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*: $s\ -\langle\tau\rangle\rightarrow\ s'$ $]\!] \Longrightarrow I\ \tau\ s'$
$\bigwedge s.\ I\ \tau\ s \Longrightarrow \tau \in \varphi$
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*: $s\ -\langle\tau\rangle\rightarrow\ s'$; *E*: $s'\ -e\rightarrow\ s''$; $I\ \tau\ s'$; *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*: $s\ -\langle\tau\rangle\rightarrow\ s'$ **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*: $s\ -\langle\tau\rangle\rightarrow\ s'$; *?E*: $s'{-}e{\rightarrow}\ s''$; *?I* τ s'; *reach ?E s'*$]\!] \Longrightarrow$ *?I* $(\tau\ @\ [e])\ s''$; $\bigwedge\tau\ s.\ [\![$*?I* τ *s*; *reach ?E s*$]\!] \Longrightarrow \tau \in$ *?φ*$]\!] \Longrightarrow$ *?E* $\models_{ES}$ *?φ*, but allows matching pure state invariants directly.

lemma *Inv-trace-property*:

assumes *Inv E I* **and** $[] \in \varphi$
and $(\bigwedge s\ \tau\ s'\ e\ s''.$
$[\![init\ E\ s;\ E\colon s\ -\langle\tau\rangle\rightarrow\ s';\ E\colon s'\ -e\rightarrow\ s'';\ I\ s;\ I\ s';\ reach\ E\ s';\ \tau \in \varphi]\!] \Longrightarrow \tau@[e] \in \varphi)$
shows $E \models_{ES} \varphi$
using *assms(1,2)*
by (*intro trace-property-rule*[**where** $I{=}\lambda\tau\ s.\ \tau \in \varphi$]) (*auto intro*: *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
$[\![\ \bigwedge e\ s'.\ (E\colon s\ -e\rightarrow\ s') \Longrightarrow \exists\, t'.\ (F\colon t\ -\pi\ e\rightarrow\ t') \wedge sim\ E\ F\ \pi\ s'\ t'\ ]\!] \Longrightarrow sim\ E\ F\ \pi\ s\ t$

abbreviation
$simS :: ('e,\ 's\ )\ ES \Rightarrow ('f,\ 't\ )\ ES \Rightarrow 's \Rightarrow ('e \Rightarrow 'f) \Rightarrow 't \Rightarrow bool$
(‹(5-,-: - ⊑- -)› [*50*, *50*, *50*, *60*, *50*] *90*)
where
$simS\ E\ F\ s\ \pi\ t \equiv sim\ E\ F\ \pi\ s\ t$

lemmas *sim-coinduct-id* = *sim.coinduct*[**where** π=*id*, *consumes 1*, *case-names sim*]

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

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

lemma *sim-refl*: $E,E\colon s \sqsubseteq_i d\ s$
by (*coinduction arbitrary*: *s*) *auto*

lemma *sim-trans*: $[\![\ E,F\colon s \sqsubseteq_{\pi}1\ t;\ F,G\colon t \sqsubseteq_{\pi}2\ u\ ]\!] \Longrightarrow E,G\colon s \sqsubseteq_{(}\pi2 \circ \pi1)\ u$
proof (*coinduction arbitrary*: *s t u*)
case (*sim a s' s t*)
with ‹$E,F\colon s \sqsubseteq_{\pi}1\ t$› **obtain** t' **where** $F\colon t\ -\pi1\ a\rightarrow\ t'\ E,F\colon s' \sqsubseteq_{\pi}1\ t'$
by (*cases rule*: *sim.cases*) *auto*
moreover
from ‹$F,G\colon t \sqsubseteq_{\pi}2\ u$› ‹$F\colon t\ -\pi1\ a\rightarrow\ t'$› **obtain** u' **where** $G\colon u\ -\pi2\ (\pi1\ a)\rightarrow\ u'\ F,G\colon t' \sqsubseteq_{\pi}2\ u'$
by (*cases rule*: *sim.cases*) *auto*
ultimately
have $\exists\, t'\ u'.\ (G\colon u\ -\pi2\ (\pi1\ a)\rightarrow\ u') \wedge (E,F\colon s' \sqsubseteq_{\pi}1\ t') \wedge (F,G\colon t' \sqsubseteq_{\pi}2\ u')$

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

Extend transition simulation to traces.

lemma *trace-sim*:
assumes $E\colon s -\langle\tau\rangle\rightarrow s'$ $E,F\colon s \sqsubseteq_\pi t$
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** $F\colon t -\langle map\ \pi\ \tau\rangle\rightarrow t'$ $E,F\colon s' \sqsubseteq_\pi t'$ **by** *auto*
from ‹$E,F\colon s' \sqsubseteq_\pi t'$› ‹$E\colon s' - e\rightarrow s''$›
obtain t'' **where** $F\colon t' -\pi\ e\rightarrow t''$ $E,F\colon s'' \sqsubseteq_\pi t''$ **by** (*elim sim.cases*) *fastforce*
then show *?case* **using** ‹$F\colon t -\langle map\ \pi\ \tau\rangle\rightarrow t'$› ‹$E\colon s -\langle\tau\rangle\rightarrow s'$› ‹$E\colon s' - e\rightarrow s''$› **by** *auto*
qed

Simulation for event systems

definition
sim-ES :: $('e, 's)\ ES \Rightarrow ('e \Rightarrow 'f) \Rightarrow ('f, 't)\ ES \Rightarrow bool$ (‹(3- $\sqsubseteq$- -)› [*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-ES-I*[**where** $R{=}\lambda s\ t.\ R\ s\ t \wedge reach\ E\ s \wedge reach\ F\ t$] *dest*: *init*
intro: *sim-coinduct-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: $\bigwedge s\ t\ a\ s'.$
$[\![\ R\ s\ t;\ I\ s;\ J\ t;\ E\colon s{-}a{\rightarrow}\ s'\ ]\!] \Longrightarrow (\exists\, t'.\ (F\colon t{-}\pi\ a{\rightarrow}\ t') \wedge R\ s'\ t')$ **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: $\bigwedge s\ a\ s'.\ [\![\ E\colon s{-}a{\rightarrow}\ s';\ reach\ E\ s;\ reach\ F\ (h\ s)\ ]\!] \Longrightarrow F\colon h\ s{-}\pi\ a{\rightarrow}\ h\ s'$
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: $\bigwedge s\ a\ s'.\ [\![\ E\colon s{-}a{\rightarrow}\ s';\ I\ s;\ J\ (h\ s)\ ]\!] \Longrightarrow F\colon h\ s{-}\pi\ a{\rightarrow}\ h\ s'$ **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*: $E \sqsubseteq_{id}\ E$
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 ‹$E \sqsubseteq_\pi 1\ F$› **obtain** R_1 **where**
$\bigwedge s0.\ init\ E\ s0 \Longrightarrow (\exists\, t0.\ init\ F\ t0 \wedge R_1\ s0\ t0)$
$\bigwedge s\ t.\ R_1\ s\ t \Longrightarrow E{,}F\colon s \sqsubseteq_\pi 1\ t$
by (*auto elim*!: *sim-ES-E*)
moreover
from ‹$F \sqsubseteq_\pi 2\ G$› **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*: $E \sqsubseteq_\pi F \Longrightarrow (map\ \pi)\text{`}traces\ E \subseteq traces\ F$
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 HOL.Finite-Set HOL−Library.FSet*
begin

datatype *msgterm* =
  ε — Empty message. Used for instance to denote non-existent interface
  | *AS as* — Autonomous System identifier, i.e. agents. Note that AS is an alias of nat
  | *Num nat* — Ordinary integers, timestamps, ...
  | *Key key* — Crypto keys
  | *Nonce nonce* — Unguessable nonces
  | *L msgterm list* — Lists
  | *FS msgterm fset* — Finite Sets. Used to represent XOR values.
  | *MPair msgterm msgterm* — Compound messages
  | *Hash msgterm* — Hashing
  | *Crypt key msgterm* — Encryption, public- or shared-key

Syntax sugar

syntax
  -MTuple :: ['a, args] ⇒ 'a ∗ 'b (‹(‹indent=2 notation=‹mixfix message tuple››⟨-,/ -⟩)›)
syntax-consts
  -MTuple ⇌ *MPair*
translations
  ⟨x, y, z⟩ ⇌ ⟨x, ⟨y, z⟩⟩
  ⟨x, y⟩ ⇌ *CONST MPair x y*

syntax
  -MHF :: ['a, 'b , 'c, 'd, 'e] ⇒ '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. ∃X. *Crypt K X* ∈ H}

Inductive Definition of "All Parts" of a Message

inductive-set
  parts :: *msgterm set* ⇒ *msgterm set*
  for H :: *msgterm set*
  where
    Inj [*intro*]: X ∈ H ⟹ X ∈ *parts* H
  | *Fst*: ⟨X,-⟩ ∈ *parts* H ⟹ X ∈ *parts* H
  | *Snd*: ⟨-,Y⟩ ∈ *parts* H ⟹ Y ∈ *parts* H

| *Lst*: ⟦ *L xs* ∈ *parts H*; *X* ∈ *set xs* ⟧ ⟹ *X* ∈ *parts H*
| *FSt*: ⟦ *FS xs* ∈ *parts H*; *X* |∈| *xs* ⟧ ⟹ *X* ∈ *parts H*

| *Body*: *Crypt K X* ∈ *parts H* ⟹ *X* ∈ *parts H*

Monotonicity

lemma *parts-mono*: *G* ⊆ *H* ⟹ *parts G* ⊆ *parts H*
apply *auto*
apply (*erule parts.induct*)
apply (*blast dest*: *parts.Fst parts.Snd parts.Lst parts.FSt parts.Body*)+
done

Equations hold because constructors are injective.

lemma *Other-image-eq* [*simp*]: (*AS x* ∈ *AS‘A*) = (*x*:*A*)
by *auto*

lemma *Key-image-eq* [*simp*]: (*Key x* ∈ *Key‘A*) = (*x*∈*A*)
by *auto*

lemma *AS-Key-image-eq* [*simp*]: (*AS x* ∉ *Key‘A*)
by *auto*

lemma *Num-Key-image-eq* [*simp*]: (*Num x* ∉ *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* ∪ *H′*) = *keysFor H* ∪ *keysFor H′*
by (*unfold keysFor-def*, *blast*)

lemma *keysFor-UN* [*simp*]: *keysFor* (⋃ *i*∈*A*. *H i*) = (⋃ *i*∈*A*. *keysFor* (*H i*))
by (*unfold keysFor-def*, *blast*)

Monotonicity

lemma *keysFor-mono*: *G* ⊆ *H* ⟹ *keysFor G* ⊆ *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-def*, *auto*)

lemma *keysFor-insert-FS* [*simp*]: *keysFor* (*insert* (*FS X*) *H*) = *keysFor H*
by (*unfold keysFor-def*, *auto*)

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

lemma *keysFor-insert-MPair* [*simp*]: *keysFor* (*insert* ⟨*X*,*Y*⟩ *H*) = *keysFor H*
by (*unfold keysFor-def*, *auto*)

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

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

lemma *Crypt-imp-invKey-keysFor*: *Crypt K X* ∈ *H* ⟹ *invKey K* ∈ *keysFor H*
by (*unfold keysFor-def*, *blast*)

1.4.2 Inductive relation "parts"

lemma *MPair-parts*:
⟦
⟨*X*,*Y*⟩ ∈ *parts H*;
⟦ *X* ∈ *parts H*; *Y* ∈ *parts H* ⟧ ⟹ *P*
⟧ ⟹ *P*
by (*blast dest*: *parts.Fst parts.Snd*)

lemma *L-parts*:
⟦
L l ∈ *parts H*;
⟦ *set l* ⊆ *parts H* ⟧ ⟹ *P*
⟧ ⟹ *P*
by (*blast dest*: *parts.Lst*)

lemma *FS-parts*:
⟦
FS l ∈ *parts H*;
⟦ *fset l* ⊆ *parts H* ⟧ ⟹ *P*
⟧ ⟹ *P*
by (*simp add*: *parts.FSt subsetI*)
thm *parts.FSt subsetI*

declare *MPair-parts* [*elim*!] *L-parts* [*elim*!] *FS-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* ⊆ *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*∈ *parts*{} ⟹ *P*
by *simp*

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

lemma *parts-singleton*: *X* ∈ *parts H* ⟹ ∃ *Y* ∈ *H*. *X* ∈ *parts* {*Y*}
  apply (*erule parts.induct*, *fast*)
  using *parts.FSt* **by** *blast*+

lemma *parts-singleton-set*: *x* ∈ *parts* {*s* . *P s*} ⟹ ∃ *Y*. *P Y* ∧ *x* ∈ *parts* {*Y*}
  by(*auto dest*: *parts-singleton*)

lemma *parts-singleton-set-rev*: ⟦*x* ∈ *parts* {*Y*}; *P Y*⟧ ⟹ *x* ∈ *parts* {*s* . *P s*}
  by (*induction rule*: *parts.induct*)
    (*blast dest*: *parts.Fst parts.Snd parts.Lst parts.FSt parts.Body*)+

lemma *parts-Hash*: ⟦⋀*t* . *t* ∈ *H* ⟹ ∃ *t′* . *t* = *Hash t′*⟧ ⟹ *parts H* = *H*
  by (*auto*, *erule parts.induct*, *fast*+)

Unions

lemma *parts-Un-subset1*: *parts G* ∪ *parts H* ⊆ *parts*(*G* ∪ *H*)
by (*intro Un-least parts-mono Un-upper1 Un-upper2*)

lemma *parts-Un-subset2*: *parts*(*G* ∪ *H*) ⊆ *parts G* ∪ *parts H*
  by (*rule subsetI*) (*erule parts.induct*, *blast*+)

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

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

lemma *parts-two*: ⟦*x* ∈ *parts* {*e1*, *e2*}; *x* ∉ *parts* {*e1*}⟧⟹ *x* ∈ *parts* {*e2*}

by (*simp add*: *parts-insert2*)

Added to simplify arguments to parts, analz and synth.

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

Idempotence

lemma *parts-partsD* [*dest*!]: *X*∈ *parts* (*parts H*) ⟹ *X*∈ *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* ⊆ *parts H*) = (*G* ⊆ *parts H*)
apply (*rule iffI*)
apply (*iprover intro*: *subset-trans parts-increasing*)
apply (*frule parts-mono*, *simp*)
done

Transitivity

lemma *parts-trans*: ⟦ *X*∈ *parts G*; *G* ⊆ *parts H* ⟧ ⟹ *X*∈ *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* = ⋃ { *parts* {*h*} | *h* . *h* ∈ *H*}
apply *auto*
apply (*erule parts.induct*)
apply (*blast dest*: *parts.Fst parts.Snd parts.Lst parts.FSt parts.Body*)+
using *parts-trans* **apply** *blast*
done

Cut

lemma *parts-cut*:
⟦ *Y*∈ *parts* (*insert X G*); *X* ∈ *parts H* ⟧ ⟹ *Y* ∈ *parts* (*G* ∪ *H*)
by (*blast intro*: *parts-trans*)

lemma *parts-cut-eq* [*simp*]: *X* ∈ *parts H* ⟹ *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 elim*!: *FS-parts*)

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-insert-FS* [*simp*]:
parts (*insert* (*FS xs*) *H*) =
insert (*FS xs*) (*parts* ((*fset xs*) ∪ *H*))
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule parts.induct*, *auto*)
by (*auto intro*: *parts.FSt*)+

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

Parts of lists and finite sets.

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

lemma *parts-fset-set* :
parts (*FS'ls*) = (*FS'ls*) ∪ (⋃ *l* ∈ *ls*. *parts* (*fset l*))
apply (*rule equalityI*, *rule subsetI*)
apply (*erule parts.induct*, *auto*)
by (*meson FS-parts image-subset-iff parts-increasing parts-trans*)

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*] : $X \in H \Longrightarrow X \in analz\ H$
| *Fst*: $\langle X, Y \rangle \in analz\ H \Longrightarrow X \in analz\ H$
| *Snd*: $\langle X, Y \rangle \in analz\ H \Longrightarrow Y \in analz\ H$
| *Lst*: $(L\ y) \in analz\ H \Longrightarrow x \in set\ (y) \Longrightarrow x \in analz\ H$
| *FSt*: $[\![\ FS\ xs \in analz\ H;\ X\ |\in|\ xs\ ]\!] \Longrightarrow X \in analz\ H$
| *Decrypt* [*dest*]: $[\![\ Crypt\ K\ X \in analz\ H;\ Key\ (invKey\ K) \in analz\ H\ ]\!] \Longrightarrow X \in analz\ H$

Monotonicity; Lemma 1 of Lowe's paper

lemma *analz-mono*: $G \subseteq H \Longrightarrow analz(G) \subseteq analz(H)$
apply *auto*
apply (*erule analz.induct*)
apply (*auto dest*: *analz.Fst analz.Snd analz.Lst analz.FSt*)
done

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

Making it safe speeds up proofs

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

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

lemma *FS-analz* [*elim*!]:
$[\![$
$FS\ l \in analz\ H;$
$[\![\ fset\ l \subseteq analz\ H\ ]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
by (*simp add*: *analz.FSt subsetI*)

thm *parts.FSt subsetI*
lemma *analz-increasing*: $H \subseteq analz(H)$
by *blast*

lemma *analz-subset-parts*: $analz\ H \subseteq parts\ H$

by (*rule subsetI*) (*erule analz.induct*, *blast*+)

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*+, *auto*)

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

lemma *analz-insert*: *insert X* (*analz H*) $\subseteq$ *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*]:
  $K \notin keysFor\ (analz\ H) \Longrightarrow$
    $analz\ (insert\ (Key\ K)\ H) = insert\ (Key\ K)\ (analz\ H)$
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*]:
  $analz\ (insert\ (L\ l)\ H) = insert\ (L\ l)\ (analz\ (set\ l \cup H))$
apply (*rule equalityI*)
apply (*rule subsetI*)
apply (*erule analz.induct, auto*)
apply (*erule analz.induct, auto*)
using *analz.Inj* **by** *blast*

lemma *analz-insert-FS* [*simp*]:
  $analz\ (insert\ (FS\ l)\ H) = insert\ (FS\ l)\ (analz\ (fset\ l \cup H))$
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*) ∉ *analz H*
⟹ *analz* (*insert* (*Crypt K X*) *H*) = *insert* (*Crypt K X*) (*analz H*)
apply (*rule analz-insert-eq-I*)
by (*erule analz.induct*, *auto*)

lemma *analz-insert-Decrypt1*:
Key (*invKey K*) ∈ *analz H* ⟹
analz (*insert* (*Crypt K X*) *H*) ⊆
insert (*Crypt K X*) (*analz* (*insert X H*))
apply (*rule subsetI*)
by (*erule-tac x* = *x* **in** *analz.induct*, *auto*)

lemma *analz-insert-Decrypt2*:
Key (*invKey K*) ∈ *analz H* ⟹
insert (*Crypt K X*) (*analz* (*insert X H*)) ⊆
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*) ∈ *analz H* ⟹
analz (*insert* (*Crypt K X*) *H*) =
insert (*Crypt K X*) (*analz* (*insert X H*))
by (*intro equalityI analz-insert-Decrypt1 analz-insert-Decrypt2*)

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*, *auto*)

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*:
⟦ *analz G* = *analz G′*; *analz H* = *analz H′* ⟧
⟹ *analz* (*G* ∪ *H*) = *analz* (*G′* ∪ *H′*)
by (*intro equalityI analz-subset-cong*, *simp-all*)

lemma *analz-insert-cong*:
analz H = *analz H′* ⟹ *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*:
⟦
∀ *X Y*. ⟨*X*,*Y*⟩ ∉ *H*; ∀ *xs*. *L xs* ∉ *H*; ∀ *xs*. *FS xs* ∉ *H*;
∀ *X K*. *Crypt K X* ∉ *H*
⟧ ⟹ *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∈ *analz* (⋃*i*∈*A*. *analz* (*H i*)) ⟹ *X*∈ *analz* (⋃*i*∈*A*. *H i*)

apply (*erule analz.induct*)
by (*auto intro*: *analz-mono* [*THEN* [*2*] *rev-subsetD*])

lemma *analz-UN-analz* [*simp*]: *analz* ($\bigcup$ *i*∈*A*. *analz* (*H i*)) = *analz* ($\bigcup$ *i*∈*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*:
  ¬(∃ *K* . *Key K* ∈ *analz H*)
    ⟹ *analz H* = $\bigcup$ { *analz* {*h*} | *h* . *h* ∈ *H*}
apply *auto*
subgoal
  apply (*erule analz.induct*)
  apply (*auto dest*: *analz.Fst analz.Snd analz.Lst analz.FSt*)
done
  apply (*erule analz.induct*)
  apply (*auto dest*: *analz.Fst analz.Snd analz.Lst analz.FSt*)
done

lemma *analz-Un-eq*:
  assumes ¬(∃ *K* . *Key K* ∈ *analz H*) **and** ¬(∃ *K* . *Key K* ∈ *analz G*)
  shows *analz* (*H* ∪ *G*) = *analz H* ∪ *analz G*
apply (*intro equalityI*, *rule subsetI*)
apply (*erule analz.induct*)
using *assms* **by** *auto*

lemma *analz-Un-eq-Crypt*:
  assumes ¬(∃ *K* . *Key K* ∈ *analz G*) **and** ¬(∃ *K X* . *Crypt K X* ∈ *analz G*)
  shows *analz* (*H* ∪ *G*) = *analz H* ∪ *analz G*
apply (*intro equalityI*, *rule subsetI*)
apply (*erule analz.induct*)
using *assms* **by** *auto*

lemma *analz-list-set* :
  ¬(∃ *K* . *Key K* ∈ *analz* (*L*‘*ls*))
    ⟹ *analz* (*L*‘*ls*) = (*L*‘*ls*) ∪ ($\bigcup$ *l* ∈ *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-fset-set* :
  ¬(∃ *K* . *Key K* ∈ *analz* (*FS*‘*ls*))
    ⟹ *analz* (*FS*‘*ls*) = (*FS*‘*ls*) ∪ ($\bigcup$ *l* ∈ *ls*. *analz* (*fset l*))
apply (*rule equalityI*, *rule subsetI*)
apply (*erule analz.induct*, *auto*)
using *FS-analz image-subset-iff analz-increasing analz-trans* **by** *metis*

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* ⇒ *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*]: $[\![\bigwedge x\ .\ x \in set\ xs \Longrightarrow x \in synth\ H]\!] \Longrightarrow L\ xs \in synth\ H$
| *FSt* [*intro*]: $[\![\bigwedge x\ .\ x \in fset\ xs \Longrightarrow x \in synth\ H;$
$\bigwedge x\ ys\ .\ x \in fset\ xs \Longrightarrow x \neq FS\ ys]\!]$
$\Longrightarrow FS\ xs \in synth\ H$
| *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*: $G \subseteq H \Longrightarrow synth(G) \subseteq synth(H)$
apply (*auto*, *erule synth.induct*, *auto*)
by *blast*

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 *FS-synth* [*elim*!]: $FS\ X \in synth\ H$
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* **by** *blast*

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*]: $(Nonce\ N \in synth\ H) = (Nonce\ N \in H)$
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*

lemma *FS-cons-synth* [*simp*]:
  $[\![fset\ xs \subseteq H;\ \bigwedge x\ ys.\ x \in fset\ xs \Longrightarrow x \neq FS\ ys;\ fcard\ xs \neq Suc\ 0\ ]\!] \Longrightarrow (FS\ xs \in synth\ H)$
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*)

(*auto intro*: *parts.Fst parts.Snd parts.Lst parts.FSt parts.Body*)
next
fix *X*
assume $X \in$ *parts H*
thus $X \in$ *parts* (*synth H*)
by (*induction rule*: *parts.induct*)
(*auto intro*: *parts.Fst parts.Snd parts.Lst parts.FSt parts.Body*)
next
fix *X*
assume $X \in$ *synth H*
thus $X \in$ *parts* (*synth H*)
apply (*induction rule*: *synth.induct*)
apply(*auto intro*: *parts.Fst parts.Snd parts.Lst parts.FSt parts.Body*)
by *blast*
qed

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

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

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

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

lemma *analz-synth-Un2* [*simp*]: *analz* (*G* ∪ *synth H*) = *analz* (*G* ∪ *H*) ∪ *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*) ⊆ *parts G* ∪ *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*) ∪ *parts H*

lemma *Fake-parts-insert*:
$X \in$ *synth* (*analz H*) $\Longrightarrow$
parts (*insert X H*) ⊆ *synth* (*analz H*) ∪ *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 *FS-cons-synth-analz* [*iff*]:
  $[\![\bigwedge x\ ys\ .\ x \in fset\ xs \Longrightarrow x \neq FS\ ys;\ fcard\ xs \neq Suc\ 0\ ]\!] \Longrightarrow$
  $(FS\ xs \in synth\ (analz\ H)) =$
  $(fset\ xs \subseteq synth\ (analz\ H))$
by *blast*

lemma *FS-cons-synth-parts* [*iff*]:
  $[\![\bigwedge x\ ys\ .\ x \in fset\ xs \Longrightarrow x \neq FS\ ys;\ fcard\ xs \neq Suc\ 0\ ]\!] \Longrightarrow$
    $(FS\ xs \in synth\ (parts\ H)) =$
    $(fset\ xs \subseteq synth\ (parts\ H))$
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* ∉ *Key* ' *A*
by *auto*

lemma *synth-analz-mono*: *G*⊆*H* ⟹ *synth* (*analz*(*G*)) ⊆ *synth* (*analz*(*H*))
by (*iprover intro*: *synth-mono analz-mono*)

lemma *synth-parts-mono*: *G*⊆*H* ⟹ *synth* (*parts G*) ⊆ *synth* (*parts H*)
by (*iprover intro*: *synth-mono parts-mono*)

lemma *Fake-analz-eq* [*simp*]:
  X ∈ *synth*(*analz H*) ⟹ *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 ∈ *analz H* ⟹ *ALL G*. *H* ⊆ *G* −−> *analz* (*insert X G*) = *analz G*
by (*blast intro*: *analz-cut analz-insertI analz-mono* [*THEN* [*2*] *rev-subsetD*])

lemma *Fake-parts-sing*:
  X ∈ *synth* (*analz H*) ⟹ *parts*{*X*} ⊆ *synth* (*analz H*) ∪ *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* ⊆ *synth* (*analz H'*)
  shows *analz* (*insert X H*) ⊆ *synth* (*analz* (*insert X H'*))
proof
  fix *x*
  assume *x* ∈ *analz* (*insert X H*)
  then have *x* ∈ *analz* (*insert X* (*synth* (*analz H'*)))
    using *assms* **by** (*meson analz-increasing analz-monotonic insert-mono*)
  then show *x* ∈ *synth* (*analz* (*insert X H'*))
    by (*metis* (*no-types*) *Un-iff analz-idem analz-insert analz-monotonic analz-synth synth.Inj*
      synth-insert synth-mono)
qed

lemma *synth-parts-insert*:

assumes *parts H* ⊆ *synth* (*parts H′*)
shows *parts* (*insert X H*) ⊆ *synth* (*parts* (*insert X H′*))
proof
fix *x*
assume *x* ∈ *parts* (*insert X H*)
then have *x* ∈ *parts* (*insert X* (*synth* (*parts H′*)))
using *assms parts-increasing*
by (*metis UnE UnI1 analz-monotonic analz-parts parts-insert parts-insertI*)
then show *x* ∈ *synth* (*parts* (*insert X H′*))
using *Un-iff parts-idem parts-insert parts-synth synth.Inj*
by (*metis Un-subset-iff synth-increasing synth-trans*)
qed

lemma *parts-insert-subset-impl*:
⟦*x* ∈ *parts* (*insert a G*); *x* ∈ *parts G* ⟹ *x* ∈ *synth* (*parts H*); *a* ∈ *synth* (*parts H*)⟧
⟹ *x* ∈ *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-types*, *lifting*) *analz-parts*)

lemma *synth-parts-subset-elem*:
⟦*A* ⊆ *synth* (*parts B*); *x* ∈ *parts A*⟧ ⟹ *x* ∈ *synth* (*parts B*)
by (*meson parts-emptyE parts-insert-subset-impl parts-singleton subset-iff*)

lemma *synth-parts-subset*:
A ⊆ *synth* (*parts B*) ⟹ *parts A* ⊆ *synth* (*parts B*)
by (*auto simp add*: *synth-parts-subset-elem*)

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

lemma *synth-parts-trans*:
assumes *A* ⊆ *synth* (*parts B*) **and** *B* ⊆ *synth* (*parts C*)
shows *A* ⊆ *synth* (*parts C*)
using *assms* **by** (*metis order-trans parts-synth-parts synth-idem synth-parts-mono*)

lemma *synth-parts-trans-elem*:
assumes *x* ∈ *A* **and** *A* ⊆ *synth* (*parts B*) **and** *B* ⊆ *synth* (*parts C*)
shows *x* ∈ *synth* (*parts C*)
using *synth-parts-trans assms* **by** *auto*

lemma *synth-un-parts-split*:
assumes *x* ∈ *synth* (*parts A* ∪ *parts B*)
and ⋀*x* . *x* ∈ *A* ⟹ *x* ∈ *synth* (*parts C*)
and ⋀*x* . *x* ∈ *B* ⟹ *x* ∈ *synth* (*parts C*)
shows *x* ∈ *synth* (*parts C*)
proof −
have *parts A* ⊆ *synth* (*parts C*) *parts B* ⊆ *synth* (*parts C*)
using *assms*(*2*) *assms*(*3*) *synth-parts-subset* **by** *blast+*
then have *x* ∈ *synth* (*synth* (*parts C*) ∪ *synth* (*parts C*)) **using** *assms*(*1*)
using *synth-trans* **by** *auto*

then show *?thesis* **by** *auto*
qed

Normalization of Messages

Prevent FS from being contained directly in other FS. For instance, a term *FS* {|*FS* {|*Num 0*|}, *Num 0*|} is not normalized, whereas *FS* {|*Hash* (*FS* {|*Num 0*|}), *Num 0*|} is normalized.

inductive *normalized* :: *msgterm* ⇒ *bool* **where**
  ε [*simp*,*intro*!]: *normalized ε*
| *AS* [*simp*,*intro*!]: *normalized* (*AS agt*)
| *Num* [*simp*,*intro*!]: *normalized* (*Num n*)
| *Key* [*simp*,*intro*!]: *normalized* (*Key n*)
| *Nonce* [*simp*,*intro*!]: *normalized* (*Nonce n*)
| *Lst* [*intro*]: ⟦ ⋀*x* . *x* ∈ *set xs* ⟹ *normalized x* ⟧ ⟹ *normalized* (*L xs*)
| *FSt* [*intro*]: ⟦ ⋀*x* . *x* ∈ *fset xs* ⟹ *normalized x*;
  ⋀*x ys* . *x* ∈ *fset xs* ⟹ *x* ≠ *FS ys* ⟧
  ⟹ *normalized* (*FS xs*)
| *Hash* [*intro*]: *normalized X* ⟹ *normalized* (*Hash X*)
| *MPair* [*intro*]: ⟦ *normalized X*; *normalized Y* ⟧ ⟹ *normalized* ⟨*X*,*Y*⟩
| *Crypt* [*intro*]: ⟦ *normalized X* ⟧ ⟹ *normalized* (*Crypt K X*)

thm *normalized.simps*
find-theorems *normalized*

Examples

lemma *normalized* (*FS* {| *Hash* (*FS* {| *Num 0* |}), *Num 0* |}) **by** *fastforce*
lemma ¬ *normalized* (*FS* {| *FS* {| *Num 0* |}, *Num 0* |}) **by** (*auto elim*: *normalized.cases*)

Closure of *normalized* under *parts*, *analz* and *synth*

All synthesized terms are normalized (since *synth* prevents directly nested FSets).

lemma *normalized-synth*[*elim*!]: ⟦*t* ∈ *synth H*; ⋀*t*. *t* ∈ *H* ⟹ *normalized t*⟧ ⟹ *normalized t*
  by(*induction t*, *auto 3 4*)

lemma *normalized-parts*[*elim*!]: ⟦*t* ∈ *parts H*; ⋀*t*. *t* ∈ *H* ⟹ *normalized t*⟧ ⟹ *normalized t*
  by(*induction t rule*: *parts.induct*)
  (*auto elim*: *normalized.cases*)

lemma *normalized-analz*[*elim*!]: ⟦*t* ∈ *analz H*; ⋀*t*. *t* ∈ *H* ⟹ *normalized t*⟧ ⟹ *normalized t*
  by(*induction t rule*: *analz.induct*)
  (*auto elim*: *normalized.cases*)

Properties of *normalized*

lemma *normalized-FS*[*elim*]: ⟦*normalized* (*FS xs*); *x* |∈| *xs*⟧ ⟹ *normalized x*
  by(*auto simp add*: *normalized.simps*[*of FS xs*])

lemma *normalized-FS-FS*[*elim*]: ⟦*normalized* (*FS xs*); *x* |∈| *xs*; *x* = *FS ys*⟧ ⟹ *False*
  by(*auto simp add*: *normalized.simps*[*of FS xs*])

lemma *normalized-subset*: ⟦*normalized* (*FS xs*); *ys* |⊆| *xs*⟧ ⟹ *normalized* (*FS ys*)

by (*auto intro*!: *normalized.FSt*)

lemma *normalized-insert*[*elim*!]: *normalized* (*FS* (*finsert x xs*)) $\Longrightarrow$ *normalized* (*FS xs*)
by(*auto elim*!: *normalized-subset*)

lemma *normalized-union*:
assumes *normalized* (*FS xs*) *normalized* (*FS ys*) *zs* |⊆| *xs* |∪| *ys*
shows *normalized* (*FS zs*)
using *assms* **by**(*auto intro*!: *normalized.FSt*)

lemma *normalized-minus*[*elim*]:
assumes *normalized* (*FS* (*ys* |−| *xs*)) *normalized* (*FS xs*)
shows *normalized* (*FS ys*)
using *normalized-union assms* **by** *blast*

Lemmas that do not use *normalized*, but are helpful in proving its properties

lemma *FS-mono*: ⟦*zs-s* = *finsert* (*f* (*FS zs-s*)) *zs-b*; ⋀ *x*. *size* (*f x*) > *size x*⟧ $\Longrightarrow$ *False*
by (*metis* (*no-types*) *add.right-neutral add-Suc-right finite-fset finsert.rep-eq less-add-Suc1 msgterm.size*(*17*) *not-less-eq size-fset-simps sum.insert-remove*)

lemma *FS-contr*: ⟦*zs* = *f* (*FS* {|*zs*|}); ⋀ *x*. *size* (*f x*) > *size x*⟧ $\Longrightarrow$ *False*
using *FS-mono* **by** *blast*

end

1.5 Tools

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

1.5.1 Prefixes, suffixes, and fragments

thm *Cons-eq-appendI*
lemma *prefix-cons*: $[\![$*prefix xs ys*; *zs* = *x* # *ys*; *prefix xs'* (*x* # *xs*)$]\!] \Longrightarrow$ *prefix xs' zs*
  by (*auto simp add*: *prefix-def Cons-eq-appendI*)

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

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

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

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

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

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

1.5.2 Fragments

definition *fragment* :: *'a list* ⇒ *'a list set* ⇒ *bool*
  where *fragment xs St* ⟷ (∃ *zs1 zs2*. *zs1* @ *xs* @ *zs2* ∈ *St*)

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

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

lemma *fragment-Nil* [*simp*]: *fragment* [] *St* ⟷ *St* ≠ {}
by (*force simp add*: *fragment-def dest*: *spec* [**where** *x*=[]])

lemma *fragment-subset*: $[\![$*St* ⊆ *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* ∈ *St*⟧ $\Longrightarrow$ *fragment l St*
by(*auto simp add*: *fragment-def intro*!: *exI* [**where** *x*=[]])

lemma *fragment-prefix-self* [*simp*, *intro*]:
⟦*l* ∈ *St*; *prefix l′ l*⟧ $\Longrightarrow$ *fragment l′ St*
using *fragment-prefix fragment-self* **by** *blast*

lemma *fragment-suffix-self* [*simp*, *intro*]:
⟦*l* ∈ *St*; *suffix l′ l*⟧ $\Longrightarrow$ *fragment l′ St*
using *fragment-suffix fragment-self* **by** *metis*

lemma *fragment-is-prefix-suffix*:
fragment l St $\Longrightarrow$ ∃ *pre suff* . *prefix l pre* ∧ *suffix pre suff* ∧ *suff* ∈ *St*
by (*meson fragment-def prefixI suffixI*)

1.5.3 Pair Fragments

definition *pfragment* :: ′*a* ⇒ (′*b list*) ⇒ (′*a* × (′*b list*)) *set* ⇒ *bool*
where *pfragment a xs St* ⟷ (∃ *zs1 zs2*. (*a*, *zs1* @ *xs* @ *zs2*) ∈ *St*)

lemma *pfragmentI*: ⟦ (*ainf*, *zs1* @ *xs* @ *zs2*) ∈ *St* ⟧ $\Longrightarrow$ *pfragment ainf xs St*
by (*auto simp add*: *pfragment-def*)

lemma *pfragmentE* [*elim*]: ⟦*pfragment ainf xs St*; ⋀*zs1 zs2*. ⟦ (*ainf*, *zs1* @ *xs* @ *zs2*) ∈ *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*) ∈ *St*⟧ $\Longrightarrow$ *pfragment ainf l St*
by(*auto simp add*: *pfragment-def intro*!: *exI* [**where** *x*=[]])

lemma *pfragment-suffix-self* [*simp*, *intro*]:
⟦(*ainf*, *l*) ∈ *St*; *suffix l′ l*⟧ $\Longrightarrow$ *pfragment ainf l′ St*
using *pfragment-suffix pfragment-self* **by** *metis*

lemma *pfragment-self-eq*:
⟦*pfragment ainf l S*; ⋀*zs1 zs2* . (*ainf*, *zs1*@*l*@*zs2*) ∈ *S* $\Longrightarrow$ (*ainf*, *zs1*@*l′*@*zs2*) ∈ *S*⟧ $\Longrightarrow$ *pfragment*

ainf l′ S
  by(*auto simp add*: *pfragment-def*)

lemma *pfragment-self-eq-nil*:
⟦*pfragment ainf l S*; ⋀*zs1 zs2* . (*ainf*, *zs1*@*l*@*zs2*) ∈ *S* ⟹ (*ainf*, *l′*@*zs2*) ∈ *S*⟧ ⟹ *pfragment ainf l′ S*
  apply(*auto simp add*: *pfragment-def*)
  apply(*rule exI*[*of* - []])
  by *auto*

lemma *pfragment-cons*: *pfragment ainfo* (*x* # *fut*) *S* ⟹ *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* ≠ [] ⟹ *head xs* = *Some* (*hd xs*) **by**(*cases xs*, *auto*)
lemma *tail-cons*: *xs* ≠ [] ⟹ *tail xs* = *Some* (*last xs*) **by**(*cases xs*, *auto*)
lemma *tail-snoc*: *tail* (*xs* @ [*x*]) = *Some x* **by**(*cases xs*, *auto*)
lemma ∀ *y ys* . *l* ≠ *ys* @ [*y*] ⟹ *l* = []
  using *rev-exhaust* **by** *blast*

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

end

theory *Take-While* **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.

The predicate P gets both a predecessor and a successor (if existant). We originally used this theory for both the interface check (which makes use of the predecessor) and the cryptographic check (which makes use of the successor). However, with the introduction of mutable uinfo fields, we have split up the takeWhile formalization for the cryptographic check into a separate theory (*Take-While-Update*). Since the interface check does not make use of the successor, the third parameter of the function P defined in this theory is not actually required.

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*: $t \in set$ (*TW.takeW P pre l nxt*) $\Longrightarrow t \in set\ l$
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*⟧
⟹ *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*))⟧
⟹ *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*⟧ ⟹ ∃ *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*⟧ ⟹ ∃ *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*⟧ ⟹ ∃ *pre'*. *holds pre' l' nxt*
by (*metis holds-append suffix-def*)

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

end
end
theory *Take-While-Update* **imports** *Tools*
begin

1.7 Extending *Take-While* with an additional, mutable parameter

This theory defines takeW, holds and extract similarly to the other *Take-While* theory, but removes the predecessor parameter and adds a parameter to P and an update function that is applied to this parameter. In our formalization, the additional parameter is the uinfo field and the update function is the update on uinfo fields.

locale *TWu* =
  fixes *P* :: (*'b* ⇒ *'a* ⇒ *'a option* ⇒ *bool*)
  fixes *upd* :: (*'b* ⇒ *'a* ⇒ *'b*)
begin

1.7.1 Definitions

Apply *upds* on a sequence

abbreviation *upds* :: *'b* ⇒ *'a list* ⇒ *'b* **where**
  upds ≡ *foldl upd*

fun *upd-opt* :: (*'b* ⇒ *'a option* ⇒ *'b*) **where**
  upd-opt info (*Some hf*) = *upd info hf*
| *upd-opt info None* = *info*

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

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

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

function *takeW* :: *'b* ⇒ *'a list* ⇒ *'a option* ⇒ *'a list* **where**
  takeW - [] - = []
| *P info x xo* ⟹ *takeW info* [*x*] *xo* = [*x*]
| ¬ *P info x xo* ⟹ *takeW info* [*x*] *xo* = []
| *P info x* (*Some y*) ⟹ *takeW info* (*x* # *y* # *xs*) *xo* = *x* # *takeW* (*upd info y*) (*y* # *xs*) *xo*
| ¬ *P info x* (*Some y*) ⟹ *takeW info* (*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* :: *'b* ⇒ *'a list* ⇒ *'a option* ⇒ *'a option*
where
  extract info (*x* # *y* # *ys*) *nxt* = (*if P info x* (*Some y*) *then extract* (*upd info y*) (*y* # *ys*) *nxt else Some x*)
| *extract info* [*x*] *nxt* = (*if P info x nxt then nxt else* (*Some x*))
| *extract info* [] *nxt* = *nxt*

1.7.2 Lemmas

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

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

lemma *takeW-two-or-more*:
  takeW info (*x* # *y* # *zs*) *xo* = (*if P info x* (*Some y*) *then x* # *takeW* (*upd info y*) (*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 info (*x* # *xs*) *nxt* =
    (*if xs* = []
     then (*if P info x nxt then* [*x*] *else* [])
     else (*if P info x* (*Some* (*hd xs*)) *then x* # *takeW* (*upd info* (*hd xs*)) *xs nxt else* []))
by (*cases xs*, *auto*)

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

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

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

lemma *holds-Cons-holds*:
  holds info (*x* # *xs*) *nxt* ⟹ *holds* (*upd-opt info* (*head xs*)) *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 info* (*takeW info xs nxt*) (*extract info xs nxt*)
apply(*induction info xs nxt rule*: *takeW.induct*)

apply *auto*
subgoal for *info x y ys*
  apply(*cases ys*)
  apply(*simp-all*)
  done
done

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

lemma *holds-append*:
  holds info (*xs* @ *ys*) *nxt* =
   (*case ys of* [] ⇒ *holds info xs nxt* | *x* # - ⇒
     holds info xs (*Some x*) ∧
     (*case xs of* [] ⇒ *holds info ys nxt*
              | - ⇒ *holds* (*upds info* (*tl xs*@[*x*])) *ys nxt*))
  by(*induction info xs nxt rule*: *takeW.induct*)
   (*auto split*: *list.split*)

lemma *upds-snoc*: *upds uinfo* (*xs*@[*x*]) = *upd* (*upds uinfo xs*) *x*
  by *simp*

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

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

lemma *holds-implies-takeW-is-identity*:
  holds info l nxt ⟹ *takeW info l nxt* = *l*
by (*induction info l nxt rule*: *takeW.induct*) *auto*

lemma *holds-takeW-is-identity*[*simp*]:
  takeW info l nxt = *l* ⟷ *holds info l nxt*
  by (*induction info l nxt rule*: *takeW.induct*) *auto*

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

Holds unfolding

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

lemma *holds-set-list*: ⟦*holds info l nxt*; *x* ∈ *set l*⟧ ⟹ ∃ *p y* . *P p x y*
  apply(*induction info l nxt rule*: *TWu.takeW.induct*[**where** *?Pa*=*P*]) **by** *auto*

lemma *holds-set-list-no-update*: ⟦*holds info l nxt*; *x* ∈ *set l*; ⋀*a b*. *upd a b* = *a*⟧ ⟹ ∃ *y* . *P info x y*

apply(*induction info l nxt rule*: *TWu.takeW.induct*[**where** *?Pa=P*]) **by** *auto*

lemma *holds-unfold*: *holds info l None* $\Longrightarrow$
$l = [] \vee$
$(\exists\ x\ .\ l = [x] \wedge P\ info\ x\ None) \vee$
$(\exists\ x\ y\ ys\ .\ l = (x\#y\#ys) \wedge P\ info\ x\ (Some\ y) \wedge holds\ (upd\ info\ y)\ (y\#ys)\ None)$
by *auto* (*meson holds.elims*(*2*))

lemma *holds-unfold-prexnxt'*:
$[\![holds\ info\ l\ nxt;\ l = (zs@(x1\#x2\#xs));\ zs \neq []]\!]$
$\Longrightarrow P\ (upds\ info\ ((tl\ zs)@[x1]))\ x1\ (Some\ x2)$
apply(*cases zs*) **apply** *simp*
apply(*simp only*: *TWu.holds-append*)
by *auto*

lemma *holds-suffix*:
$[\![holds\ info\ l\ nxt;\ suffix\ l'\ l]\!] \Longrightarrow \exists\ info'.\ holds\ info'\ l'\ nxt$
apply(*cases l'*)
by(*auto simp add*: *suffix-def TWu.holds-append list.case-eq-if*)

lemma *holds-unfold-prelnil*:
$[\![holds\ info\ l\ nxt;\ l = (zs@(x1\#[]));\ zs \neq []]\!]$
$\Longrightarrow P\ (upds\ info\ ((tl\ zs)@[x1]))\ x1\ nxt$
apply(*cases zs*)
subgoal by *simp*
by(*simp only*: *TWu.holds-append*) *auto*

Update shifted

Usually, the update has already been applied to the head of the list. Hence, when given a list to apply updates to (and a successor, i.e., the first element that comes after the list), we remove the first element of the list and add the successor. We apply the updates on the resulting list.

fun *upd-shifted* :: $('b \Rightarrow 'a\ list \Rightarrow 'a \Rightarrow 'b)$ **where**
upd-shifted uinfo $(x\#xs)$ *nxt* = *upds uinfo* $(xs@[nxt])$
| *upd-shifted uinfo* [] *nxt* = *uinfo*

This lemma is useful when there is an intermediate hop field hf of interest.

lemma *holds-intermediate*:
assumes *holds uinfo p nxt p* = *pre* @ *hf* # *post*
shows *holds* (*upd-shifted uinfo pre hf*) (*hf* # *post*) *nxt*
using *assms* **proof**(*induction pre arbitrary*: *p uinfo hf*)
case *Nil*
then show *?case* **using** *assms* **by** *auto*
next
case *induct-asm*: (*Cons a prev*)
show *?case*
proof(*cases prev*)
case *Nil*
then have *holds* (*upd uinfo hf*) (*hf* # *post*) *nxt*
using *induct-asm* **by** *simp*
then show *?thesis*

using *induct-asm Nil* **by** *auto*
next
case *cons-asm*: (*Cons b list*)
then have *holds* (*upd uinfo b*) (*b* # *list* @ *hf* # *post*) *nxt*
using *induct-asm*(*2*−*3*) **by** *auto*
then show *?thesis*
using *induct-asm*(*1*)
by (*simp add*: *cons-asm*)
qed
qed

lemma *holds-intermediate-ex*:
assumes *holds uinfo hfs nxt hf* ∈ *set hfs*
shows ∃ *pre post* . *holds* (*upd-shifted uinfo pre hf*) (*hf* # *post*) *nxt* ∧ *hfs* = *pre* @ *hf* # *post*
using *assms holds-intermediate*
by (*meson split-list*)

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/Take-While
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 :: ($'ainfo \times 'aahi$ *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
  $\textit{dp0-add-loc}\ s\ s'\ asid\ pkt \equiv s' = s(\!|loc := (loc\ s)(asid := loc\ s\ asid \cup \{pkt\})|\!)$

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**
  $\textit{dp0-add-chan}\ s\ s'\ a1\ i1\ pkt \equiv$
    $\exists\, a2\ i2\ .\ \textit{rev-link}\ a1\ i1 = (Some\ a2,\ Some\ i2) \wedge$
    $s' = s(\!|chan := (chan\ s)((a1,\ i1,\ a2,\ i2) := chan\ s\ (a1,\ i1,\ a2,\ i2) \cup \{pkt\})|\!)$

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

definition *dp0-in-chan* :: ('aahi, 'ainfo) *dp0-state* ⇒ *as* ⇒ *ifs* ⇒ ('aahi, 'ainfo) *pkt0* ⇒ *bool* **where**
dp0-in-chan s a1 i1 pkt ≡
∃ *a2 i2* . *rev-link a1 i1* = (*Some a2*, *Some i2*) ∧ *pkt* ∈ (*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' ≡
— 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 fut auth-seg0 ∧
— 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' ≡
— 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* |) ∧
dp0-in-chan s asid downif m ∧

ASID hf1 = *asid* ∧

— action: Update state to include message
dp0-add-loc s s' asid (|
AInfo = *ainfo*,
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* ⦈ ∧
m ∈ (*loc s*) *asid* ∧
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* ⦈ ∧
ASID hf1 = *asid* ∧
m ∈ (*loc s*) *asid* ∧
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-dispatch-ext s m asid ainfo upif pas fut hist s' ≡

$m = (\!| \; AInfo = ainfo, \; past = pas, \; future = fut, \; history = hist \; |\!) \wedge$
$hist = [] \wedge$

pfragment ainfo fut auth-seg0 ∧

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

2.2.2 Transition system

fun *dp0-trans* **where**
dp0-trans s (*evt-dispatch-int0 asid m*) *s′* ⟷
(∃ *ainfo pas fut hist*. *dp0-dispatch-int s m ainfo asid pas fut hist s′*) |
dp0-trans s (*evt-recv0 asid downif m*) *s′* ⟷
(∃ *ainfo hf1 pas fut hist*. *dp0-recv s m asid ainfo hf1 downif pas fut hist s′*) |
dp0-trans s (*evt-send0 asid upif m*) *s′* ⟷
(∃ *ainfo hf1 pas fut hist*. *dp0-send s m asid ainfo hf1 upif pas fut hist s′*) |
dp0-trans s (*evt-deliver0 asid m*) *s′* ⟷
(∃ *ainfo hf1 pas fut hist*. *dp0-deliver s m asid ainfo hf1 pas fut hist s′*) |
dp0-trans s (*evt-dispatch-ext0 asid upif m*) *s′* ⟷
(∃ *ainfo pas fut hist*. *dp0-dispatch-ext s m asid ainfo upif pas fut hist s′*) |
dp0-trans s (*evt-observe0 s″*) *s′* ⟷ *s* = *s′* ∧ *s* = *s″* |
dp0-trans s evt-skip0 s′ ⟷ *s* = *s′*

definition *dp0-init* :: (′*aahi*, ′*ainfo*) *dp0-state* **where**
dp0-init ≡ (|*chan* = (λ-. {}), *loc* = (λ-. {})|)

definition *dp0* :: ((′*aahi*, ′*ainfo*) *evt0*, (′*aahi*, ′*ainfo*) *dp0-state*) *ES* **where**
dp0 ≡ (|
init = (=) *dp0-init*,
trans = *dp0-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* ≡ ∃ *x*. *m* ∈ (*loc s*) *x* ∨ (∃ *x*. *m* ∈ (*chan s*) *x*)

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

lemma *dp0-add-chan-msgs*:
assumes *dp0-add-chan s s′ asid upif m* **and** *soup n s′* **and** *n* ≠ *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, makes this invariant inductive. 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*→ *s'*; *inv-auth s*⟧ ⟹ *inv-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* ≡
$\{\tau \mid \tau\ .\ \forall\ s\ .\ evt\text{-}observe0\ s \in set\ \tau \longrightarrow inv\text{-}auth\ s\}$

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

Easier to read

definition *inv-authorized* :: (*′aahi*, *′ainfo*) *dp0-state* ⇒ *bool* **where**
inv-authorized s ≡ ∀ *m* . *soup m s* ⟶
(∃ *timestamp auth-path*. (*timestamp*, *auth-path*) ∈ *auth-seg0* ∧
(∃ *pre post*. *auth-path* = *pre* @ (*rev* (*history m*)) @ *post*))

lemma *inv-auth s* ⟹ *inv-authorized s*
apply (*auto simp add*: *inv-authorized-def inv-auth-def*)
by (*metis auth-path-def pfragment-def pfragment-prefix*)+

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**
inv-detect s ≡ ∀ *m* . *soup m s* ⟶ *prefix* (*history m*) (*past m*)

lemma *inv-detectI*:
assumes ⋀*m x* . *soup m s* ⟹ *prefix* (*history m*) (*past m*)
shows *inv-detect s*
using *assms* **by**(*auto simp add*: *inv-detect-def*)

lemma *inv-detectD*:
assumes *inv-detect s*
shows ⋀*m x* .*m* ∈ (*loc s*) *x* ⟹ *prefix* (*history m*) (*past m*)
and ⋀*m x* .*m* ∈ (*chan s*) *x* ⟹ *prefix* (*history m*) (*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*→ *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* ≡ {τ | τ . ∀ *s* . *evt-observe0 s* ∈ *set* τ ⟶ *inv-detect s*}

lemma *tr0-satisfies-detectability*: *dp0* $\models_{ES}$ *TR-detect*
  using *Inv-inv-detect*
  by(*intro trace-property-rule*[**where** *?I*=λτ *s*. τ ∈ *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' ≡
m = (| *AInfo* = *ainfo*, *past* = *pas*, *future* = *fut*, *history* = *hist* |) ∧
hist = [] ∧
pfragment ainfo (*ifs-valid-prefix* (*prev-hf asid upif*) *fut None*) *auth-seg0* ∧

— 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' ≡
DownIF hf1 = *Some downif*
∧ *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'* ⟷
(∃ *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'* ⟷
(∃ *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'* ⟷
(∃ *ainfo hf1 pas fut hist*. *dp1-recv s m asid ainfo hf1 downif pas fut hist s'*) |
dp1-trans s e s' ⟷ *dp0-trans s e s'*

definition *dp1-init* :: (*'aahi*, *'ainfo*) *dp1-state* **where**
dp1-init ≡ (|*chan* = (λ-. {}), *loc* = (λ-. {})|)

definition *dp1* :: ((*'aahi*, *'ainfo*) *evt1*, (*'aahi*, *'ainfo*) *dp1-state*) *ES* **where**
dp1 ≡ (|
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* ⇒ *ifs* ⇒ (*'aahi*, *'ainfo*) *pkt1* ⇒ (*'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* ⇒ (*'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* ⇒ (*'aahi*, *'ainfo*) *dp0-state* **where**
R10 s =
(|*chan* = λ(*a1*, *i1*, *a2*, *i2*) . (*pkt1to0chan a1 i1*) ‘ ((*chan s*) (*a1*, *i1*, *a2*, *i2*)),
loc = λ*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 ⊑_π_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.

Check if interface option is matched by a msgterm.

fun *ASIF* :: *ifs option* ⇒ *msgterm* ⇒ *bool* **where**
  ASIF (*Some a*) (*AS a′*) = (*a*=*a′*)
| *ASIF None* ε = *True*
| *ASIF* - - = *False*

lemma *ASIF-None*[*simp*]: *ASIF ifopt* ε ⟷ *ifopt* = *None* **by**(*cases ifopt*, *auto*)
lemma *ASIF-AS*[*simp*]: *ASIF ifopt* (*AS a*) ⟷ *ifopt* = *Some a* **by**(*cases ifopt*, *auto*)

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* ⇒ *ifs option* **where**
  term2if (*AS a*) = *Some a*
| *term2if* ε = *None*
| *term2if* - = *None*

lemma *ASIF-term2if*[*intro*]: *ASIF i mi* ⟹ *ASIF* (*term2if mi*) *mi*
  by(*cases mi*, *auto*)

fun *if2term* :: *ifs option* ⇒ *msgterm* **where** *if2term* (*Some a*) = *AS a* | *if2term None* = ε

lemma *if2term-eq*[*elim*]: *if2term a* = *if2term b* ⟹ *a* = *b*
  apply(*cases a*, *cases b*, *auto*)
  using *if2term.elims msgterm.distinct*(*1*)
  by (*metis term2if.simps*(*1*))

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

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

fun *term2hf* :: *msgterm* ⇒ *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*⟧ ⟹ *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-scheme
  UHI :: 'uhi
  HVF :: msgterm

record ('aahi, 'uinfo, 'uhi, 'ainfo) pkt2 =
  AInfo :: 'ainfo
  UInfo :: 'uinfo
  past  :: ('aahi, 'uhi) HF list
  future  :: ('aahi, 'uhi) HF list
  history  :: 'aahi ahi-scheme list
```

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

```
record ('aahi, 'uinfo, 'uhi, 'ainfo) dp2-state =
  chan2 :: (as × ifs × as × ifs) ⇒ ('aahi, 'uinfo, 'uhi, 'ainfo) pkt2 set
  loc2 :: as ⇒ ('aahi, 'uinfo, 'uhi, 'ainfo) pkt2 set

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

definition soup2 where soup2 m s ≡ ∃x. m ∈ (loc2 s) x ∨ (∃x. m ∈ (chan2 s) x)

declare soup2-def [simp]

fun fwd-pkt :: ('aahi, 'uinfo, 'uhi, 'ainfo) pkt2 ⇒ ('aahi, 'uinfo, 'uhi, 'ainfo) pkt2 where
  fwd-pkt (| AInfo = ainfo, UInfo = uinfo, past = pas, future = hf1#fut, history = hist |)
      = (| AInfo = ainfo, UInfo = uinfo, past = hf1#pas, future = fut, history = (AHI hf1)#hist |)
```

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* × *'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* ⇒ *'uinfo*
⇒ (*'aahi*, *'uhi*) *HF list*
⇒ (*'aahi*, *'uhi*) *HF option*
⇒ (*'aahi*, *'uhi*) *HF*
⇒ (*'aahi*, *'uhi*) *HF option* ⇒ *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 ⇒ *'uinfo*
⇒ (*'aahi*, *'uhi*) *HF list*
⇒ (*'aahi*, *'uhi*) *HF option*
⇒ (*'aahi*, *'uhi*) *HF list*
⇒ (*'aahi*, *'uhi*) *HF option* ⇒ (*'aahi*, *'uhi*) *HF list*
— We need *auth-restrict* to further restrict the set of authorized segments. For instance, we need it for the empty segment (ainfo, []) since according to the definition any such ainfo will be contained in the intruder knowledge. With *auth-restrict* we can restrict this.
and *auth-restrict* :: *'ainfo* ⇒ *'uinfo* ⇒ (*'aahi*, *'uhi*) *HF list* ⇒ *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*
— *term-ainfo* extracts what msgterms the intruder can learn from analyzing a given authenticated info field.
and *term-ainfo* :: *'ainfo* ⇒ *msgterm*
— *terms-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 *terms-hf* :: (*'aahi*, *'uhi*) *HF* ⇒ *msgterm set*
— *terms-uinfo* extracts what msgterms the intruder can learn from analyzing a given uinfo field.
and *terms-uinfo* :: *'uinfo* ⇒ *msgterm set*
— *upd-uinfo* takes a uinfo field an a hop field and returns the updated uinfo field.
and *upd-uinfo* :: *'uinfo* ⇒ (*'aahi*, *'uhi*) *HF* ⇒ *'uinfo*
— As *ik-oracle* (defined below) 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* ⇒ *'uinfo* ⇒ *bool*

begin

Auxiliary definitions and lemmas

Define uinfo field updates.

fun *upd-uinfo-pkt* :: ('*aahi*, '*uinfo*, '*uhi*, '*ainfo*) *pkt2* ⇒ '*uinfo* **where**
upd-uinfo-pkt (| *AInfo* = *ainfo*, *UInfo* = *uinfo*, *past* = *pas*, *future* = *hf1*#*fut*, *history* = *hist* |)
= *upd-uinfo uinfo hf1*
| *upd-uinfo-pkt* (| *AInfo* = *ainfo*, *UInfo* = *uinfo*, *past* = *pas*, *future* = [], *history* = *hist* |) = *uinfo*

definition *upd-pkt* :: ('*aahi*, '*uinfo*, '*uhi*, '*ainfo*) *pkt2* ⇒ ('*aahi*, '*uinfo*, '*uhi*, '*ainfo*) *pkt2* **where**
upd-pkt pkt = *pkt*(|*UInfo* := *upd-uinfo-pkt pkt*|)

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

definition *AHIS* :: ('*aahi*, '*uhi*) *HF list* ⇒ '*aahi ahi-scheme list* **where**
AHIS hfs ≡ *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 *pkt2to1loc* :: ('*aahi*, '*uinfo*, '*uhi*, '*ainfo*) *pkt2* ⇒ ('*aahi*, '*ainfo*) *pkt1* **where**
pkt2to1loc (| *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*|)

fun *pkt2to1chan* :: ('*aahi*, '*uinfo*, '*uhi*, '*ainfo*) *pkt2* ⇒ ('*aahi*, '*ainfo*) *pkt1* **where**
pkt2to1chan (| *AInfo* = *ainfo*, *UInfo* = *uinfo*, *past* = *pas*, *future* = *fut*, *history* = *hist* |) =
(| *pkt0.AInfo* = *ainfo*,
past = *AHIS pas*,
future = *AHIS* (*hfs-valid-prefix-generic ainfo*
(*upd-uinfo-pkt* (| *AInfo* = *ainfo*, *UInfo* = *uinfo*, *past* = *pas*, *future* = *fut*, *history* = *hist* |))
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* :: '*uinfo* ⇒ ('*ainfo* × ('*aahi*, '*uhi*) *HF list*) *set* **where**
auth-seg2 uinfo ≡ ({(*ainfo*, *l*) | *ainfo l* . *hfs-valid-prefix-generic ainfo uinfo* [] *None l None* = *l*
∧ *auth-restrict ainfo uinfo l*
∧ *no-oracle ainfo uinfo*
∧ (*ainfo*, *AHIS l*) ∈ *auth-seg0*})

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

lemma *pfragment-auth-seg20*:
pfragment ainfo l (*auth-seg2 uinfo*) ⟹ *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 uinfo*); *l'* = *AHIS l*⟧ ⟹ *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*, '*uinfo*, '*uhi*, '*ainfo*, '*more*) *dp2-state-scheme* ⇒
('*aahi*, '*uinfo*, '*uhi*, '*ainfo*, '*more*) *dp2-state-scheme* ⇒ *as* ⇒ ('*aahi*, '*uinfo*, '*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*, '*uinfo*, '*uhi*, '*ainfo*, '*more*) *dp2-state-scheme* ⇒ ('*aahi*, '*uinfo*, '*uhi*, '*ainfo*, '*more*) *dp2-state-scheme*
⇒ *as* ⇒ *ifs* ⇒ ('*aahi*, '*uinfo*, '*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*, '*uinfo*, '*uhi*, '*ainfo*, '*more*) *dp2-state-scheme* ⇒ *as* ⇒ *ifs* ⇒ ('*aahi*, '*uinfo*, '*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* - - - - - - - *upd-uinfo*
for *hf-valid-generic* :: '*ainfo* ⇒ '*uinfo*
⇒ ('*aahi*, '*uhi*) *HF list*
⇒ ('*aahi*, '*uhi*) *HF option*
⇒ ('*aahi*, '*uhi*) *HF*
⇒ ('*aahi*, '*uhi*) *HF option* ⇒ *bool*
and *upd-uinfo* :: '*uinfo* ⇒ ('*aahi*, '*uhi*) *HF* ⇒ '*uinfo* +
— *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*
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 but not the uinfo field (see below).

definition *ik-hfs* :: *msgterm set* **where**
ik-hfs = {*t* | *t hf hfs ainfo uinfo*. *t* ∈ *terms-hf hf* ∧ *hf* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ (*auth-seg2 uinfo*)}

This set should contain all terms that can be learned from analyzing the uinfo field.

definition *ik-uinfo* :: *msgterm set* **where**
ik-uinfo = {*t* | *ainfo hfs uinfo t*. *t* ∈ *terms-uinfo uinfo* ∧ (*ainfo*, *hfs*) ∈ (*auth-seg2 uinfo*)}

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

definition *ik* :: *msgterm set* **where**
ik = *ik-hfs*
∪ {*term-ainfo ainfo* | *ainfo hfs uinfo*. (*ainfo*, *hfs*) ∈ (*auth-seg2 uinfo*)}
∪ *ik-uinfo*
∪ *Key*‘(*macK*‘*bad*)
∪ *ik-add*
∪ *ik-oracle*

definition *terms-pkt* :: ('*aahi*, '*uinfo*, '*uhi*, '*ainfo*) *pkt2* ⇒ *msgterm set* **where**

$terms\text{-}pkt\ m \equiv \{t \mid t\ hf.\ t \in terms\text{-}hf\ hf \land hf \in set\ (past\ m) \cup set\ (future\ m)\}$
$\cup \{term\text{-}ainfo\ ainfo \mid ainfo\ .\ ainfo = AInfo\ m\}$
$\cup \bigcup\{terms\text{-}uinfo\ uinfo \mid uinfo\ .\ uinfo = UInfo\ m\}$

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

Different way of presenting the intruder knowledge

definition *ik-dynamic* :: (*'aahi*, *'uinfo*, *'uhi*, *'ainfo*, *'more*) *dp2-state-scheme* ⇒ *msgterm set* **where**
$ik\text{-}dynamic\ s \equiv ik \cup (\bigcup\{terms\text{-}pkt\ m \mid m\ .\ soup2\ m\ s\})$

lemma $ik\text{-}dynamic\ s = ik\text{-}dyn\ s$
apply(*auto simp add*: *ik-dyn-def ik-dynamic-def*)
by *metis+*

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*]:
$(ainfo,\ hfs) \in (auth\text{-}seg2\ uinfo) \Longrightarrow term\text{-}ainfo\ ainfo \in synth\ (analz\ ik)$
by(*auto simp add*: *ik-def*)*blast*

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

2.4.3 Events

This is an attacker event.

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
$dp2\text{-}dispatch\text{-}int\ s\ m\ ainfo\ uinfo\ asid\ pas\ fut\ hist\ s' \equiv$
$m = (\!|\ AInfo = ainfo,\ UInfo = uinfo,\ past = pas,\ future = fut,\ history = hist\ |\!) \land$
$hist = [] \land$
$terms\text{-}pkt\ m \subseteq synth\ (analz\ (ik\text{-}dyn\ s)) \land$
$no\text{-}oracle\ ainfo\ uinfo \land$
— action: Update the state to include m
$dp2\text{-}add\text{-}loc2\ s\ s'\ asid\ m$

definition
dp2-recv
where

dp2-recv s m asid ainfo uinfo hf1 downif pas fut hist s' ≡
— 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-in-chan2 s (*ASID* (*AHI hf1*)) *downif m* ∧
DownIF (*AHI hf1*) = *Some downif* ∧
ASID (*AHI hf1*) = *asid* ∧
hf-valid-generic ainfo (*upd-uinfo uinfo hf1*) (*rev*(*pas*)@*hf1*#*fut*) (*head pas*) *hf1* (*head fut*) ∧

— action: Update local state to include message
dp2-add-loc2 s s' asid (*upd-pkt m*)

definition
dp2-send
where
dp2-send s m asid ainfo uinfo hf1 upif pas fut hist s' ≡
— 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 ∈ (*loc2 s*) *asid* ∧
UpIF (*AHI hf1*) = *Some upif* ∧
ASID (*AHI hf1*) = *asid* ∧
hf-valid-generic ainfo uinfo (*rev*(*pas*)@*hf1*#*fut*) (*head pas*) *hf1* (*head fut*) ∧

— action: Update state to include modified message
dp2-add-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' ≡
$m = (\!| \; AInfo = ainfo,\ UInfo = uinfo,\ past = pas,\ future = hf1\#fut,\ history = hist \; |\!) \wedge$
m ∈ (*loc2 s*) *asid* ∧
ASID (*AHI hf1*) = *asid* ∧
fut = [] ∧
hf-valid-generic ainfo uinfo (*rev*(*pas*)@*hf1*#*fut*) (*head pas*) *hf1* (*head fut*) ∧

— 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.

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′ ≡
    m = ⦇ *AInfo* = *ainfo*, *UInfo* = *uinfo*, *past* = *pas*, *future* = *fut*, *history* = *hist* ⦈ ∧
    asid ∈ *bad* ∧
    hist = [] ∧
    terms-pkt m ⊆ *synth* (*analz* (*ik-dyn s*)) ∧
    no-oracle ainfo uinfo ∧

    — 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′* ⟷
    (∃ *ainfo uinfo pas fut hist* . *dp2-dispatch-int s m ainfo uinfo asid pas fut hist s′*) |
  dp2-trans s (*evt-recv2 asid downif m*) *s′* ⟷
    (∃ *ainfo uinfo hf1 pas fut hist* . *dp2-recv s m asid ainfo uinfo hf1 downif pas fut hist s′*) |
  dp2-trans s (*evt-send2 asid upif m*) *s′* ⟷
    (∃ *ainfo uinfo hf1 pas fut hist*. *dp2-send s m asid ainfo uinfo hf1 upif pas fut hist s′*) |
  dp2-trans s (*evt-deliver2 asid m*) *s′* ⟷
    (∃ *ainfo uinfo hf1 pas fut hist*. *dp2-deliver s m asid ainfo uinfo hf1 pas fut hist s′*) |
  dp2-trans s (*evt-dispatch-ext2 asid upif m*) *s′* ⟷
    (∃ *ainfo uinfo pas fut hist* . *dp2-dispatch-ext s m asid ainfo uinfo upif pas fut hist s′*) |
  dp2-trans s (*evt-observe2 s′′*) *s′* ⟷ *s* = *s′* ∧ *s* = *s′′* |
  dp2-trans s evt-skip2 s′ ⟷ *s* = *s′*

definition *dp2-init* :: (*′aahi*, *′uinfo*, *′uhi*, *′ainfo*) *dp2-state* **where**
  dp2-init ≡ ⦇*chan2* = (λ-. {}), *loc2* = (λ-. {})⦈

definition *dp2* :: ((*′aahi*, *′uinfo*, *′uhi*, *′ainfo*) *evt2*, (*′aahi*, *′uinfo*, *′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.

print-locale *dataplane-2-ik-defs*
locale *dataplane-2* = *dataplane-2-ik-defs* - - - - - - - - - - - - *hf-valid-generic upd-uinfo* - -
  for *hf-valid-generic* :: *′ainfo* ⇒ *′uinfo*
    ⇒ (*′aahi*, *′uhi*) *HF list*

⇒ (′aahi, ′uhi) *HF option*
⇒ (′aahi, ′uhi) *HF*
⇒ (′aahi, ′uhi) *HF option* ⇒ *bool*
and *upd-uinfo* :: ′uinfo ⇒ (′aahi, ′uhi) *HF* ⇒ ′uinfo +

assumes *ik-seg-is-auth*:
⟦*terms-pkt m* ⊆ *synth* (*analz ik*);
future m = *hfs*; *AInfo m* = *ainfo*;
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
and *upd-uinfo-ik*:
⟦*terms-uinfo uinfo* ⊆ *synth* (*analz ik*); *terms-hf hf* ⊆ *synth* (*analz ik*)⟧
⟹ *terms-uinfo* (*upd-uinfo uinfo hf*) ⊆ *synth* (*analz ik*)

and *upd-uinfo-no-oracle*: *no-oracle ainfo uinfo* ⟹ *no-oracle ainfo* (*upd-uinfo uinfo fld*)

— We require that *hfs-valid-prefix-generic* behaves as expected, i.e., that it implements the check mentioned above.
and *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*;
m = (|*AInfo* = *ainfo*, *UInfo* = *uinfo*, *past* = *pas*, *future* = *hf1* # *fut*, *history* = *hist*|)⟧
⟹ *hfs-valid-prefix-generic ainfo uinfo pas* (*head pas*) (*hf1* # *fut*) *None* =
hf1 # (*hfs-valid-prefix-generic ainfo* (*upd-uinfo-pkt* (*fwd-pkt m*)) (*hf1*#*pas*) (*Some hf1*) *fut None*)
begin

2.4.6 Mapping dp2 state to dp1 state

definition *R21* :: (′aahi, ′uinfo, ′uhi, ′ainfo) *dp2-state* ⇒ (′aahi, ′ainfo) *dp1-state* **where**
R21 s = (|*chan* = λ*x* . *pkt2to1chan* ‘ ((*chan2 s*) *x*),
loc = λ*x* . *pkt2to1loc* ‘ ((*loc2 s*) *x*)|)

lemma *auth-seg2-pfragment*:
⟦*pfragment ainfo* (*hf* # *fut*) (*auth-seg2 uinfo*); *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*; *pkt2to1chan 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*; *pkt2to1loc pkt2* = *pkt0*; *P* = *pkt2to1loc* ‘ *loc2 s1 asid*⟧ ⟹
pkt0 ∈ *P*
by *blast*

lemma *dp2-add-loc20E*:

⟦*dp2-add-loc2 s1 s1′ asid p1*; *p0* = *pkt2to1loc p1*; *s0* = *R21 s1*; *s0′* = *R21 s1′*⟧
  ⟹ *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*:
⟦*dp2-add-chan2 s1 s1′ a1 i1 p1*; *p0* = *pkt2to1chan p1*; *s0* = *R21 s1*; *s0′* = *R21 s1′*⟧
  ⟹ *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*, *′uinfo*, *′uhi*, *′ainfo*) *dp2-state* ⇒ *bool* **where**
 inv-ik-dyn s ≡ *ik-dyn s* ⊆ *synth* (*analz ik*)

lemma *inv-ik-dynI*:
 assumes ⋀*t m x* . ⟦*t* ∈ *terms-pkt m*; *m* ∈ *loc2 s x*⟧ ⟹ *t* ∈ *synth* (*analz ik*)
 and ⋀*t m x* . ⟦*t* ∈ *terms-pkt m*; *m* ∈ *chan2 s x*⟧ ⟹ *t* ∈ *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 ⋀*t m x* . ⟦*m* ∈ *chan2 s x*; *t* ∈ *terms-pkt m*⟧ ⟹ *t* ∈ *synth* (*analz ik*)
  ⋀*t m x* . ⟦*m* ∈ *loc2 s x*; *t* ∈ *terms-pkt m*⟧ ⟹ *t* ∈ *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*!]:
 ⟦*dp2-add-loc2 s s′ asid m*; *inv-ik-dyn s*; *terms-pkt m* ⊆ *synth* (*analz ik*)⟧
  ⟹ *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*!]:
 ⟦*dp2-add-chan2 s s′ a1 i1 m*; *inv-ik-dyn s*; *terms-pkt m* ⊆ *synth* (*analz ik*)⟧
  ⟹ *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* ⊆ *synth* (*analz ik*) **by**(*auto simp add*: *ik-dyn-def inv-ik-dyn-def*)
 moreover have *ik* ⊆ *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 *terms-pkt-upd*:
 ⟦*x* ∈ *terms-pkt* (*upd-pkt p*); ⋀*x*. *x* ∈ *terms-pkt p* ⟹ *x* ∈ *synth* (*analz ik*)⟧ ⟹ *x* ∈ *synth* (*analz ik*)
 apply(*cases p*)

subgoal for *AInfo UInfo past future history*
by(*cases future*)
(*auto simp add*: *upd-pkt-def terms-pkt-def elim*!: *upd-uinfo-ik*[*THEN subsetD*, *rotated 2*])
done

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*)
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*: *terms-pkt-upd dest*: *inv-ik-dynD*(*1*))
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*)
qed(*auto simp add*: *dp2-defs terms-pkt-def elim*!: *inv-ik-dynE*)
qed

Attacker dispatch events also capture honest dispatchers

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*:
assumes *pfragment ainfo fut* (*auth-seg2 uinfo*) *past m* = [] *AInfo m* = *ainfo UInfo m* = *uinfo future m* = *fut*
shows *terms-pkt m* $\subseteq$ *synth* (*analz* (*ik-dyn s*))
proof−
from *assms* **have** *terms-pkt m* $\subseteq$ *ik*
by (*cases m*)
(*auto 3 4 simp add*: *terms-pkt-def ik-def*)
then show *?thesis* **by** (*auto simp add*: *ik-dyn-def*)
qed

2.4.8 Refinement proof

fun π_2 :: (*'aahi*, *'uinfo*, *'uhi*, *'ainfo*) *evt2* ⇒ (*'aahi*, *'ainfo*) *evt0* **where**
π_2 (*evt-dispatch-int2 asid m*) = *evt-dispatch-int0 asid* (*pkt2to1loc m*)
| π_2 (*evt-recv2 asid downif m*) = *evt-recv0 asid downif* (*pkt2to1chan m*)
| π_2 (*evt-send2 asid upif m*) = *evt-send0 asid upif* (*pkt2to1loc m*)
| π_2 (*evt-deliver2 asid m*) = *evt-deliver0 asid* (*pkt2to1loc m*)
| π_2 (*evt-dispatch-ext2 asid upif m*) = *evt-dispatch-ext0 asid upif* (*pkt2to1chan 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* (*pkt2to1loc 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* (*pkt2to1chan m*)→ *R21 s'*
apply(*auto simp add*: *TW.takeW-split-tail dp1-defs dp2-defs terms-pkt-def*
elim!: *dp2-in-chan2-to-0E dp2-add-loc20E intro*: *head.cases*[**where** *?x=fut*]
intro!: *exI*[*of - AHI hf1*])
apply(*rule exI*[*of - AHIS* (*hfs-valid-prefix-generic ainfo* (*upd-uinfo-pkt* (*fwd-pkt* (*upd-pkt m*)))
(*hf1#pas*) (*Some hf1*) *fut None*)])
apply *auto*
subgoal
thm *cons-hfs-valid-prefix-generic*[**where** *?uinfo* =*upd-uinfo* - -, **where** *?hist* = *hist*]
apply(*frule cons-hfs-valid-prefix-generic*[**where** *?uinfo* =*upd-uinfo* - -, **where** *?hist* = *hist*])
by (*auto simp add*: *upd-pkt-def*)
apply(*auto simp add*: *TW.takeW-split-tail dp1-defs dp2-defs terms-pkt-def*
elim!: *dp2-in-chan2-to-0E dp2-add-loc20E intro*: *head.cases*[**where** *?x=fut*])
apply(*frule cons-hfs-valid-prefix-generic*[**where** *?uinfo* =*upd-uinfo* - -, **where** *?hist* = *hist*])
by (*auto simp add*: *upd-pkt-def*)
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* (*pkt2to1loc 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 *asm*: *dp2-deliver s m asid ainfo uinfo hf1 pas fut hist s'*
then show *dp1*: *R21 s—evt-deliver0 asid* (*pkt2to1loc 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* (*upd-uinfo-pkt* (*fwd-pkt m*)) (*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 asm* **by** (*simp add*: *cons-hfs-valid-prefix-generic dp2-defs*)
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* (*pkt2to1chan m*)→ *R21 s'*
apply(*auto simp add*: *dp1-defs dp2-defs* ‹*inv-ik-dyn s*› *upd-uinfo-no-oracle simp del*: *AHIS-def*
intro!: *ik-seg-is-auth elim*!: *dp2-add-chan20E*)
apply(*cases fut*)
by(*auto simp add*: *upd-uinfo-no-oracle*)
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*, *'uinfo*, *'uhi*, *'ainfo*) *pkt2* ⇒ *bool* **where**
auth-path2 m ≡ *pfragment* (*AInfo m*) (*rev* (*history m*)) *auth-seg0*

abbreviation *TR-auth2-hist* :: (*'aahi*, *'uinfo*, *'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* (*pkt2to1loc 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* (*pkt2to1chan 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* (*pkt2to1loc 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* (*pkt2to1loc mc*)) @ *zs2*])
done

lemma *chan2to0-auth*:
⟦*mc* ∈ *chan2 sc* (*a1*, *i1*, *a2*, *i2*); *sa* = *R10* (*R21 sc*); *ma* = *pkt1to0chan a1 i1* (*pkt2to1chan 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* (*pkt1to0chan a1 i1* (*pkt2to1chan 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*) **using** *loc2to0*
by(*auto simp add*: *soup-def inv-auth-def elim*!: *allE*)
subgoal
apply(*frule chan2to0-auth*) **using** *chan2to0*
by(*fastforce simp add*: *soup-def inv-auth-def elim*!: *allE*)+
done
done

definition *inv-detect2* :: ('*aahi*, '*uinfo*, '*uhi*, '*ainfo*) *dp2-state* ⇒ *bool* **where**
inv-detect2 s ≡ ∀ *m* . *soup2 m s* ⟶ *prefix* (*history m*) (*AHIS* (*past m*))

abbreviation *TR-detect2* **where** *TR-detect2* ≡ {τ | τ . ∀ *s* . *evt-observe2 s* ∈ *set* τ ⟶ *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 (pkt2to1loc 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 (pkt2to1chan 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 infrastructure/Take-While-Update
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 next hop field as well as on ainfo and uinfo. Note that this includes checking the validity of the info fields.
 fixes *hf-valid* :: *'ainfo* ⇒ *'uinfo*
  ⇒ (*'aahi*, *'uhi*) *HF*
  ⇒ (*'aahi*, *'uhi*) *HF option* ⇒ *bool*
— We need *auth-restrict* to further restrict the set of authorized segments. For instance, we need it for the empty segment (ainfo, []) since according to the definition any such ainfo will be contained in the intruder knowledge. With *auth-restrict* we can restrict this.
 and *auth-restrict* :: *'ainfo* ⇒ *'uinfo* ⇒ (*'aahi*, *'uhi*) *HF list* ⇒ *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*
— *term-ainfo* extracts what msgterms the intruder can learn from analyzing a given authenticated info field.
 and *term-ainfo* :: *'ainfo* ⇒ *msgterm*
— *terms-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 *terms-hf* :: (*'aahi*, *'uhi*) *HF* ⇒ *msgterm set*
— *terms-uinfo* extracts what msgterms the intruder can learn from analyzing a given uinfo field.

and *terms-uinfo* :: *'uinfo* ⇒ *msgterm set*
— *upd-uinfo* returns the updated uinfo field of a packet.
and *upd-uinfo* :: *'uinfo* ⇒ (*'aahi*, *'uhi*) *HF* ⇒ *'uinfo*
— As *ik-oracle* (defined below) 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* ⇒ *'uinfo* ⇒ *bool*
begin

abbreviation *hf-valid-generic* :: *'ainfo* ⇒ *'uinfo*
⇒ (*'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 hf nxt*

definition *hfs-valid-prefix-generic* ::
'ainfo ⇒ *'uinfo* ⇒ (*'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 ≡
TWu.takeW (λ *uinfo hf nxt* . *hf-valid ainfo uinfo hf nxt*) *upd-uinfo uinfo fut nxt*

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

sublocale *dataplane-2-defs - - - auth-seg0 hf-valid-generic hfs-valid-prefix-generic auth-restrict extr extr-ainfo term-ainfo terms-hf terms-uinfo upd-uinfo*
apply *unfold-locales* **done**

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

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

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

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

abbreviation *upds-uinfo* **where**
upds-uinfo ≡ *foldl upd-uinfo*

abbreviation *upds-uinfo-shifted* **where**
upds-uinfo-shifted uinfo l nxt ≡ *TWu.upd-shifted upd-uinfo uinfo l nxt*

end

print-locale *dataplane-3-directed-defs*
locale *dataplane-3-directed-ik-defs* = *dataplane-3-directed-defs - - - - hf-valid auth-restrict extr extr-ainfo term-ainfo terms-hf - upd-uinfo* **for**
hf-valid :: *'ainfo* ⇒ *'uinfo* ⇒ (*'aahi*, *'uhi*) *HF* ⇒ (*'aahi*, *'uhi*) *HF option* ⇒ *bool*
and *auth-restrict* :: *'ainfo* => *'uinfo* ⇒ (*'aahi*, *'uhi*) *HF list* ⇒ *bool*
and *extr* :: *msgterm* ⇒ *'aahi ahi-scheme list*
and *extr-ainfo* :: *msgterm* ⇒ *'ainfo*
and *term-ainfo* :: *'ainfo* ⇒ *msgterm*
and *terms-hf* :: (*'aahi*, *'uhi*) *HF* ⇒ *msgterm set*
and *upd-uinfo* :: *'uinfo* ⇒ (*'aahi*, *'uhi*) *HF* ⇒ *'uinfo*
\+
— *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*
begin

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

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

lemma *cons-hfs-valid-prefix-generic*:
⟦*hf-valid-generic ainfo uinfo hfs* (*head pas*) *hf1* (*head fut*); *hfs* = (*rev pas*)@*hf1* #*fut*;
m = (|*AInfo* = *ainfo*, *UInfo* = *uinfo*, *past* = *pas*, *future* = *hf1* # *fut*, *history* = *hist*|)⟧
⟹ *hfs-valid-prefix-generic ainfo uinfo pas* (*head pas*) (*hf1* # *fut*) *None* =
hf1 # (*hfs-valid-prefix-generic ainfo* (*upd-uinfo-pkt* (*fwd-pkt m*)) (*hf1*#*pas*) (*Some hf1*) *fut None*)
apply *auto*
apply(*cases fut*)
apply *auto*
by (*auto simp add*: *TWu.takeW.simps*)

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

2.6.2 Conditions 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 - - - - - no-oracle hf-valid auth-restrict extr extr-ainfo term-ainfo - upd-uinfo ik-add ik-oracle*
for *hf-valid* :: *'ainfo* ⇒ *'uinfo*
⇒ (*'aahi*, *'uhi*) *HF*
⇒ (*'aahi*, *'uhi*) *HF option* ⇒ *bool*

and *auth-restrict* :: *'ainfo* => *'uinfo* ⇒ (*'aahi*, *'uhi*) *HF list* ⇒ *bool*
and *extr* :: *msgterm* ⇒ *'aahi ahi-scheme list*
and *extr-ainfo* :: *msgterm* ⇒ *'ainfo*
and *term-ainfo* :: *'ainfo* ⇒ *msgterm*
and *upd-uinfo* :: *'uinfo* ⇒ (*'aahi*, *'uhi*) *HF* ⇒ *'uinfo*
and *ik-add* :: *msgterm set*
and *ik-oracle* :: *msgterm set*
and *no-oracle* :: *'ainfo* ⇒ *'uinfo* ⇒ *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 *terms-hf* to its argument. *terms-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-terms-hf*:
⟦*hf-valid ainfo uinfo hf nxt*; *terms-hf hf* ⊆ *analz ik*; *no-oracle ainfo uinfo*⟧
⟹ ∃ *hfs* . *hf* ∈ *set hfs* ∧ (∃ *uinfo'* . (*ainfo*, *hfs*) ∈ (*auth-seg2 uinfo'*))
— 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 hf nxt*; *terms-hf hf* ⊆ *synth* (*analz ik*);
no-oracle ainfo uinfo⟧
⟹ *terms-hf hf* ⊆ *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 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 ?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 l nxt*; *auth-restrict ainfo uinfo l*; *nxt* = *None*⟧
⟹ *prefix* (*extr-from-hd l*) (*AHIS l*)
— A valid hop field is only valid for one specific uinfo.
and *COND-hf-valid-uinfo*:
⟦*hf-valid ainfo uinfo hf nxt*; *hf-valid ainfo' uinfo' hf nxt'*⟧
⟹ *uinfo'* = *uinfo*
— Updating a uinfo field does not reveal anything novel to the attacker.
and *COND-upd-uinfo-ik*:
⟦*terms-uinfo uinfo* ⊆ *synth* (*analz ik*); *terms-hf hf* ⊆ *synth* (*analz ik*)⟧
⟹ *terms-uinfo* (*upd-uinfo uinfo hf*) ⊆ *synth* (*analz ik*)
— The determination of whether a packet is an oracle packet is invariant under uinfo field updates.
and *COND-upd-uinfo-no-oracle*:
no-oracle ainfo uinfo ⟹ *no-oracle ainfo* (*upd-uinfo uinfo fld*)
— The restriction on authorized paths is invariant under uinfo field updates.
and *COND-auth-restrict-upd*:
auth-restrict ainfo uinfo (*hf1* # *hf2* # *xs*) ⟹ *auth-restrict ainfo* (*upd-uinfo uinfo hf2*) (*hf2* # *xs*)
begin

lemma *holds-path-eq-extr*:
⟦*hfs-valid ainfo uinfo l nxt*; *auth-restrict ainfo uinfo l*; *nxt* = *None*⟧ ⟹ *extr-from-hd l* = *AHIS l*
using *COND-extr-prefix-path COND-path-prefix-extr*
by (*metis TWu.holds-implies-takeW-is-identity prefix-order.eq-iff*)

lemma *upds-uinfo-no-oracle*:
no-oracle ainfo uinfo ⟹ *no-oracle ainfo* (*upds-uinfo uinfo hfs*)
by (*induction hfs rule*: *rev-induct*, *auto intro*!: *COND-upd-uinfo-no-oracle*)

2.6.3 Lemmas that are needed for the refinement proof

thm *COND-upd-uinfo-ik COND-upd-uinfo-ik*[*THEN subsetD*] *subsetI*
lemma *upd-uinfo-ik-elem*:
⟦*t* ∈ *terms-uinfo* (*upd-uinfo uinfo hf*); *terms-uinfo uinfo* ⊆ *synth* (*analz ik*); *terms-hf hf* ⊆ *synth* (*analz ik*)⟧
⟹ *t* ∈ *synth* (*analz ik*)
oops

lemma *honest-hf-analz-subsetI*:
⟦*t* ∈ *terms-hf hf*; *ASID* (*AHI hf*) ∉ *bad*; *hf-valid ainfo uinfo hf nxt*; *terms-hf hf* ⊆ *synth* (*analz ik*);
no-oracle ainfo uinfo⟧
⟹ *t* ∈ *analz ik*
using *COND-honest-hf-analz subsetI* **by** *blast*

lemma *extr-from-hd-eq*: (*l* ≠ [] ∧ *l′* ≠ [] ∧ *hd l* = *hd l′*) ∨ (*l* = [] ∧ *l′* = []) ⟹ *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′* ≠ []⟧ ⟹ *prefix* (*AHIS* (*hfs-valid-prefix ainfo uinfo 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′* ≠ []; *hf* = *hd l′*⟧ ⟹ *prefix* (*AHIS* (*hfs-valid-prefix ainfo uinfo 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 *auth-restrict-app*:
assumes *auth-restrict ainfo uinfo p p* = *pre* @ *hf* # *post*
shows *auth-restrict ainfo* (*upds-uinfo-shifted uinfo pre hf*) (*hf* # *post*)
using *assms* **proof**(*induction pre arbitrary*: *p uinfo hf*)
case *Nil*
then show *?case* **using** *assms* **by** (*simp add*: *TWu.upd-shifted.simps*(*2*))
next

```
  case induct-asm: (Cons a prev)
  show ?case
  proof(cases prev)
    case Nil
    then have auth-restrict ainfo (upd-uinfo uinfo hf) (hf # post)
      using induct-asm COND-auth-restrict-upd by simp
    then show ?thesis
      using induct-asm Nil by (auto simp add: TWu.upd-shifted.simps)
  next
    case cons-asm: (Cons b list)
    then have auth-restrict ainfo (upd-uinfo uinfo b) (b # list @ hf # post)
      using induct-asm(2−3) COND-auth-restrict-upd by auto
    then show ?thesis
      using induct-asm(1)
      by (simp add: cons-asm TWu.upd-shifted.simps)
  qed
qed

lemma hfs-valid-None-Cons:
  assumes hfs-valid-None ainfo uinfo p p = hf1 # hf2 # post
  shows hfs-valid-None ainfo (upd-uinfo uinfo hf2) (hf2 # post)
  using assms apply auto
  by(auto simp add: TWu.holds.simps(1))

lemma pfrag-extr-auth:
assumes hf ∈ set p and (ainfo, p) ∈ (auth-seg2 uinfo)
shows pfragment ainfo (extr (HVF hf)) auth-seg0
proof −
  have p-verified: hfs-valid-None ainfo uinfo p auth-restrict ainfo uinfo p
      using assms(2) auth-seg2-def TWu.holds-takeW-is-identity by fastforce+
    obtain pre post where p-def: p = pre @ hf # post using assms(1) split-list by metis
    then have hf-post-valid:
          hfs-valid-None ainfo (upds-uinfo-shifted uinfo pre hf) (hf # post)
          auth-restrict ainfo (upds-uinfo-shifted uinfo pre hf) (hf # post)
    using p-verified by (auto intro: TWu.holds-intermediate auth-restrict-app)

  then have pfragment ainfo (AHIS (hf # post)) auth-seg0
    using assms(2) p-def by(auto intro!: pfragmentI[of - AHIS pre - []] simp add: auth-seg2-def)

  then have pfragment ainfo (extr-from-hd (hf # post)) auth-seg0
    using holds-path-eq-extr[symmetric] hf-post-valid by metis
  then show ?thesis by simp
qed

lemma X-in-ik-is-auth:
  assumes terms-hf hf1 ⊆ analz ik and no-oracle ainfo uinfo
  shows pfragment ainfo (AHIS (hfs-valid-prefix ainfo uinfo
              (hf1 # fut)
              nxt))
            auth-seg0
proof −
  let ?pFu = hf1 # fut
  let ?takW = (hfs-valid-prefix ainfo uinfo ?pFu nxt)
```

have *prefix* (*AHIS* (*hfs-valid-prefix ainfo uinfo ?takW* (*TWu.extract* (*hf-valid ainfo*) *upd-uinfo uinfo ?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*: *TWu.takeW-takeW-extract*)
moreover from *assms* **have** *pfragment ainfo* (*extr-from-hd ?takW*) *auth-seg0*
by (*auto simp add*: *TWu.takeW-split-tail dest*!: *COND-terms-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-terms-hf*[*elim*]:
⟦*x* ∈ *terms-hf hf*; *hf* ∈ *set hfs*; (*ainfo*, *hfs*) ∈ (*auth-seg2 uinfo*) ⟧ ⟹ *x* ∈ *analz ik*
by(*auto 3 4 simp add*: *ik-def*)

lemma ⟦*hfs-valid ainfo uinfo hfs nxt*; *hfs* = *pref* @ [*hf*]⟧ ⟹ *hf-valid ainfo* (*upds-uinfo uinfo pref*) *hf nxt*
apply *auto*
thm *TWu.holds-append*[**where** *?P*=(*hf-valid ainfo*), **where** *?upd*=*upd-uinfo*]
apply(*auto simp add*: *TWu.holds-append*[**where** *?P*=(*hf-valid ainfo*), **where** *?upd*=*upd-uinfo*])
apply(*cases pref*) **apply** *auto*
apply (*simp add*: *TWu.holds.simps*(*2*))
apply (*simp add*: *TWu.holds.simps*(*2*))

oops

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 *terms-hf hf2* ⊆ *synth* (*analz ik*)
  and *ASID* (*AHI hf1*) ∈ *bad*
  and *ASID* (*AHI hf2*) ∉ *bad*
  and *hf-valid ainfo uinfo hf1* (*Some hf2*)
  and *no-oracle ainfo uinfo*
  shows
   pfragment ainfo
     (*ifs-valid-prefix pre′*
       (*AHIS* (*hfs-valid-prefix ainfo uinfo*
         (*hf1* # *hf2* # *fut*)
         None))
     None)
    auth-seg0
proof(*cases*)
  assume *hf-valid ainfo* (*upd-uinfo uinfo hf2*) *hf2* (*head fut*)
    ∧ *if-valid* (*Some* (*AHI hf1*)) (*AHI hf2*) (*AHIo* (*head fut*))

  then have *hf2true*: *hf-valid ainfo* (*upd-uinfo uinfo hf2*) *hf2* (*head fut*)
      if-valid (*Some* (*AHI hf1*)) (*AHI hf2*) (*AHIo* (*head fut*)) **by** *blast+*
  then have ∃ *hfs uinfo′* . *hf2* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ (*auth-seg2 uinfo′*)
    using *assms* **by**(*auto intro*!: *COND-terms-hf COND-upd-uinfo-no-oracle*
      elim!: *honest-hf-analz-subsetI*)
  then obtain *hfs uinfo′* **where** *hfs-def*:
   hf2 ∈ *set hfs* (*ainfo*, *hfs*) ∈ (*auth-seg2 uinfo′*) *hfs-valid-None ainfo uinfo′ hfs*
   no-oracle ainfo uinfo′
    using *COND-terms-hf* **by**(*auto simp add*: *auth-seg2-def TWu.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*(*3*) *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* (*upds-uinfo uinfo′* (*tl* (*pref*@[*hf1′*, *hf2*]))) *hf2* (*head ys*)
    and *hf1′true*: *hf-valid ainfo* (*upds-uinfo uinfo′* (*tl* (*pref*@[*hf1′*]))) *hf1′* (*Some hf2*)

    using *hfs-def*(*3*) *hfs-unfold*

apply(*auto simp add*: *TWu.holds-append*[**where** *?P*=(*hf-valid ainfo*), **where** *?upd*=*upd-uinfo*])
apply (*auto simp add*: *hfs-def hfs-unfold TWu.holds-unfold-prelnil tail-snoc TWu.holds.simps*(*1*)
elim!: *TWu.holds-unfold-prexnxt′ intro*: *rev-exhaust*[**where** *?xs*=*pref*])
subgoal
apply(*cases pref*) **apply** *auto*
apply (*metis TWu.holds.simps*(*1*) *TWu.holds.simps*(*2*) *head.elims*)
by (*metis TWu.holds.simps*(*1*) *TWu.holds.simps*(*2*) *head.elims*)
subgoal
apply(*cases pref*) **by** (*auto simp add*: *TWu.holds.simps*)
done

have *uinfo′-eq*: *upd-uinfo uinfo hf2* = *upds-uinfo uinfo′* (*tl* (*pref* @ [*hf1′*, *hf2*]))
using *hf2-valid hf2true*(*1*) **by**(*auto elim*!: *COND-hf-valid-uinfo*)
then have *uinfo′-eq2*: *upd-uinfo uinfo hf2* = *upd-uinfo* (*upds-uinfo uinfo′* (*tl* (*pref* @ [*hf1′*]))) *hf2*
by(*simp add*: *tl-append2*)

have *if-valid-hf2hf1′*: *if-valid* (*Some* (*AHI hf1′*)) (*AHI hf2*) (*head* (*AHIS ys*))
apply(*cases ys*)
using *assms*(*3*) *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* (*upds-uinfo uinfo′* (*tl* (*pref*@[*hf1′*])))
(*hf1′* # *hf2* # *fut*)
None))
auth-seg0
apply(*rule X-in-ik-is-auth*)
using *hfs-def*(*1*,*2*,*4*) *assms*(*2*,*5*)
by(*fastforce simp add*: *hfs-unfold ik-def*
intro!: *upds-uinfo-no-oracle COND-upd-uinfo-no-oracle*)+

then show *?thesis*
apply−
apply(*rule strip-ifs-valid-prefix*)
apply(*erule pfragment-self-eq-nil*)
apply *auto*
apply(*auto simp add*: *TWu.takeW-split-tail*[**where** *?x*=*hf1′*])
subgoal
using *assms*(*2*−*4*) *hf2true*(*2*) *if-valid-hf2hf1′ hf1′true*
by(*auto elim*!: *attacker-modify-cutoff*[**where** *?hf′*=*AHI hf1*] *simp add*: *TWu.takeW-split-tail uinfo′-eq2*)
subgoal
using *hf1′true* **by**(*auto*)
done
next
assume *hf2false*: ¬(*hf-valid ainfo* (*upd-uinfo uinfo hf2*) *hf2* (*head fut*)
∧ *if-valid* (*Some* (*AHI hf1*)) (*AHI hf2*) (*AHIo* (*head fut*)))
show *?thesis*
proof(*cases*)
assume *hf1-correct*: *hf-valid ainfo uinfo hf1* (*Some hf2*)

∧ *if-valid pre′* (*AHI hf1*) (*Some* (*AHI hf2*))
show *?thesis*
proof(*cases*)
assume *hf2-valid*: *hf-valid ainfo* (*upd-uinfo uinfo hf2*) *hf2* (*head fut*)
then have ¬*if-valid* (*Some* (*AHI hf1*)) (*AHI hf2*) (*AHIo* (*head fut*)) **using** *hf2false* **by** *simp*
then show *?thesis*
using *hf1-correct hf2-valid* **apply**(*auto*)
using *assms*(*2*) **by**(*auto simp add*: *TW.takeW-split-tail TWu.takeW-split-tail intro*: *ASM-singleton*)

next
assume ¬*hf-valid ainfo* (*upd-uinfo uinfo hf2*) *hf2* (*head fut*)
then show *?thesis* **apply** *auto* **apply**(*cases fut*)
using *assms*(*2*)
by(*auto simp add*: *TW.takeW-split-tail*[**where** *?x=hf1*] *TWu.takeW-split-tail TW.takeW.simps*

intro: *ASM-singleton intro*!: *strip-ifs-valid-prefix*)
qed
next
assume ¬ (*hf-valid ainfo uinfo hf1* (*Some hf2*)
∧ *if-valid pre′* (*AHI hf1*) (*Some* (*AHI hf2*)))
then show *?thesis*
using *hf2false* **apply** *auto*
by (*auto simp add*: *TW.takeW.simps TWu.takeW-split-tail*[**where** *?x=hf1*] *TWu.takeW-split-tail*[**where** *?x=hf2*]
ASM-singleton assms(*2*) *strip-ifs-valid-prefix*)
qed
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* (*upd-uinfo uinfo hf2*)
(*hf2* # *fut*)
nxt))
None)
auth-seg0
assumes *hf-valid ainfo uinfo 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*
(*hf1* # *hf2* # *fut*)
nxt))
None)
auth-seg0
using *assms*
apply(*auto simp add*: *TWu.takeW-split-tail*[**where** *?P=hf-valid ainfo*])
apply(*auto simp add*: *TWu.takeW-split-tail*[**where** *?P=if-valid*] *TWu.takeW.simps*(*1*)

intro: *ASM-singleton strip-ifs-valid-prefix intro*!: *extend-pfragment0*)
by (*simp add*: *TW.takeW-split-tail extend-pfragment0*)

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

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 *terms-pkt m* ⊆ *synth* (*analz ik*) **and** *future m* = *hfs* **and** *AInfo m* = *ainfo*
and *nxt* = *None* **and** *no-oracle ainfo uinfo*
shows *pfragment ainfo*
(*ifs-valid-prefix prev′*
(*AHIS* (*hfs-valid-prefix ainfo uinfo hfs nxt*))
None)
auth-seg0
using *assms*
proof(*induction hfs nxt arbitrary*: *prev′ m rule*: *TWu.takeW.induct*[**where** *?Pa=hf-valid ainfo*, **where** *?upd=upd-uinfo*])
case (*1 - -*)
then show *?case* **using** *append-Nil ASM-empty pfragment-def Nil-is-map-conv TWu.takeW.simps*(*1*) *AHIS-def*
by (*metis TW.takeW.simps*(*1*) *hfs-valid-prefix-generic-def*)
next
case (*2 pre hf nxt*)
then show *?case*
proof(*cases*)
assume *ASID* (*AHI hf*) ∈ *bad*
then show *?thesis* **apply**−
by(*intro strip-ifs-valid-prefix*)
(*auto simp add*: *pfragment-def ASM-singleton TWu.takeW-singleton intro*!: *exI*[*of* - []])
next
assume *ASID* (*AHI hf*) ∉ *bad*
then show *?thesis*
apply(*intro strip-ifs-valid-prefix*) **apply**(*cases m*)
using *2 assms* **by** (*auto simp add*: *terms-pkt-def*
simp del: *AHIS-def intro*!: *X-in-ik-is-auth dest*: *COND-honest-hf-analz*)
qed
next
case (*3 info hf nxt*)
then show *?case*
by (*intro strip-ifs-valid-prefix*, *simp add*: *TWu.takeW-singleton*)
next
case (*4 info hf1 hf2 xs nxt*)
then show *?case*
proof(*cases*)
assume *hf1bad*: *ASID* (*AHI hf1*) ∈ *bad*
then show *?thesis*
proof(*cases*)
assume *hf2bad*: *ASID* (*AHI hf2*) ∈ *bad*
show *?thesis*

```
      apply(intro extend-pfragment2)
      subgoal
        apply (auto simp add: 4(6)) apply(cases m)
        apply(rule 4(2)[simplified, where ?m1=fwd-pkt (upd-pkt m)])
        using 4(3−7) by (auto simp add: terms-pkt-def upd-pkt-def
                          intro: COND-upd-uinfo-no-oracle COND-upd-uinfo-ik[THEN subsetD])
      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(1,4−7) hf1bad apply(simp add: hfs-valid-prefix-generic-def)
        using 4(3)
        by(auto 3 4 intro!: fragment-with-Eo-Some-extendable[simplified]
              simp add: terms-pkt-def simp del: hfs-valid-prefix-generic-def AHIS-def)
    qed
  next
    assume ASID (AHI hf1) ∉ bad
    then show ?thesis
      apply(intro strip-ifs-valid-prefix)
      using 4(1,3−7) by(auto intro!: X-in-ik-is-auth simp del: AHIS-def simp add: terms-pkt-def
                          dest: COND-honest-hf-analz)
  qed
next
  case 5
  then show ?case
    by(intro strip-ifs-valid-prefix, simp add: TWu.takeW-two-or-more)
qed

lemma ik-seg-is-auth′:
  assumes terms-pkt m ⊆ synth (analz ik)
    and future m = hfs and AInfo m = ainfo and nxt = None and no-oracle ainfo uinfo
  shows pfragment ainfo
          (ifs-valid-prefix prev′
            (AHIS (hfs-valid-prefix-generic ainfo uinfo pas pre hfs nxt))
           None)
             auth-seg0
  using ik-seg-is-auth hfs-valid-prefix-generic-def assms
  by simp

print-locale dataplane-2
sublocale dataplane-2 - - - - hfs-valid-prefix-generic - - - - - - no-oracle - - hf-valid-generic upd-uinfo
  apply unfold-locales
  using ik-seg-is-auth′ COND-upd-uinfo-ik COND-upd-uinfo-no-oracle
    prefix-hfs-valid-prefix-generic cons-hfs-valid-prefix-generic apply auto
  by (metis list.inject)

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.

Note that we don't use the update function in the undirected setting, since none of the instances require it.

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

type-synonym *UINFO* = *msgterm*

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* × *'aahi ahi-scheme list*) *set* +
— *hf-valid* is the check that every hop performs on its own and the entire path as well as on ainfo and uinfo. Note that this includes checking the validity of the info fields.
  fixes *hf-valid* :: *'ainfo* ⇒ *UINFO*
    ⇒ (*'aahi*, *'uhi*) *HF list*
    ⇒ (*'aahi*, *'uhi*) *HF*
    ⇒ *bool*
— We need *auth-restrict* to further restrict the set of authorized segments. For instance, we need it for the empty segment (ainfo, []) since according to the definition any such ainfo will be contained in the intruder knowledge. With *auth-restrict* we can restrict this.
  and *auth-restrict* :: *'ainfo* ⇒ *UINFO* ⇒ (*'aahi*, *'uhi*) *HF list* ⇒ *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*
— *term-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 *term-ainfo* :: *'ainfo* ⇒ *msgterm*
— *terms-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 *terms-hf* :: (*'aahi*, *'uhi*) *HF* ⇒ *msgterm set*
— *terms-uinfo* extracts what msgterms the intruder can learn from analyzing a given uinfo field.
and *terms-uinfo* :: *UINFO* ⇒ *msgterm set*
— As *ik-oracle* (defined below) 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* ⇒ *UINFO* ⇒ *bool*

begin

abbreviation *upd-uinfo* :: *UINFO* ⇒ (*'aahi*, *'uhi*) *HF* ⇒ *UINFO* **where**
upd-uinfo u hf ≡ *u*

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 hfs pre hf nxt ≡ *hf-valid ainfo uinfo hfs hf*

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

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 ≡
hfs-valid-prefix ainfo uinfo pas fut

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

sublocale *dataplane-2-defs - - - auth-seg0 hf-valid-generic hfs-valid-prefix-generic auth-restrict extr extr-ainfo term-ainfo terms-hf terms-uinfo upd-uinfo*
apply *unfold-locales* **done**

lemma *auth-seg2-elem*: ⟦(*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*; *hf* ∈ *set hfs*⟧
⟹ ∃ *uinfo* . *hf-valid ainfo uinfo hfs hf* ∧ *auth-restrict ainfo uinfo hfs* ∧ (*ainfo*, *AHIS hfs*) ∈ *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 auth-restrict extr extr-ainfo term-ainfo terms-hf -* **for**
hf-valid :: *'ainfo* ⇒ *UINFO* ⇒ (*'aahi*, *'uhi*) *HF list* ⇒ (*'aahi*, *'uhi*) *HF* ⇒ *bool*

and *auth-restrict* :: *'ainfo* => *UINFO* ⇒ (*'aahi*, *'uhi*) *HF list* ⇒ *bool*
and *extr* :: *msgterm* ⇒ *'aahi ahi-scheme list*
and *extr-ainfo* :: *msgterm* ⇒ *'ainfo*
and *term-ainfo* :: *'ainfo* ⇒ *msgterm*
and *terms-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*
begin

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*⟧
⟹ *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*)

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

2.7.2 Conditions of the parametrized model

We now list the assumptions of this parametrized model.

print-locale *dataplane-3-undirected-ik-defs*
locale *dataplane-3-undirected* = *dataplane-3-undirected-ik-defs* - - - - *terms-uinfo no-oracle hf-valid auth-restrict extr*
extr-ainfo term-ainfo terms-hf ik-add ik-oracle
for *hf-valid* :: *'ainfo* ⇒ *msgterm* ⇒ (*'aahi*, *'uhi*) *HF list* ⇒ (*'aahi*, *'uhi*) *HF* ⇒ *bool*
and *auth-restrict* :: *'ainfo* => *UINFO* ⇒ (*'aahi*, *'uhi*) *HF list* ⇒ *bool*
and *extr* :: *msgterm* ⇒ *'aahi ahi-scheme list*
and *extr-ainfo* :: *msgterm* ⇒ *'ainfo*
and *term-ainfo* :: *'ainfo* ⇒ *msgterm*
and *terms-uinfo* :: *UINFO* ⇒ *msgterm set*
and *ik-add* :: *msgterm set*
and *terms-hf* :: (*'aahi*, *'uhi*) *HF* ⇒ *msgterm set*
and *ik-oracle* :: *msgterm set*
and *no-oracle* :: *'ainfo* ⇒ *UINFO* ⇒ *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 *terms-hf* to its argument. *terms-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-terms-hf*:
$[\![$*hf-valid ainfo uinfo l hf*; *terms-hf hf* $\subseteq$ *analz ik*; *no-oracle ainfo uinfo*; *hf* $\in$ *set l*$]\!]$
$\Longrightarrow \exists$ *hfs* . *hf* $\in$ *set hfs* $\wedge$ ($\exists$ *uinfo'* . (*ainfo*, *hfs*) $\in$ (*auth-seg2 uinfo'*))

— 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 l hf*; *terms-hf hf* $\subseteq$ *synth* (*analz ik*);
no-oracle ainfo uinfo; *hf* $\in$ *set l*$]\!]$
$\Longrightarrow$ *terms-hf hf* $\subseteq$ *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 *terms-pkt m* $\subseteq$ *synth* (*analz ik*) **and**
future m = *fut* **and** *AInfo m* = *ainfo* **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 hfhonestc-def* **have** *terms-hf hfhonestc* $\subseteq$ *analz ik*
apply(*elim COND-honest-hf-analz*[**where** *l*=(*rev*(*pas*)@*fut*)])
using *assms hfhonesta-def set-takeWhileD*
apply(*auto simp add*: *terms-pkt-def*)
by *force*+
then obtain *hfshonest uinfo'* **where** *hfshonest-def*: *hfhonestc* $\in$ *set hfshonest* (*ainfo*, *hfshonest*) $\in$

auth-seg2 uinfo'
    using *hfhonestc-valid*
    apply−
    apply(*drule COND-terms-hf*) **using** *assms* **apply** *auto*
    using *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

lemma *upd-uinfo-pkt-id*[*simp*]: *upd-uinfo-pkt pkt* = *UInfo pkt*
  apply(*cases pkt*)
  subgoal for - - - *hfs*
    apply(*cases hfs*)
    by *auto*
  done

print-locale *dataplane-2*
sublocale *dataplane-2* - - - - *hfs-valid-prefix-generic* - - - - - - *no-oracle* - - *hf-valid-generic upd-uinfo*
  apply *unfold-locales*
  using *prefix-hfs-valid-prefix-generic*
  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*
⇒ *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*

definition *upd-uinfo* :: *msgterm* ⇒ *SCION-HF* ⇒ *msgterm* **where**
upd-uinfo uinfo hf ≡ *uinfo*

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* = *term2if upif*, *DownIF* = *term2if downif*, *ASID* = *asid*|) # *extr x2*
| *extr* (*Mac*[*macKey asid*] (*L* [*ts*, *upif*, *downif*]))
= [(|*UpIF* = *term2if upif*, *DownIF* = *term2if 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 *term-ainfo* :: *msgterm* ⇒ *msgterm* **where**
term-ainfo ≡ *id*

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

abbreviation *terms-uinfo* :: *msgterm* ⇒ *msgterm set* **where**
terms-uinfo x ≡ {*x*}

An authenticated info field is always a number (corresponding to a timestamp). The unauthenticated info field is set to the empty term ε.

definition *auth-restrict* **where**
auth-restrict ainfo uinfo l ≡ (∃ *ts*. *ainfo* = *Num ts*) ∧ (*uinfo* = ε)

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 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-auth-restrict*[*dest*]: *hf-valid ainfo uinfo hf z* ⟹ *auth-restrict ainfo uinfo l*
by(*auto simp add*: *hf-valid-invert auth-restrict-def*)

lemma *info-hvf*:
assumes *hf-valid ainfo uinfo m z hf-valid ainfo′ uinfo′ 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.

print-locale *dataplane-3-directed-defs*
sublocale *dataplane-3-directed-defs - - - auth-seg0 hf-valid auth-restrict extr extr-ainfo term-ainfo terms-hf terms-uinfo upd-uinfo no-oracle*
by *unfold-locales*

declare *TWu.holds-set-list*[*dest*]
declare *TWu.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 terms-uinfo no-oracle hf-valid auth-restrict extr extr-ainfo term-ainfo terms-hf upd-uinfo ik-add ik-oracle
by *unfold-locales*

lemma *auth-ainfo*[*dest*]: ⟦(*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*⟧ ⟹ ∃ *ts* . *ainfo* = *Num ts*
by(*auto simp add*: *auth-seg2-def auth-restrict-def*)

lemma *auth-uinfo*[*dest*]: ⟦(*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*⟧ ⟹ *uinfo* = *ε*
by(*auto simp add*: *auth-seg2-def auth-restrict-def*)

lemma *upds-simp*[*simp*]: *TWu.upds upd-uinfo uinfo hfs* = *uinfo*
by(*induction hfs*, *auto simp add*: *upd-uinfo-def*)

lemma *upd-shifted-simp*[*simp*]: *TWu.upd-shifted upd-uinfo uinfo hfs nxt* = *uinfo*
by(*induction hfs*, *auto simp only*: *TWu.upd-shifted.simps upds-simp*)

lemma *ik-hfs-form*: *t* ∈ *parts ik-hfs* ⟹ ∃ *t'* . *t* = *Hash t'*
by(*auto 3 4 simp add*: *auth-seg2-def hf-valid-invert*)

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

lemma *parts-ik-hfs*[*simp*]: *parts ik-hfs* = *ik-hfs*
by (*auto intro*!: *parts-Hash ik-hfs-form*)

This lemma allows us not only to expand the definition of *ik-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-hfs-simp*:
  $t \in ik\text{-}hfs \longleftrightarrow (\exists\, t' \,.\, t = Hash\ t') \land (\exists\, hf \,.\, t = HVF\ hf$
  $\land (\exists\, hfs.\ hf \in set\ hfs \land (\exists\, ainfo \,.\, (ainfo, hfs) \in (auth\text{-}seg2\ \varepsilon)$
  $\land (\exists\ nxt.\ hf\text{-}valid\ ainfo\ \varepsilon\ hf\ nxt))))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
  assume *asm*: *?lhs*
  then obtain *ainfo uinfo hf hfs* **where**
    dfs: $hf \in set\ hfs\ (ainfo, hfs) \in auth\text{-}seg2\ uinfo\ t = HVF\ hf$
    by(*auto simp add*: *ik-hfs-def*)
  then have *dfs-prop*: $hfs\text{-}valid\text{-}None\ ainfo\ \varepsilon\ hfs\ \ (ainfo, AHIS\ hfs) \in auth\text{-}seg0$
    using *auth-uinfo* **by**(*auto simp add*: *auth-seg2-def*)
  then obtain *nxt* **where** *hf-val*: $hf\text{-}valid\ ainfo\ \varepsilon\ hf\ nxt$ **using** *dfs* **apply** *auto*
    by(*auto dest*: *TWu.holds-set-list-no-update simp add*: *upd-uinfo-def*)
  then show *?rhs* **using** *asm dfs dfs-prop hf-val* **by**(*auto intro*: *ik-hfs-form*)
qed(*auto simp add*: *ik-hfs-def*)

Properties of Intruder Knowledge

lemma *Num-ik*[*intro*]: $Num\ ts \in ik$
  by(*auto simp add*: *ik-def*)
    (*auto simp add*: *auth-seg2-def auth-restrict-def TWu.holds.simps*
      intro!: *exI*[*of* - []] *exI*[*of* - ε])

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-hfs-simp auth-restrict-def*)

lemma *parts-ik*[*simp*]: $parts\ ik = ik$
  by(*fastforce simp add*: *ik-def auth-seg2-def auth-restrict-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-hfs-simp hf-valid-invert*)

lemma *MAC-synth-helper*:
  assumes $hf\text{-}valid\ ainfo\ uinfo\ m\ z\ HVF\ m = Mac[Key\ (macK\ asid)]\ j\ HVF\ m \in ik$
  shows $\exists\, hfs.\ m \in set\ hfs \land (\exists\, uinfo'.\ (ainfo, hfs) \in auth\text{-}seg2\ uinfo')$
proof−
  from *assms*(*2*−*3*) **obtain** *ainfo′ uinfo′ m′ hfs′ nxt′* **where** *dfs*:
    $m' \in set\ hfs'\ (ainfo', hfs') \in auth\text{-}seg2\ uinfo'\ hf\text{-}valid\ ainfo'\ uinfo'\ m'\ nxt'$
    $HVF\ m = HVF\ m'$
    by(*auto simp add*: *ik-def ik-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 ≡ ∃ *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 m z HVF m* ∈ *synth ik mac-format* (*HVF m*) *asid*
    asid ∉ *bad checkInfo ainfo*
  shows ∃ *hfs* . *m* ∈ *set hfs* ∧ (∃ *uinfo'*. (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo'*)
  using *assms*
  apply(*auto simp add*: *mac-format-def elim*!: *MAC-synth-helper dest*!: *key-ik-bad*)
  by(*auto simp add*: *ik-def ik-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 hf nxt terms-hf hf* ⊆ *synth* (*analz ik*)
    no-oracle ainfo uinfo
    shows *terms-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*
    by(*auto simp add*: *mac-format-def hf-valid-invert*)
  then obtain *hfs uinfo'* **where**
    hf ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo'*
    using *assms*(*1*,*2*) *hf-synth-ik* **by**(*auto dest*!: *MAC-synth*)
  then have *HVF hf* ∈ *ik*
    using *assms*(*2*)
    by(*auto simp add*: *ik-hfs-def intro*!: *ik-ik-hfs intro*!: *exI*)
  then show *?thesis* **by** *auto*
qed

lemma *COND-terms-hf*:
  assumes *hf-valid ainfo uinfo hf z* **and** *terms-hf hf* ⊆ *analz ik* **and** *no-oracle ainfo uinfo*
  shows ∃ *hfs*. *hf* ∈ *set hfs* ∧ (∃ *uinfo'* . (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo'*)
proof−
  obtain *hfs ainfo uinfo* **where** *hfs-def*: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*
    using *assms*
    using *assms*
    by *simp*
      (*auto 3 4 simp add*: *hf-valid-invert ik-hfs-simp ik-def dest*: *ahi-eq*)
  show *?thesis*
    using *hfs-def* **apply** (*auto simp add*: *auth-seg2-def dest*!: *TWu.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 l nxt*; *nxt* = *None*⟧ ⟹ *prefix* (*extr-from-hd l*) (*AHIS l*)
  by(*induction l nxt rule*: *TWu.holds.induct*[**where** *?upd*=*upd-uinfo*])
    (*auto simp add*: *TWu.holds-split-tail TWu.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 l nxt*))
(*extr-from-hd l*)
apply(*induction l nxt rule*: *TWu.takeW.induct*[**where** *?Pa=hf-valid ainfo*,**where** *?upd=upd-uinfo*])
by(*auto simp add*: *TWu.takeW-split-tail TWu.takeW.simps*(*1*))
(*auto 3 4 simp add*: *hf-valid-invert intro*!: *ahi-eq elim*: *ASIF.elims*)

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

lemma *COND-upd-uinfo-ik*:
⟦*terms-uinfo uinfo* ⊆ *synth* (*analz ik*); *terms-hf hf* ⊆ *synth* (*analz ik*)⟧
⟹ *terms-uinfo* (*upd-uinfo uinfo hf*) ⊆ *synth* (*analz ik*)
by (*auto simp add*: *upd-uinfo-def*)

lemma *COND-upd-uinfo-no-oracle*:
no-oracle ainfo uinfo ⟹ *no-oracle ainfo* (*upd-uinfo uinfo fld*)
by (*auto simp add*: *upd-uinfo-def*)

lemma *COND-auth-restrict-upd*:
auth-restrict ainfo uinfo (*x#y#hfs*)
⟹ *auth-restrict ainfo* (*upd-uinfo uinfo y*) (*y#hfs*)
by (*auto simp add*: *auth-restrict-def upd-uinfo-def*)

3.1.5 Instantiation of *dataplane-3-directed* locale

print-locale *dataplane-3-directed*
sublocale
dataplane-3-directed - - - auth-seg0 terms-uinfo terms-hf hf-valid auth-restrict extr extr-ainfo term-ainfo

upd-uinfo ik-add
ik-oracle no-oracle
apply *unfold-locales*
using *COND-terms-hf COND-honest-hf-analz COND-extr-prefix-path*
COND-path-prefix-extr COND-hf-valid-uinfo COND-upd-uinfo-ik COND-upd-uinfo-no-oracle
COND-auth-restrict-upd **by** *auto*

end
end

3.2 SCION Variant

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*.

3.3 SCION

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

3.3.1 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*
  ⇒ *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*

definition *upd-uinfo* :: *msgterm* ⇒ *SCION-HF* ⇒ *msgterm* **where**
 upd-uinfo uinfo hf ≡ *uinfo*

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*, *x2*]))
 = (|*UpIF* = *term2if upif*, *DownIF* = *term2if downif*, *ASID* = *asid*|) # *extr x2*
| *extr* (*Mac*[*macKey asid*] (*L* [*ts*, *upif*, *downif*]))
 = [(|*UpIF* = *term2if upif*, *DownIF* = *term2if 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 *term-ainfo* :: *msgterm* ⇒ *msgterm* **where**
  term-ainfo ≡ *id*

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

abbreviation *terms-uinfo* :: *msgterm* ⇒ *msgterm set* **where**
  terms-uinfo x ≡ {*x*}

An authenticated info field is always a number (corresponding to a timestamp). The unauthenticated info field is set to the empty term ε.

definition *auth-restrict* **where**
  auth-restrict ainfo uinfo l ≡ (∃ *ts*. *ainfo* = *Num ts*) ∧ (*uinfo* = ε)

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 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-auth-restrict*[*dest*]: *hf-valid ainfo uinfo hf z* ⟹ *auth-restrict ainfo uinfo l*
  by(*auto simp add*: *hf-valid-invert auth-restrict-def*)

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

3.3.2 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.

print-locale *dataplane-3-directed-defs*
sublocale *dataplane-3-directed-defs - - - auth-seg0 hf-valid auth-restrict extr extr-ainfo term-ainfo terms-hf terms-uinfo upd-uinfo no-oracle*
by *unfold-locales*

declare *TWu.holds-set-list*[*dest*]
declare *TWu.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* ≡ {}

3.3.3 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 terms-uinfo no-oracle hf-valid auth-restrict extr extr-ainfo term-ainfo
terms-hf upd-uinfo ik-add ik-oracle
by *unfold-locales*

lemma *auth-ainfo*[*dest*]: ⟦(*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*⟧ ⟹ ∃ *ts* . *ainfo* = *Num ts*
by(*auto simp add*: *auth-seg2-def auth-restrict-def*)

lemma *auth-uinfo*[*dest*]: ⟦(*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*⟧ ⟹ *uinfo* = *ε*
by(*auto simp add*: *auth-seg2-def auth-restrict-def*)

lemma *upds-simp*[*simp*]: *TWu.upds upd-uinfo uinfo hfs* = *uinfo*
by(*induction hfs*, *auto simp add*: *upd-uinfo-def*)

lemma *upd-shifted-simp*[*simp*]: *TWu.upd-shifted upd-uinfo uinfo hfs nxt* = *uinfo*
by(*induction hfs*, *auto simp only*: *TWu.upd-shifted.simps upds-simp*)

lemma *ik-hfs-form*: *t* ∈ *parts ik-hfs* ⟹ ∃ *t'* . *t* = *Hash t'*
by(*auto 3 4 simp add*: *auth-seg2-def hf-valid-invert*)

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

lemma *parts-ik-hfs*[*simp*]: *parts ik-hfs* = *ik-hfs*
by (*auto intro*!: *parts-Hash ik-hfs-form*)

This lemma allows us not only to expand the definition of *ik-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-hfs-simp*:
  $t \in ik\text{-}hfs \longleftrightarrow (\exists\, t'\, .\ t = Hash\ t') \wedge (\exists\, hf\, .\ t = HVF\ hf$
  $\wedge (\exists\, hfs.\ hf \in set\ hfs \wedge (\exists\, ainfo\, .\ (ainfo,\ hfs) \in (auth\text{-}seg2\ \varepsilon)$
  $\wedge (\exists\ nxt.\ hf\text{-}valid\ ainfo\ \varepsilon\ hf\ nxt))))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
  assume *asm*: *?lhs*
  then obtain *ainfo uinfo hf hfs* **where**
    dfs: $hf \in set\ hfs\ (ainfo,\ hfs) \in auth\text{-}seg2\ uinfo\ t = HVF\ hf$
    by(*auto simp add*: *ik-hfs-def*)
  then have *dfs-prop*: *hfs-valid-None ainfo* ε *hfs* $(ainfo,\ AHIS\ hfs) \in auth\text{-}seg0$
    using *auth-uinfo* **by**(*auto simp add*: *auth-seg2-def*)
  then obtain *nxt* **where** *hf-val*: *hf-valid ainfo* ε *hf nxt* **using** *dfs* **apply** *auto*
    by(*auto dest*: *TWu.holds-set-list-no-update simp add*: *upd-uinfo-def*)
  then show *?rhs* **using** *asm dfs dfs-prop hf-val* **by**(*auto intro*: *ik-hfs-form*)
qed(*auto simp add*: *ik-hfs-def*)

Properties of Intruder Knowledge

lemma *Num-ik*[*intro*]: $Num\ ts \in ik$
  by(*auto simp add*: *ik-def*)
    (*auto simp add*: *auth-seg2-def auth-restrict-def TWu.holds.simps*
      intro!: *exI*[*of* - []] *exI*[*of* - ε])

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-hfs-simp auth-restrict-def*)

lemma *parts-ik*[*simp*]: $parts\ ik = ik$
  by(*fastforce simp add*: *ik-def auth-seg2-def auth-restrict-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-hfs-simp hf-valid-invert*)

lemma *MAC-synth-helper*:
  assumes *hf-valid ainfo uinfo m z* $HVF\ m = Mac[Key\ (macK\ asid)]\ j\ HVF\ m \in ik$
  shows $\exists\, hfs.\ m \in set\ hfs \wedge (\exists\, uinfo'.\ (ainfo,\ hfs) \in auth\text{-}seg2\ uinfo')$
proof−
  from *assms*(*2*−*3*) **obtain** *ainfo' uinfo' m' hfs' nxt'* **where** *dfs*:
    $m' \in set\ hfs'\ (ainfo',\ hfs') \in auth\text{-}seg2\ uinfo'\ hf\text{-}valid\ ainfo'\ uinfo'\ m'\ nxt'$
    $HVF\ m = HVF\ m'$
    by(*auto simp add*: *ik-def ik-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 ≡ ∃ *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 m z HVF m* ∈ *synth ik mac-format* (*HVF m*) *asid*
    asid ∉ *bad checkInfo ainfo*
  shows ∃ *hfs* . *m* ∈ *set hfs* ∧ (∃ *uinfo′*. (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo′*)
  using *assms*
  apply(*auto simp add*: *mac-format-def elim*!: *MAC-synth-helper dest*!: *key-ik-bad*)
  by(*auto simp add*: *ik-def ik-hfs-simp*)

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 hf nxt terms-hf hf* ⊆ *synth* (*analz ik*)
    no-oracle ainfo uinfo
    shows *terms-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*
    by(*auto simp add*: *mac-format-def hf-valid-invert*)
  then obtain *hfs uinfo′* **where**
    hf ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo′*
    using *assms*(*1*,*2*) *hf-synth-ik* **by**(*auto dest*!: *MAC-synth*)
  then have *HVF hf* ∈ *ik*
    using *assms*(*2*)
    by(*auto simp add*: *ik-hfs-def intro*!: *ik-ik-hfs intro*!: *exI*)
  then show *?thesis* **by** *auto*
qed

lemma *COND-terms-hf*:
  assumes *hf-valid ainfo uinfo hf z* **and** *terms-hf hf* ⊆ *analz ik* **and** *no-oracle ainfo uinfo*
  shows ∃ *hfs*. *hf* ∈ *set hfs* ∧ (∃ *uinfo′* . (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo′*)
proof−
  obtain *hfs ainfo uinfo* **where** *hfs-def*: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*
    using *assms*
    using *assms*
    by *simp*
      (*auto 3 4 simp add*: *hf-valid-invert ik-hfs-simp ik-def dest*: *ahi-eq*)
  show *?thesis*
    using *hfs-def* **apply** (*auto simp add*: *auth-seg2-def dest*!: *TWu.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 l nxt*; *nxt* = *None*⟧ ⟹ *prefix* (*extr-from-hd l*) (*AHIS l*)
  by(*induction l nxt rule*: *TWu.holds.induct*[**where** *?upd*=*upd-uinfo*])
    (*auto simp add*: *TWu.holds-split-tail TWu.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 l nxt*))
(*extr-from-hd l*)
apply(*induction l nxt rule*: *TWu.takeW.induct*[**where** *?Pa=hf-valid ainfo*,**where** *?upd=upd-uinfo*])
by(*auto simp add*: *TWu.takeW-split-tail TWu.takeW.simps*(*1*))
(*auto 3 4 simp add*: *hf-valid-invert intro*!: *ahi-eq elim*: *ASIF.elims*)

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

lemma *COND-upd-uinfo-ik*:
⟦*terms-uinfo uinfo* ⊆ *synth* (*analz ik*); *terms-hf hf* ⊆ *synth* (*analz ik*)⟧
⟹ *terms-uinfo* (*upd-uinfo uinfo hf*) ⊆ *synth* (*analz ik*)
by (*auto simp add*: *upd-uinfo-def*)

lemma *COND-upd-uinfo-no-oracle*: *no-oracle ainfo uinfo* ⟹ *no-oracle ainfo* (*upd-uinfo-pkt m*)
by *simp*

lemma *COND-auth-restrict-upd*:
auth-restrict ainfo uinfo (*x#y#hfs*)
⟹ *auth-restrict ainfo* (*upd-uinfo uinfo y*) (*y#hfs*)
by (*auto simp add*: *auth-restrict-def upd-uinfo-def*)

3.3.5 Instantiation of *dataplane-3-directed* locale

print-locale *dataplane-3-directed*
sublocale
dataplane-3-directed - - - auth-seg0 terms-uinfo terms-hf hf-valid auth-restrict extr extr-ainfo term-ainfo

upd-uinfo ik-add
ik-oracle no-oracle
apply *unfold-locales*
using *COND-terms-hf COND-honest-hf-analz COND-extr-prefix-path*
COND-path-prefix-extr COND-hf-valid-uinfo COND-upd-uinfo-ik COND-upd-uinfo-no-oracle
COND-auth-restrict-upd **by** *auto*

end
end

3.4 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

3.4.1 Hop validation check and extract functions

type-synonym *EPIC-HF* = (*unit*, *msgterm*) *HF*
type-synonym *UINFO* = *nat*

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* ⇒ *UINFO*
⇒ *EPIC-HF*
⇒ *EPIC-HF option* ⇒ *bool* **where**
hf-valid (*Num ts*) *tspkt* (|*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*, *Num tspkt*⟩)
| *hf-valid* (*Num ts*) *tspkt* (|*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*, *Num tspkt*⟩)
| *hf-valid* - - - - = *False*

definition *upd-uinfo* :: *nat* ⇒ *EPIC-HF* ⇒ *nat* **where**

upd-uinfo uinfo hf ≡ *uinfo*

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* = *term2if upif*, *DownIF* = *term2if downif*, *ASID* = *asid*|) # *extrUhi uhi2*
| *extrUhi* (*Hash* (*Mac*[*macKey asid*] (*L* [*ts*, *upif*, *downif*])))
= [(|*UpIF* = *term2if upif*, *DownIF* = *term2if 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 *term-ainfo* :: *msgterm* ⇒ *msgterm* **where**
term-ainfo ≡ *id*

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

fun *terms-hf* :: *EPIC-HF* ⇒ *msgterm set* **where**
terms-hf hf = {*HVF hf*, *UHI hf*}

abbreviation *terms-uinfo* :: *UINFO* ⇒ *msgterm set* **where**
terms-uinfo x ≡ {}

An authenticated info field is always a number (corresponding to a timestamp). The unauthenticated info field is as well a number, representing combination of timestamp offset and SRC address.

definition *auth-restrict* **where**
auth-restrict ainfo uinfo l ≡ (∃ *ts*. *ainfo* = *Num ts*)

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 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* ∧

$mo = Some$ ⦇$AHI = ahi2$, $UHI = uhi2$, $HVF = x2$⦈ ∧
$ASID\ ahi2 = asid2 \land ASIF\ (DownIF\ ahi2)\ downif2 \land ASIF\ (UpIF\ ahi2)\ upif2 \land$
$\sigma = Mac[macKey\ asid]\ (L\ [tsn, upif, downif, uhi2]) \land$
$tsn = Num\ ts \land$
$uhi = Hash\ \sigma \land$
$x = Mac[\sigma]\ \langle tsn, Num\ uinfo\rangle)$
$\lor (\exists\ ahi\ \sigma\ ts\ upif\ downif\ asid\ uhi\ x.$
$hf =$ ⦇$AHI = ahi$, $UHI = uhi$, $HVF = x$⦈ ∧
$ASID\ ahi = asid \land ASIF\ (DownIF\ ahi)\ downif \land ASIF\ (UpIF\ ahi)\ upif \land$
$mo = None \land$
$\sigma = Mac[macKey\ asid]\ (L\ [tsn, upif, downif]) \land$
$tsn = Num\ ts \land$
$uhi = Hash\ \sigma \land$
$x = Mac[\sigma]\ \langle tsn, Num\ uinfo\rangle)$
)
apply(*auto elim*!: *hf-valid.elims*) **using** *option.exhaust ASIF.simps* **by** *metis+*

lemma *hf-valid-auth-restrict*[*dest*]: *hf-valid ainfo uinfo hf z* ⟹ *auth-restrict ainfo uinfo l*
by(*auto simp add*: *hf-valid-invert auth-restrict-def*)

lemma *auth-restrict-ainfo*[*dest*]: *auth-restrict ainfo uinfo l* ⟹ ∃ *ts. ainfo* = *Num ts*
by(*auto simp add*: *auth-restrict-def*)

lemma *info-hvf*:
assumes *hf-valid ainfo uinfo m z HVF m* = $Mac[\sigma]\ \langle ainfo', Num\ uinfo'\rangle \lor$ *hf-valid ainfo′ uinfo′ m z′*
shows *uinfo* = *uinfo′ ainfo′* = *ainfo*
using *assms* **by**(*auto simp add*: *hf-valid-invert*)

3.4.2 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.

print-locale *dataplane-3-directed-defs*
sublocale *dataplane-3-directed-defs - - - auth-seg0 hf-valid auth-restrict extr extr-ainfo term-ainfo terms-hf terms-uinfo upd-uinfo no-oracle*
by *unfold-locales*

declare *TWu.holds-set-list*[*dest*]
declare *TWu.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-hfs*), but to the underlying hop authenticators that are used to create them.

definition *ik-add* :: *msgterm set* **where**
ik-add ≡ { σ | *ainfo uinfo l hf* σ.
(*ainfo*, *l*) ∈ *auth-seg2 uinfo* ∧ *hf* ∈ *set l* ∧ *HVF hf* = $Mac[\sigma]\ \langle ainfo, Num\ uinfo\rangle$ }

lemma *ik-addI*:
⟦(*ainfo*, *l*) ∈ *auth-seg2 uinfo*; *hf* ∈ *set l*; *HVF hf* = $Mac[\sigma]\ \langle ainfo, Num\ uinfo\rangle$⟧ ⟹ σ ∈ *ik-add*
by(*auto simp add*: *ik-add-def*)

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 hf-valid-invert dest*!: *TWu.holds-set-list*)

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* $\equiv$ {}

3.4.3 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 terms-uinfo no-oracle hf-valid auth-restrict extr extr-ainfo term-ainfo
terms-hf upd-uinfo ik-add ik-oracle
by *unfold-locales*

lemma *ik-hfs-form*: $t \in$ *parts ik-hfs* $\Longrightarrow \exists\ t'$. $t =$ *Hash* t'
by(*auto 3 4 simp add*: *auth-seg2-def hf-valid-invert*)

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

lemma *parts-ik-hfs*[*simp*]: *parts ik-hfs* = *ik-hfs*
by (*auto intro*!: *parts-Hash ik-hfs-form*)

This lemma allows us not only to expand the definition of *ik-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-hfs-simp*:
$t \in$ *ik-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 uinfo*. (*ainfo*, *hfs*) $\in$ *auth-seg2 uinfo*
$\wedge$ ($\exists$ *nxt*. *hf-valid ainfo uinfo hf nxt*)))) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *asm*: *?lhs*
then obtain *ainfo uinfo hf hfs* **where**
dfs: *hf* $\in$ *set hfs* (*ainfo*, *hfs*) $\in$ *auth-seg2 uinfo* $t =$ *HVF hf* $\vee$ $t =$ *UHI hf*
by(*auto simp add*: *ik-hfs-def*)
then have *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*
using *upd-uinfo-def*
by (*auto 3 4 simp add*: *auth-seg2-def intro*!: *ik-hfs-form exI*[*of - hf*] *exI*[*of - hfs*]
dest: *TWu.holds-set-list-no-update*)
qed(*auto simp add*: *ik-hfs-def*)

Properties of Intruder Knowledge

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

lemma *Num-ik*[*intro*]: *Num ts* ∈ *ik*
by(*auto simp add*: *ik-def auth-seg2-def auth-restrict-def TWu.holds.simps 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*)
(*auto 3 4 simp add*: *ik-add-def auth-seg2-def hf-valid-invert ik-hfs-simp*)

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

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

Hop authenticators are agnostic to uinfo field

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 changes uinfo in a hop validation field.

fun *uinfo-change-hf* :: *UINFO* ⇒ *EPIC-HF* ⇒ *EPIC-HF* **where**
uinfo-change-hf new-uinfo hf =
(*case HVF hf of Mac*[σ] ⟨*ainfo*, *uinfo*⟩ ⇒ *hf*⦇*HVF* := *Mac*[σ] ⟨*ainfo*, *Num new-uinfo*⟩⦈ | - ⇒ *hf*)

fun *uinfo-change* :: *UINFO* ⇒ *EPIC-HF list* ⇒ *EPIC-HF list* **where**
uinfo-change new-uinfo hfs = *map* (*uinfo-change-hf new-uinfo*) *hfs*

lemma *uinfo-change-valid*:
hfs-valid ainfo uinfo l nxt ⟹ *hfs-valid ainfo new-uinfo* (*uinfo-change new-uinfo l*) *nxt*
apply(*induction l nxt rule*: *TWu.holds.induct*[**where** *?upd*=*upd-uinfo*])
apply *auto*
subgoal for *info x y ys nxt*
by(*cases map* (*uinfo-change-hf new-uinfo*) *ys*)
(*cases info*, *auto 3 4 simp add*: *TWu.holds-split-tail hf-valid-invert upd-uinfo-def*)+
by(*auto 3 4 simp add*: *TWu.holds-split-tail hf-valid-invert TWu.holds.simps upd-uinfo-def*)

lemma *uinfo-change-hf-AHI*: *AHI* (*uinfo-change-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-change-hf-AHIS*[*simp*]: *AHIS* (*map* (*uinfo-change-hf new-uinfo*) *l*) = *AHIS l*
apply(*induction l*) **using** *uinfo-change-hf-AHI* **by** *auto*

lemma *uinfo-change-auth-seg2*:
assumes *hf-valid ainfo uinfo m z σ* = *Mac*[*Key* (*macK asid*)] *j*

HVF m = Mac[*σ*] ⟨*ainfo*, *Num uinfo'*⟩ *σ* ∈ *ik-add*
shows ∃ *hfs*. *m* ∈ *set hfs* ∧ (∃ *uinfo''*. (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo''*)
proof−
from *assms*(*4*) **obtain** *ainfo-add uinfo-add l-add hf-add* **where**
(*ainfo-add*, *l-add*) ∈ *auth-seg2 uinfo-add hf-add* ∈ *set l-add HVF hf-add* = *Mac*[*σ*] ⟨*ainfo-add*, *Num uinfo-add*⟩
by(*auto simp add*: *ik-add-def*)
then have *add*: *m* ∈ *set* (*uinfo-change uinfo l-add*) (*ainfo-add*, (*uinfo-change uinfo l-add*)) ∈ *auth-seg2 uinfo*
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*!: *TWu.holds-set-list*)[*1*]
by(*auto simp add*: *auth-restrict-def intro*!: *exI elim*: *ahi-eq dest*: *uinfo-change-valid simp del*: *AHIS-def*)
then have *ainfo-add* = *ainfo*
using *assms*(*1*) **by**(*auto simp add*: *auth-seg2-def dest*!: *TWu.holds-set-list dest*: *info-hvf*)
then show *?thesis* **using** *add* **by** *fastforce*
qed

lemma *MAC-synth-helper*:
⟦*hf-valid ainfo uinfo m z*;
HVF m = *Mac*[*σ*] ⟨*ainfo*, *Num uinfo*⟩; *σ* = *Mac*[*Key* (*macK asid*)] *j*; *σ* ∈ *ik* ∨ *HVF m* ∈ *ik*⟧
⟹ ∃ *hfs*. *m* ∈ *set hfs* ∧ (∃ *uinfo'*. (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo'*)
apply(*auto simp add*: *ik-def ik-hfs-simp dest*: *ik-add-form*)
prefer *3* **subgoal by**(*auto elim*!: *uinfo-change-auth-seg2*)
prefer *3* **subgoal by**(*auto elim*!: *uinfo-change-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 m z HVF m* ∈ *synth ik mac-format* (*HVF m*) *asid*
asid ∉ *bad*
shows ∃ *hfs* . *m* ∈ *set hfs* ∧ (∃ *uinfo'*. (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo'*)
using *assms*
apply(*auto simp add*: *mac-format-def elim*!: *MAC-synth-helper dest*!: *key-ik-bad*)
apply(*auto simp add*: *ik-def ik-hfs-simp dest*: *ik-add-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 hf nxt terms-hf hf* ⊆ *synth* (*analz ik*)
no-oracle ainfo uinfo
shows *terms-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*
by(*auto simp add*: *mac-format-def hf-valid-invert*)
then obtain *hfs uinfo* **where** *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*
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-hfs-def intro*!: *ik-ik-hfs intro*!: *exI*)
then show *?thesis* **by** *auto*
qed

lemma *COND-terms-hf*:
assumes *hf-valid ainfo uinfo hf z* **and** *HVF hf* ∈ *ik* **and** *no-oracle ainfo uinfo*
shows ∃ *hfs*. *hf* ∈ *set hfs* ∧ (∃ *uinfo* . (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*)
proof−
obtain *hfs ainfo* **where** *hfs-def*: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*
using *assms* **by**(*auto 3 4 simp add*: *hf-valid-invert ik-hfs-simp ik-def dest*: *ahi-eq*
dest!: *ik-add-form*)
then obtain *hfs ainfo* **where** *hfs-def*: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo* **by** *auto*
show *?thesis*
using *hfs-def* **apply** (*auto simp add*: *auth-seg2-def dest*!: *TWu.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 l nxt*; *nxt* = *None*⟧ ⟹ *prefix* (*extr-from-hd l*) (*AHIS l*)
by(*induction l nxt rule*: *TWu.holds.induct*[**where** *?upd*=*upd-uinfo*])
(*auto simp add*: *upd-uinfo-def TWu.holds-split-tail TWu.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 l nxt*))
(*extr-from-hd l*)
apply(*induction l nxt rule*: *TWu.takeW.induct*[**where** *?Pa*=*hf-valid ainfo*,**where** *?upd*=*upd-uinfo*])
by(*auto simp add*: *upd-uinfo-def TWu.takeW-split-tail TWu.takeW.simps*(*1*))
(*auto 3 4 simp add*: *hf-valid-invert intro*!: *ahi-eq elim*: *ASIF.elims*)

lemma *COND-hf-valid-uinfo*:
⟦*hf-valid ainfo uinfo hf nxt*; *hf-valid ainfo′ uinfo′ hf nxt′*⟧ ⟹ *uinfo′* = *uinfo*
by(*auto dest*: *info-hvf*)

lemma *COND-upd-uinfo-ik*:
⟦*terms-uinfo uinfo* ⊆ *synth* (*analz ik*); *terms-hf hf* ⊆ *synth* (*analz ik*)⟧
⟹ *terms-uinfo* (*upd-uinfo uinfo hf*) ⊆ *synth* (*analz ik*)
by (*auto simp add*: *upd-uinfo-def*)

lemma *COND-upd-uinfo-no-oracle*:
no-oracle ainfo uinfo ⟹ *no-oracle ainfo* (*upd-uinfo uinfo fld*)
by (*auto simp add*: *upd-uinfo-def*)

lemma *COND-auth-restrict-upd*:
auth-restrict ainfo uinfo (*x*#*y*#*hfs*)
⟹ *auth-restrict ainfo* (*upd-uinfo uinfo y*) (*y*#*hfs*)

by (*auto simp add*: *auth-restrict-def upd-uinfo-def*)

3.4.5 Instantiation of *dataplane-3-directed* locale

print-locale *dataplane-3-directed*
sublocale
dataplane-3-directed - - - auth-seg0 terms-uinfo terms-hf hf-valid auth-restrict extr extr-ainfo term-ainfo

upd-uinfo ik-add
ik-oracle no-oracle
apply *unfold-locales*
using *COND-terms-hf COND-honest-hf-analz COND-extr-prefix-path*
COND-path-prefix-extr COND-hf-valid-uinfo COND-upd-uinfo-ik COND-upd-uinfo-no-oracle
COND-auth-restrict-upd **by** *auto*

end
end

3.5 EPIC Level 1 in the Strong Attacker Model

theory *EPIC-L1-SA*
  imports
    ../Parametrized-Dataplane-3-directed
    ../infrastructure/Keys
begin

type-synonym *EPIC-HF* = (*unit*, *msgterm*) *HF*
type-synonym *UINFO* = *nat*

locale *epic-l1-defs* = *network-assums-direct* - - - *auth-seg0*
  for *auth-seg0* :: (*msgterm* × *ahi list*) *set* +
  fixes *no-oracle* :: *msgterm* ⇒ *UINFO* ⇒ *bool*
begin

3.5.1 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* ⇒ *UINFO*
  ⇒ *EPIC-HF*
  ⇒ *EPIC-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*, *Num 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*, *Num uinfo*⟩)
| *hf-valid* - - - - = *False*

definition *upd-uinfo* :: *nat* ⇒ *EPIC-HF* ⇒ *nat* **where**
upd-uinfo uinfo hf ≡ *uinfo*

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* = *term2if upif*, *DownIF* = *term2if downif*, *ASID* = *asid*|) # *extrUhi uhi2*
| *extrUhi* (*Hash* (*Mac*[*macKey asid*] (*L* [*ts*, *upif*, *downif*])))
= [(|*UpIF* = *term2if upif*, *DownIF* = *term2if 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 *term-ainfo* :: *msgterm* ⇒ *msgterm* **where**
term-ainfo ≡ *id*

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

fun *terms-hf* :: *EPIC-HF* ⇒ *msgterm set* **where**
terms-hf hf = {*HVF hf*, *UHI hf*}

abbreviation *terms-uinfo* :: *UINFO* ⇒ *msgterm set* **where**
terms-uinfo x ≡ {}

An authenticated info field is always a number (corresponding to a timestamp). The unauthenticated info field is as well a number, representing combination of timestamp offset and SRC address.

definition *auth-restrict* **where**
auth-restrict ainfo uinfo l ≡ (∃ *ts*. *ainfo* = *Num ts*)

We now define useful properties of the above definition.

lemma *hf-valid-invert*:
hf-valid tsn uinfo 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*, *Num 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* ∧
uhi = *Hash* σ ∧
x = *Mac*[σ] ⟨*tsn*, *Num uinfo*⟩)
)
apply(*auto elim*!: *hf-valid.elims*) **using** *option.exhaust ASIF.simps* **by** *metis*+

lemma *hf-valid-auth-restrict*[*dest*]: *hf-valid ainfo uinfo hf z* ⟹ *auth-restrict ainfo uinfo l*
by(*auto simp add*: *hf-valid-invert auth-restrict-def*)

lemma *auth-restrict-ainfo*[*dest*]: *auth-restrict ainfo uinfo l* ⟹ ∃ *ts*. *ainfo* = *Num ts*
by(*auto simp add*: *auth-restrict-def*)

lemma *info-hvf*:
assumes *hf-valid ainfo uinfo m z HVF m* = *Mac*[σ] ⟨*ainfo*′, *Num uinfo*′⟩ ∨ *hf-valid ainfo*′ *uinfo*′ *m z*′
shows *uinfo* = *uinfo*′ *ainfo*′ = *ainfo*
using *assms* **by**(*auto simp add*: *hf-valid-invert*)

3.5.2 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.

print-locale *dataplane-3-directed-defs*
sublocale *dataplane-3-directed-defs - - - auth-seg0 hf-valid auth-restrict extr extr-ainfo term-ainfo terms-hf terms-uinfo upd-uinfo no-oracle*
by *unfold-locales*

abbreviation *is-oracle* **where** *is-oracle ainfo t* ≡ ¬ *no-oracle ainfo t*

declare *TWu.holds-set-list*[*dest*]
declare *TWu.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-hfs*), but to the underlying hop authenticators that are used to create them.

definition *ik-add* :: *msgterm set* **where**
ik-add ≡ { σ | *ainfo uinfo l hf* σ.
(*ainfo*::*msgterm*, *l*::(*EPIC-HF list*)) ∈
((*local.auth-seg2 uinfo*)::((*msgterm* × *EPIC-HF list*) *set*))

$\wedge$ *hf* $\in$ *set l* $\wedge$ *HVF hf* = *Mac*[σ] $\langle$*ainfo*, *Num uinfo*$\rangle$ }

lemma *ik-addI*:
$[\![$(*ainfo*, *l*) $\in$ *local.auth-seg2 uinfo*; *hf* $\in$ *set l*; *HVF hf* = *Mac*[σ] $\langle$*ainfo*, *Num uinfo*$\rangle]\!]$ $\Longrightarrow$ σ $\in$ *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*!: *TWu.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* | *t ainfo hf l uinfo* . *hf* $\in$ *set l* $\wedge$ *hfs-valid-None ainfo uinfo l* $\wedge$
is-oracle ainfo uinfo $\wedge$ ($\forall$ *uinfo'* . (*ainfo*, *l*) $\notin$ *auth-seg2 uinfo'*) $\wedge$
(*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*!: *TWu.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*
$\wedge$ ($\forall$ *uinfo'*. (*ainfo*, *l*) $\notin$ *auth-seg2 uinfo'*) $\wedge$ (*t* = *HVF hf* $\vee$ *t* = *UHI hf*))
by(*rule iffI*, *frule ik-oracle-parts-form*)
(*auto simp add*: *ik-oracle-def hf-valid-invert*)

3.5.3 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 terms-uinfo no-oracle hf-valid auth-restrict extr extr-ainfo term-ainfo
terms-hf upd-uinfo ik-add ik-oracle
by *unfold-locales*

lemma *ik-hfs-form*: *t* $\in$ *parts ik-hfs* $\Longrightarrow$ $\exists$ *t'* . *t* = *Hash t'*
by(*auto 3 4 simp add*: *auth-seg2-def hf-valid-invert*)

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

lemma *parts-ik-hfs*[*simp*]: *parts ik-hfs* = *ik-hfs*
  by (*auto intro*!: *parts-Hash ik-hfs-form*)

This lemma allows us not only to expand the definition of *ik-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-hfs-simp*:
  $t \in$ *ik-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 uinfo* . (*ainfo*, *hfs*) $\in$ *auth-seg2 uinfo*
    $\wedge$ ($\exists$ *nxt*. *hf-valid ainfo uinfo hf nxt*)))) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
  assume *asm*: *?lhs*
  then obtain *ainfo uinfo hf hfs* **where**
    dfs: *hf* $\in$ *set hfs* (*ainfo*, *hfs*) $\in$ *auth-seg2 uinfo* t = *HVF hf* $\vee$ t = *UHI hf*
    by(*auto simp add*: *ik-hfs-def*)
  then have *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*
    using *upd-uinfo-def*
    by (*auto 3 4 simp add*: *auth-seg2-def intro*!: *ik-hfs-form exI*[*of - hf*] *exI*[*of - hfs*]
      dest: *TWu.holds-set-list-no-update*)
qed(*auto simp add*: *ik-hfs-def*)

Properties of Intruder Knowledge

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

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 auth-restrict-def ik-hfs-simp*)
    (*auto simp add*: *ik-add-def ik-oracle-def auth-seg2-def hf-valid-invert hfs-valid-prefix-generic-def*
      dest!: *TWu.holds-set-list*)

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

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-hfs-simp ik-add-def hf-valid-invert*)

Hop authenticators are agnostic to uinfo field

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-change-hf* :: *UINFO* $\Rightarrow$ *EPIC-HF* $\Rightarrow$ *EPIC-HF* **where**

uinfo-change-hf new-uinfo hf =
  (*case HVF hf of Mac*[σ] ⟨*ainfo*, *uinfo*⟩ ⇒ *hf*⦇*HVF* := *Mac*[σ] ⟨*ainfo*, *Num new-uinfo*⟩⦈ | - ⇒ *hf*)

fun *uinfo-change* :: *UINFO* ⇒ *EPIC-HF list* ⇒ *EPIC-HF list* **where**
uinfo-change new-uinfo hfs = *map* (*uinfo-change-hf new-uinfo*) *hfs*

lemma *uinfo-change-valid*:
hfs-valid ainfo uinfo l nxt ⟹ *hfs-valid ainfo new-uinfo* (*uinfo-change new-uinfo l*) *nxt*
apply(*induction l nxt rule*: *TWu.holds.induct*[**where** *?upd*=*upd-uinfo*])
apply *auto*
subgoal for *info x y ys nxt*
  by(*cases map* (*uinfo-change-hf new-uinfo*) *ys*)
    (*cases info*, *auto 3 4 simp add*: *TWu.holds-split-tail hf-valid-invert upd-uinfo-def*)+
by(*auto 3 4 simp add*: *TWu.holds-split-tail hf-valid-invert TWu.holds.simps upd-uinfo-def*)

lemma *uinfo-change-hf-AHI*: *AHI* (*uinfo-change-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-change-hf-AHIS*[*simp*]: *AHIS* (*map* (*uinfo-change-hf new-uinfo*) *l*) = *AHIS l*
apply(*induction l*) **using** *uinfo-change-hf-AHI* **by** *auto*

lemma *uinfo-change-auth-seg2*:
assumes *hf-valid ainfo uinfo m z* σ = *Mac*[*Key* (*macK asid*)] *j*
    HVF m = *Mac*[σ] ⟨*ainfo*, *Num uinfo'*⟩ σ ∈ *ik-add no-oracle ainfo uinfo*
shows ∃ *hfs*. *m* ∈ *set hfs* ∧ (∃ *uinfo''*. (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo''*)
proof−
from *assms*(*4*) **obtain** *ainfo-add uinfo-add l-add hf-add* **where**
  (*ainfo-add*, *l-add*) ∈ *auth-seg2 uinfo-add hf-add* ∈ *set l-add HVF hf-add* = *Mac*[σ] ⟨*ainfo-add*, *Num uinfo-add*⟩
  by(*auto simp add*: *ik-add-def*)
 then have *add*: *m* ∈ *set* (*uinfo-change uinfo l-add*) (*ainfo-add*, (*uinfo-change uinfo l-add*)) ∈ *auth-seg2 uinfo*
  using *assms*(*1*−*3*,*5*) **apply**(*auto simp add*: *auth-seg2-def simp del*: *AHIS-def*)
   apply(*auto simp add*: *hf-valid-invert intro*!: *image-eqI dest*!: *TWu.holds-set-list*)[*1*]
   apply(*auto simp add*: *auth-restrict-def intro*!: *exI elim*: *ahi-eq dest*: *uinfo-change-valid simp del*: *AHIS-def*)
  by(*auto simp add*: *hf-valid-invert upd-uinfo-def dest*!: *TWu.holds-set-list-no-update*)
then have *ainfo-add* = *ainfo*
  using *assms*(*1*) **by**(*auto simp add*: *auth-seg2-def dest*!: *TWu.holds-set-list dest*: *info-hvf*)
then show *?thesis* **using** *add* **by** *fastforce*
qed

lemma *MAC-synth-oracle*:
assumes *hf-valid ainfo uinfo 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 upd-uinfo-def*
    dest!: *TWu.holds-set-list-no-update*)

lemma *ik-oracle-is-oracle*:
$[\![Mac[\sigma]\ \langle ainfo,\ Num\ uinfo\rangle \in ik\text{-}oracle]\!] \Longrightarrow is\text{-}oracle\ ainfo\ uinfo$
by (*auto simp add*: *ik-oracle-def dest*: *info-hvf*)
(*auto dest*!: *TWu.holds-set-list-no-update simp add*: *hf-valid-invert upd-uinfo-def*)

lemma *MAC-synth-helper*:
$[\![hf\text{-}valid\ ainfo\ uinfo\ m\ z;\ no\text{-}oracle\ ainfo\ uinfo;$
$HVF\ m = Mac[\sigma]\ \langle ainfo,\ Num\ uinfo\rangle;\ \sigma = Mac[Key\ (macK\ asid)]\ j;\ \sigma \in ik \lor HVF\ m \in ik]\!]$
$\Longrightarrow \exists hfs.\ m \in set\ hfs \land (\exists uinfo'.\ (ainfo,\ hfs) \in auth\text{-}seg2\ uinfo')$
apply(*auto simp add*: *ik-def ik-hfs-simp*
dest: *MAC-synth-oracle ik-add-form ik-oracle-parts-form*[*simplified*])
prefer *3* **subgoal by**(*auto elim*!: *uinfo-change-auth-seg2*)
prefer *3* **subgoal by**(*auto elim*!: *uinfo-change-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**
$mac\text{-}format\ m\ asid \equiv \exists\ j\ ts\ uinfo\ .\ m = Mac[Mac[macKey\ asid]\ j]\ \langle Num\ ts,\ uinfo\rangle$

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 $hf\text{-}valid\ ainfo\ uinfo\ m\ z\ HVF\ m \in synth\ ik\ mac\text{-}format\ (HVF\ m)\ asid$
$asid \notin bad\ no\text{-}oracle\ ainfo\ uinfo$
shows $\exists hfs\ .\ m \in set\ hfs \land (\exists uinfo'.\ (ainfo,\ hfs) \in auth\text{-}seg2\ uinfo')$
using *assms*
apply(*auto simp add*: *mac-format-def elim*!: *MAC-synth-helper dest*!: *key-ik-bad*)
apply(*auto simp add*: *ik-def ik-hfs-simp dest*: *ik-add-form dest*!: *ik-oracle-parts-form*)
using *assms*(*1*) **by**(*auto dest*: *info-hvf simp add*: *hf-valid-invert*)

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

lemma *COND-honest-hf-analz*:
assumes $ASID\ (AHI\ hf) \notin bad\ hf\text{-}valid\ ainfo\ uinfo\ hf\ nxt\ terms\text{-}hf\ hf \subseteq synth\ (analz\ ik)$
no-oracle ainfo uinfo
shows $terms\text{-}hf\ hf \subseteq analz\ ik$
proof−
let $?asid = ASID\ (AHI\ hf)$
from *assms*(*3*) **have** *hf-synth-ik*: $HVF\ hf \in synth\ ik\ UHI\ hf \in synth\ ik$ **by** *auto*
from *assms*(*2*) **have** *mac-format* (*HVF hf*) *?asid*
by(*auto simp add*: *mac-format-def hf-valid-invert*)
then obtain *hfs uinfo* **where** $hf \in set\ hfs\ (ainfo,\ hfs) \in auth\text{-}seg2\ uinfo$
using *assms*(*1*,*2*,*4*) *hf-synth-ik* **by**(*auto dest*!: *MAC-synth*)
then have $HVF\ hf \in ik\ UHI\ hf \in ik$
using *assms*(*2*)
by(*auto simp add*: *ik-hfs-def intro*!: *ik-ik-hfs intro*!: *exI*)
then show *?thesis* **by** *auto*
qed

lemma *COND-terms-hf*:
assumes *hf-valid ainfo uinfo hf z* **and** *HVF hf* ∈ *ik* **and** *no-oracle ainfo uinfo*
shows ∃ *hfs*. *hf* ∈ *set hfs* ∧ (∃ *uinfo* . (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*)
proof −
obtain *hfs ainfo* **where** *hfs-def*: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*
using *assms* **by**(*auto 3 4 simp add*: *hf-valid-invert ik-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 uinfo* **by** *auto*
show *?thesis*
using *hfs-def* **apply** (*auto simp add*: *auth-seg2-def dest*!: *TWu.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 l nxt*; *nxt* = *None*⟧ ⟹ *prefix* (*extr-from-hd l*) (*AHIS l*)
by(*induction l nxt rule*: *TWu.holds.induct*[**where** *?upd*=*upd-uinfo*])
(*auto simp add*: *upd-uinfo-def TWu.holds-split-tail TWu.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 l nxt*))
(*extr-from-hd l*)
apply(*induction l nxt rule*: *TWu.takeW.induct*[**where** *?Pa*=*hf-valid ainfo*,**where** *?upd*=*upd-uinfo*])
by(*auto simp add*: *upd-uinfo-def TWu.takeW-split-tail TWu.takeW.simps*(*1*))
(*auto 3 4 simp add*: *hf-valid-invert intro*!: *ahi-eq elim*: *ASIF.elims*)

lemma *COND-hf-valid-uinfo*:
⟦*hf-valid ainfo uinfo hf nxt*; *hf-valid ainfo′ uinfo′ hf nxt′*⟧ ⟹ *uinfo′* = *uinfo*
by(*auto dest*: *info-hvf*)

lemma *COND-upd-uinfo-ik*:
⟦*terms-uinfo uinfo* ⊆ *synth* (*analz ik*); *terms-hf hf* ⊆ *synth* (*analz ik*)⟧
⟹ *terms-uinfo* (*upd-uinfo uinfo hf*) ⊆ *synth* (*analz ik*)
by (*auto simp add*: *upd-uinfo-def*)

lemma *COND-upd-uinfo-no-oracle*:
no-oracle ainfo uinfo ⟹ *no-oracle ainfo* (*upd-uinfo uinfo fld*)
by (*auto simp add*: *upd-uinfo-def*)

lemma *COND-auth-restrict-upd*:
auth-restrict ainfo uinfo (*x*#*y*#*hfs*)
⟹ *auth-restrict ainfo* (*upd-uinfo uinfo y*) (*y*#*hfs*)
by (*auto simp add*: *auth-restrict-def upd-uinfo-def*)

3.5.5 Instantiation of *dataplane-3-directed* locale

print-locale *dataplane-3-directed*
sublocale
dataplane-3-directed - - - auth-seg0 terms-uinfo terms-hf hf-valid auth-restrict extr extr-ainfo term-ainfo

upd-uinfo ik-add
ik-oracle no-oracle

apply *unfold-locales*
using *COND-terms-hf COND-honest-hf-analz COND-extr-prefix-path COND-path-prefix-extr COND-hf-valid-uinfo COND-upd-uinfo-ik COND-upd-uinfo-no-oracle COND-auth-restrict-upd* **by** *auto*

end
end

3.6 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 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*

3.6.1 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*|)

3.6.2 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*
by(*induction ys arbitrary*: *y*)
(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def attr-segment-def TW.holds-split-tail*
TW.holds.simps list.case-eq-if)

lemma *attr-ifs-valid′*: ⟦*set ys* ⊆ *hfF-attr*; *pre* = *None*⟧ ⟹ *d0.ifs-valid pre ys nxt*
by(*induction ys nxt rule*: *TW.holds.induct*)
(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def attr-segment-def TW.holds-split-tail*
TW.holds.simps list.case-eq-if dest: *attr-ifs-valid*)

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*
by(*auto simp add*: *TW.holds-append auth-seg0-def leftsegment-wormholed-def attr-segment-def*
TW.holds-split-tail TW.holds.simps list.case-eq-if intro!: *attr-ifs-valid′*)*force*+

lemma *ASM-if-valid*: ⟦(*info*, *l*) ∈ *auth-seg0*; *pre* = *None*⟧ ⟹ *d0.ifs-valid pre l nxt*
by(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def attr-segment-def TW.holds-split-tail*
TW.holds.simps intro: *attr-ifs-valid′ leftpath-ifs-valid*)

lemma *rooted-app*[*simp*]: *d0.rooted* (*xs*@*y*#*ys*) ⟷ *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*) ∈ *auth-seg0* ⟹ *d0.rooted l*
apply(*induction l*)
apply(*auto 3 4 simp add*: *auth-seg0-def leftsegment-wormholed-def attr-segment-def TW.holds-split-tail*)
subgoal for *x xs*

by(*cases xs*, *auto*)
done

lemma *ASM-terminated*: (*info*, *l*) ∈ *auth-seg0* ⟹ *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*, []) ∈ *auth-seg0*
by(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def attr-segment-def*)

lemma *ASM-singleton*: ⟦*ASID hf* ∈ *bad*⟧ ⟹ (*info*, [*hf*]) ∈ *auth-seg0*
by(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def attr-segment-def*)

lemma *ASM-extension*:
⟦(*info*, *hf2#ys*) ∈ *auth-seg0*; *ASID hf2* ∈ *bad*; *ASID hf1* ∈ *bad*⟧
⟹ (*info*, *hf1#hf2#ys*) ∈ *auth-seg0*
by(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def TW.holds-split-tail attr-segment-def*)

lemma *ASM-modify*: ⟦(*info*, *hf#ys*) ∈ *auth-seg0*; *ASID hf* = *a*;
ASID hf′ = *a*; *UpIF hf′* = *UpIF hf*; *a* ∈ *bad*⟧ ⟹ (*info*, *hf′#ys*) ∈ *auth-seg0*
apply(*auto simp add*: *auth-seg0-def leftsegment-wormholed-def attr-segment-def*)
subgoal for *y hfa l*
by(*induction l*, *auto*)
subgoal for *y hfa l*
by(*induction l*, *auto*)
done

lemma *rightpath-no-nF*: ⟦*ASID hf* = *nF*; *zs* @ *hf* # *ys* = *rightpath*⟧ ⟹ *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‴* **by**(*cases ys‴ rule*: *rev-cases*, *auto*)
done
done
done
done

lemma *ASM-cutoff-leftpath*:
⟦*ASID hf* = *nF*;
∀ *hfa*. *UpIF hfa* = *Some nE* ⟶ *ASID hfa* = *nF* ⟶ (∀ *xs*. *hf* # *ys* = *xs* @ [*hfa*, *hiEl*, *hiDr*, *hiBr*, *hiAr*] ⟶
¬ *set xs* ⊆ *hfF-attr*); *x* ∈ *set ys*; *info* = *Num 0*;
zs @ *hf* # *ys* = *xs* @ [*hfa*, *hiEl*, *hiDr*, *hiBr*, *hiAr*]; *ASID hfa* = *nF*; *UpIF hfa* = *Some nE*; *set xs* ⊆ *hfF-attr*⟧
⟹ *ASID x* = *nF*
apply(*cases ys rule*: *rev-cases*, *simp*)
subgoal for *ys′ b*
apply(*cases ys′ rule*: *rev-cases*, *simp*)
subgoal for *ys″ c*
apply(*cases ys″ rule*: *rev-cases*, *simp*)

subgoal for *ys''' d*
apply(*cases ys'' rule*: *rev-cases*, *simp*)
subgoal for *ys''' e*
apply(*cases ys''' rule*: *rev-cases*, *simp*)
subgoal for *ys'''' f*
apply(*cases ys'''' rule*: *rev-cases*, *simp*)
by *auto blast+*
done
done
done
done
done

lemma *ASM-cutoff*: ⟦(*info*, *zs*@*hf*#*ys*) ∈ *auth-seg0*; *ASID hf* ∈ *bad*⟧ ⟹ (*info*, *hf*#*ys*) ∈ *auth-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*)

sublocale *network-assums-direct-instance*: *network-assums-direct bad tgtas tgtif auth-seg0*
apply *unfold-locales*
using *ASM-if-valid ASM-rooted ASM-terminated ASM-empty ASM-singleton ASM-extension ASM-modify ASM-cutoff*
by *simp-all*

definition *no-oracle* :: *msgterm* ⇒ *nat* ⇒ *bool* **where**
no-oracle ainfo uinfo = *True*

sublocale *e1*: *epic-l1-defs bad tgtas tgtif auth-seg0 no-oracle*
by *unfold-locales*

declare *e1.upd-uinfo-def*[*simp*]
declare *TWu.holds-takeW-is-identity*[*simp*]
thm *TWu.holds-takeW-is-identity*
declare *e1.auth-restrict-def* [*simp*]
declare *no-oracle-def* [*simp*]
declare *e1.upd-pkt-def* [*simp*]

3.6.3 Executability

Honest sender's packet forwarding

abbreviation *ainfo* **where** *ainfo* ≡ *Num 0*
abbreviation *uinfo* :: *nat* **where** *uinfo* ≡ *1*
abbreviation σA **where** σA ≡ *Mac*[*macKey nA*] (*L* [*ainfo*, ε, *AS nB*])
abbreviation σB **where** σB ≡ *Mac*[*macKey nB*] (*L* [*ainfo*, *AS nA*, *AS nD*, *Hash* σA])
abbreviation σD **where** σD ≡ *Mac*[*macKey nD*] (*L* [*ainfo*, *AS nB*, *AS nE*, *Hash* σB])
abbreviation σE **where** σE ≡ *Mac*[*macKey nE*] (*L* [*ainfo*, *AS nD*, *AS nF*, *Hash* σD])
abbreviation σF **where** σF ≡ *Mac*[*macKey nF*] (*L* [*ainfo*, *AS nE*, ε, *Hash* σE])

definition *hfAl* **where** *hfAl* ≡ (|*AHI* = *hiAl*, *UHI* = *Hash* σA, *HVF* = *Mac*[σA] ⟨*ainfo*, *Num uinfo*⟩|)
definition *hfBl* **where** *hfBl* ≡ (|*AHI* = *hiBl*, *UHI* = *Hash* σB, *HVF* = *Mac*[σB] ⟨*ainfo*, *Num uinfo*⟩|)
definition *hfDl* **where** *hfDl* ≡ (|*AHI* = *hiDl*, *UHI* = *Hash* σD, *HVF* = *Mac*[σD] ⟨*ainfo*, *Num uinfo*⟩|)
definition *hfEl* **where** *hfEl* ≡ (|*AHI* = *hiEl*, *UHI* = *Hash* σE, *HVF* = *Mac*[σE] ⟨*ainfo*, *Num uinfo*⟩|)
definition *hfFl* **where** *hfFl* ≡ (|*AHI* = *hiFl*, *UHI* = *Hash* σF, *HVF* = *Mac*[σF] ⟨*ainfo*, *Num uinfo*⟩|)

lemmas *hfl-defs* = *hfAl-def hfBl-def hfDl-def hfEl-def hfFl-def*

lemma *e1.hf-valid ainfo uinfo hfAl None*
  by (*simp add*: *e1.hf-valid-invert hfAl-def*)
lemma *e1.hf-valid ainfo uinfo 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 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*: *TWu.holds-split-tail hfl-defs*)

lemma *forwardingpath-auth*: *pfragment ainfo forwardingpath* (*e1.auth-seg2 uinfo*)
apply(*auto simp add*: *e1.auth-seg2-def pfragment-def*)
using *forwardingpath-valid*
by(*auto intro*!: *exI*[*of* - []] *simp add*: *e1.hfs-valid-prefix-generic-def 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*→ *s1*
using *forwardingpath-auth*
by(*auto dest*!: *e1.dp2-dispatch-int-also-works-for-honest*[**where** *?m* = *pkt0*])
(*auto simp add*: *e1.dp2-def e1.dp2-defs e1.dp2-msgs forwading-states pkt0-def e1.dp2-init-def*)

lemma *s1-s2*: *e1.dp2*: *s1* −*evt-send2 nF nE pkt0*→ *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*→ *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*→ *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*→ *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*→ *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*→ *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*→ *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*→ *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*→ *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-hfs-in-ik*: *t* ∈ *e1.ik-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-hfs*
apply(*auto simp add*: *e1.ik-hfs-def e1.auth-seg2-def*
intro!: *exI*[*of - hfEl*] *exI*[*of -* [*hfFl*, *hfEl*, *hfDl*, *hfBl*, *hfAl*]] *exI*[*of - ainfo*])
using *e1.hfs-valid-prefix-generic-def no-assumptions-left.forwardingpath-valid*
by(*auto intro*!: *exI*[*of - uinfo*] *simp add*: *auth-seg0-def leftsegment-wormholed-def hfl-defs*)

lemma *uhi-e-auth*: *UHI hfEl* ∈ *e1.ik-hfs*
apply(*auto simp add*: *e1.ik-hfs-def e1.auth-seg2-def*
intro!: *exI*[*of - hfEl*] *exI*[*of -* [*hfFl*, *hfEl*, *hfDl*, *hfBl*, *hfAl*]] *exI*[*of - ainfo*])
using *e1.hfs-valid-prefix-generic-def no-assumptions-left.forwardingpath-valid*
by(*auto simp add*: *e1.auth-seg2-def 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 e1.ik-hfs-def e1.terms-pkt-def*)
using *hvf-e-auth uhi-e-auth*
by(*auto dest*: *ik-hfs-in-ik simp add*: *s-attr-def s0-def e1.dp2-init-def pkt-attr-def*)
then show *?thesis* **using** *reach-s0* **by** *auto*
qed

end
end

3.7 EPIC Level 2 in the Strong Attacker Model

theory *EPIC-L2-SA*
  imports
    ../Parametrized-Dataplane-3-directed
    ../infrastructure/Keys
begin

type-synonym *EPIC-HF* = (*unit*, *msgterm*) *HF*
type-synonym *UINFO* = *nat*

locale *epic-l2-defs* = *network-assums-direct - - - auth-seg0*
  for *auth-seg0* :: (*msgterm* × *ahi list*) *set* +
  fixes *no-oracle* :: *msgterm* ⇒ *UINFO* ⇒ *bool*
begin

3.7.1 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* ⇒ *UINFO*
    ⇒ *EPIC-HF*
    ⇒ *EPIC-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*[*K-i* (*ASID ahi*) (*Num uinfo*)] ⟨*Num ts*, *Num uinfo*, σ⟩)
| *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*[*K-i* (*ASID ahi*) (*Num uinfo*)] ⟨*Num ts*, *Num uinfo*, σ⟩)
| *hf-valid* - - - - = *False*

abbreviation *upd-uinfo* :: *nat* ⇒ *EPIC-HF* ⇒ *nat* **where**
upd-uinfo uinfo hf ≡ *uinfo*

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* = *term2if upif*, *DownIF* = *term2if downif*, *ASID* = *asid*⦈ # *extrUhi uhi2*
| *extrUhi* (*Hash* (*Mac*[*macKey asid*] (*L* [*ts*, *upif*, *downif*])))
= [⦇*UpIF* = *term2if upif*, *DownIF* = *term2if 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 *term-ainfo* :: *msgterm* ⇒ *msgterm* **where**
term-ainfo ≡ *id*

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

fun *terms-hf* :: *EPIC-HF* ⇒ *msgterm set* **where**
terms-hf hf = {*HVF hf*, *UHI hf*}

abbreviation *terms-uinfo* :: *UINFO* ⇒ *msgterm set* **where**
terms-uinfo x ≡ {}

An authenticated info field is always a number (corresponding to a timestamp). The unauthenticated info field is as well a number, representing combination of timestamp offset and SRC address.

definition *auth-restrict* **where**
auth-restrict ainfo uinfo l $\equiv$ ($\exists$ *ts. ainfo* = *Num ts*)

We now define useful properties of the above definition.

lemma *hf-valid-invert*:
hf-valid tsn uinfo hf mo $\longleftrightarrow$
(($\exists$ *ahi ahi2* σ *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* $\wedge$
σ = *Mac*[*macKey asid*] (*L* [*tsn*, *upif*, *downif*, *uhi2*]) $\wedge$
tsn = *Num ts* $\wedge$
uhi = *Hash* σ $\wedge$
x = *Mac*[*K-i* (*ASID ahi*) (*Num uinfo*)] $\langle$*tsn*, *Num uinfo*, $\sigma\rangle$)
$\vee$ ($\exists$ *ahi* σ *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* $\wedge$
σ = *Mac*[*macKey asid*] (*L* [*tsn*, *upif*, *downif*]) $\wedge$
tsn = *Num ts* $\wedge$
uhi = *Hash* σ $\wedge$
x = *Mac*[*K-i* (*ASID ahi*) (*Num uinfo*)] $\langle$*tsn*, *Num uinfo*, $\sigma\rangle$)
)
apply(*auto elim*!: *hf-valid.elims*) **using** *option.exhaust ASIF.simps* **by** *metis+*

lemma *hf-valid-auth-restrict*[*dest*]: *hf-valid ainfo uinfo hf z* $\Longrightarrow$ *auth-restrict ainfo uinfo l*
by(*auto simp add*: *hf-valid-invert auth-restrict-def*)

lemma *auth-restrict-ainfo*[*dest*]: *auth-restrict ainfo uinfo l* $\Longrightarrow$ $\exists$ *ts. ainfo* = *Num ts*
by(*auto simp add*: *auth-restrict-def*)

lemma *info-hvf*:
assumes *hf-valid ainfo uinfo m z HVF m* = *Mac*[*k-i*] $\langle$*ainfo'*, *Num uinfo'*, $\sigma\rangle$ $\vee$ *hf-valid ainfo' uinfo' m z'*
shows *uinfo* = *uinfo' ainfo'* = *ainfo*
using *assms* **by**(*auto simp add*: *hf-valid-invert*)

3.7.2 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.

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.

print-locale *dataplane-3-directed-defs*
sublocale *dataplane-3-directed-defs - - - auth-seg0 hf-valid auth-restrict extr extr-ainfo term-ainfo terms-hf terms-uinfo upd-uinfo no-oracle*
by *unfold-locales*

abbreviation *is-oracle* **where** *is-oracle ainfo t* $\equiv$ $\neg$ *no-oracle ainfo t*

declare *TWu.holds-set-list*[*dest*]
declare *TWu.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-hfs*), but to the underlying hop authenticators that are used to create them.

definition *ik-add* :: *msgterm set* **where**
ik-add ≡ { σ | *ainfo uinfo l hf* σ *k-i*.
(*ainfo*, *l*) ∈ *auth-seg2 uinfo*
∧ *hf* ∈ *set l* ∧ *HVF hf* = *Mac*[*k-i*] ⟨*ainfo*, *Num uinfo*, σ⟩ }

lemma *ik-addI*:
⟦(*ainfo*, *l*) ∈ *auth-seg2 uinfo*; *hf* ∈ *set l*; *HVF hf* = *Mac*[*k-i*] ⟨*ainfo*, *Num uinfo*, σ⟩⟧ ⟹ σ ∈ *ik-add*
apply(*auto simp add*: *ik-add-def*)
by *blast*

lemma *ik-add-form*: *t* ∈ *ik-add* ⟹ ∃ *asid l* . *t* = *Mac*[*macKey asid*] *l*

by(*auto simp add*: *ik-add-def auth-seg2-def dest*!: *TWu.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* | *t ainfo hf l uinfo* . *hf* ∈ *set l* ∧ *hfs-valid-None ainfo uinfo l* ∧
is-oracle ainfo uinfo ∧ (*ainfo*, *l*) ∉ *auth-seg2 uinfo* ∧ (*t* = *HVF hf* ∨ *t* = *UHI hf*) }

lemma *ik-oracle-parts-form*:
t ∈ *ik-oracle* ⟹
(∃ *asid l ainfo uinfo k-i* . *t* = *Mac*[*k-i*] ⟨*ainfo*, *Num uinfo*, *Mac*[*macKey asid*] *l*⟩) ∨
(∃ *asid l* . *t* = *Hash* (*Mac*[*macKey asid*] *l*))
by(*auto simp add*: *ik-oracle-def hf-valid-invert dest*!: *TWu.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* ∈ *ik-oracle* ⟷
(∃ *ainfo hf l uinfo*. *hf* ∈ *set l* ∧ *hfs-valid-None ainfo uinfo l* ∧ *is-oracle ainfo uinfo*
∧ (*ainfo*, *l*) ∉ *auth-seg2 uinfo* ∧ (*t* = *HVF hf* ∨ *t* = *UHI hf*))
by(*rule iffI*, *frule ik-oracle-parts-form*)
(*auto simp add*: *ik-oracle-def hf-valid-invert*)

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-directed* locale.

sublocale
  dataplane-3-directed-ik-defs - - - auth-seg0 terms-uinfo no-oracle hf-valid auth-restrict extr extr-ainfo term-ainfo
    terms-hf upd-uinfo ik-add ik-oracle
  by *unfold-locales*

lemma *ik-hfs-form*: $t \in$ *parts ik-hfs* $\Longrightarrow \exists\ t'$. $t =$ *Hash* t'
  by(*auto 3 4 simp add*: *auth-seg2-def hf-valid-invert*)

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

lemma *parts-ik-hfs*[*simp*]: *parts ik-hfs* = *ik-hfs*
  by (*auto intro*!: *parts-Hash ik-hfs-form*)

This lemma allows us not only to expand the definition of *ik-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-hfs-simp*:
  $t \in$ *ik-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 uinfo* . (*ainfo*, *hfs*) $\in$ *auth-seg2 uinfo*
    $\wedge$ ($\exists$ *nxt*. *hf-valid ainfo uinfo hf nxt*)))) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
  assume *asm*: *?lhs*
  then obtain *ainfo uinfo hf hfs* **where**
    dfs: *hf* $\in$ *set hfs* (*ainfo*, *hfs*) $\in$ *auth-seg2 uinfo* $t =$ *HVF hf* $\vee$ $t =$ *UHI hf*
    by(*auto simp add*: *ik-hfs-def*)
  then have *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 (*auto 3 4 simp add*: *auth-seg2-def intro*!: *ik-hfs-form exI*[*of - hf*] *exI*[*of - hfs*]
      dest: *TWu.holds-set-list-no-update*)
qed(*auto simp add*: *ik-hfs-def*)

Properties of Intruder Knowledge

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

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 auth-restrict-def ik-hfs-simp*)
    (*auto simp add*: *ik-add-def ik-oracle-def auth-seg2-def hf-valid-invert hfs-valid-prefix-generic-def*
      dest!: *TWu.holds-set-list*)

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

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-hfs-simp ik-add-def hf-valid-invert*)

Hop authenticators are agnostic to uinfo field

fun *K-i-upd* :: *msgterm* ⇒ *msgterm* ⇒ *msgterm* **where**
K-i-upd ⟨*AS asid*, -⟩ *uinfo'* = ⟨*AS asid*, *source-extract uinfo'*⟩
| *K-i-upd* - - = ε

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-change-hf* :: *UINFO* ⇒ *EPIC-HF* ⇒ *EPIC-HF* **where**
uinfo-change-hf new-uinfo hf =
(*case HVF hf of Mac*[*k-i*] ⟨*ainfo*, *Num uinfo*, σ⟩
⇒ *hf*⦇*HVF* := *Mac*[*K-i-upd k-i* (*Num new-uinfo*)] ⟨*ainfo*, *Num new-uinfo*, σ⟩⦈ | - ⇒ *hf*)

fun *uinfo-change* :: *UINFO* ⇒ *EPIC-HF list* ⇒ *EPIC-HF list* **where**
uinfo-change new-uinfo hfs = *map* (*uinfo-change-hf new-uinfo*) *hfs*

lemma *uinfo-change-valid*:
hfs-valid ainfo uinfo l nxt ⟹ *hfs-valid ainfo new-uinfo* (*uinfo-change new-uinfo l*) *nxt*
apply(*induction l nxt rule*: *TWu.holds.induct*[**where** *?upd=upd-uinfo*])
apply *auto*
subgoal for *info x y ys nxt*
by(*cases map* (*uinfo-change-hf new-uinfo*) *ys*)
(*cases info*, *auto 3 4 simp add*: *K-i-def TWu.holds-split-tail hf-valid-invert*)+
by(*auto 3 4 simp add*: *K-i-def TWu.holds-split-tail hf-valid-invert TWu.holds.simps*)

lemma *uinfo-change-hf-AHI*: *AHI* (*uinfo-change-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*) **apply** *auto*
subgoal for *x3* **apply**(*cases x3*) **by** *auto*
done
done
done
done

lemma *uinfo-change-hf-AHIS*[*simp*]: *AHIS* (*map* (*uinfo-change-hf new-uinfo*) *l*) = *AHIS l*
apply(*induction l*) **using** *uinfo-change-hf-AHI* **by** *auto*

lemma *uinfo-change-auth-seg2*:
assumes *hf-valid ainfo uinfo m z* σ = *Mac*[*Key* (*macK asid*)] *j*
HVF m = *Mac*[*k-i*] ⟨*ainfo*, *uinfo'*, σ⟩ σ ∈ *ik-add no-oracle ainfo uinfo*
shows ∃ *hfs*. *m* ∈ *set hfs* ∧ (∃ *uinfo''*. (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo''*)
proof−
from *assms*(*4*) **obtain** *ainfo-add uinfo-add l-add hf-add k-i-add* **where**
(*ainfo-add*, *l-add*) ∈ *auth-seg2 uinfo-add hf-add* ∈ *set l-add*
HVF hf-add = *Mac*[*k-i-add*] ⟨*ainfo-add*, *Num uinfo-add*, σ⟩
by(*auto simp add*: *ik-add-def*)
then have *add*: *m* ∈ *set* (*uinfo-change uinfo l-add*) (*ainfo-add*, (*uinfo-change uinfo l-add*)) ∈ *auth-seg2 uinfo*

using *assms*(*1*−*3*,*5*) **apply**(*auto simp add*: *auth-seg2-def simp del*: *AHIS-def*)
apply(*auto simp add*: *hf-valid-invert intro*!: *image-eqI dest*!: *TWu.holds-set-list*)[*1*]
apply(*auto simp add*: *auth-restrict-def intro*!: *exI elim*: *ahi-eq dest*: *uinfo-change-valid simp del*: *AHIS-def*)
by(*auto simp add*: *hf-valid-invert K-i-def dest*!: *TWu.holds-set-list-no-update*)
then have *ainfo-add* = *ainfo*
using *assms*(*1*) **by**(*auto simp add*: *auth-seg2-def dest*!: *TWu.holds-set-list dest*: *info-hvf*)
then show *?thesis* **using** *add* **by** *fastforce*
qed

lemma *MAC-synth-oracle*:
assumes *hf-valid ainfo uinfo 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*!: *TWu.holds-set-list-no-update*)

lemma *ik-oracle-is-oracle*:
⟦*Mac*[σ] ⟨*ainfo*, *Num uinfo*⟩ ∈ *ik-oracle*⟧ ⟹ *is-oracle ainfo uinfo*
by (*auto simp add*: *ik-oracle-def dest*: *info-hvf*)
(*auto dest*!: *TWu.holds-set-list-no-update simp add*: *hf-valid-invert*)

lemma *MAC-synth-helper*:
⟦*hf-valid ainfo uinfo m z*; *no-oracle ainfo uinfo*;
HVF m = *Mac*[*k-i*] ⟨*ainfo*, *Num uinfo*, σ⟩; σ = *Mac*[*Key* (*macK asid*)] *j*; σ ∈ *ik* ∨ *HVF m* ∈ *ik*⟧
⟹ ∃ *hfs*. *m* ∈ *set hfs* ∧ (∃ *uinfo*′. (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*′)
apply(*auto simp add*: *ik-def ik-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-change-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 m z HVF m* ∈ *synth ik mac-format* (*HVF m*) *asid*
asid ∉ *bad checkInfo ainfo no-oracle ainfo uinfo*
shows ∃ *hfs* . *m* ∈ *set hfs* ∧ (∃ *uinfo*′. (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*′)

using *assms*
apply(*auto simp add*: *mac-format-def elim*!: *MAC-synth-helper dest*!: *key-ik-bad*)
apply(*auto simp add*: *ik-def ik-hfs-simp dest*: *ik-add-form dest*!: *ik-oracle-parts-form*)
using *assms*(*1*) **by**(*auto dest*: *info-hvf simp add*: *hf-valid-invert*)

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

lemma *COND-honest-hf-analz*:
assumes *ASID* (*AHI hf*) $\notin$ *bad hf-valid ainfo uinfo hf nxt terms-hf hf* $\subseteq$ *synth* (*analz ik*)
no-oracle ainfo uinfo
shows *terms-hf hf* $\subseteq$ *analz ik*
proof−
let *?asid* = *ASID* (*AHI hf*)
from *assms*(*3*) **have** *hf-synth-ik*: *HVF hf* $\in$ *synth ik UHI hf* $\in$ *synth ik* **by** *auto*
from *assms*(*2*) **have** *mac-format* (*HVF hf*) *?asid*
by(*auto simp add*: *mac-format-def hf-valid-invert*)
then obtain *hfs uinfo* **where** *hf* $\in$ *set hfs* (*ainfo*, *hfs*) $\in$ *auth-seg2 uinfo*
using *assms*(*1*,*2*,*4*) *hf-synth-ik* **by**(*auto dest*!: *MAC-synth*)
then have *HVF hf* $\in$ *ik UHI hf* $\in$ *ik*
using *assms*(*2*)
by(*auto simp add*: *ik-hfs-def intro*!: *ik-ik-hfs intro*!: *exI*)
then show *?thesis* **by** *auto*
qed

lemma *COND-terms-hf*:
assumes *hf-valid ainfo uinfo hf z* **and** *HVF hf* $\in$ *ik* **and** *no-oracle ainfo uinfo*
shows $\exists$ *hfs*. *hf* $\in$ *set hfs* $\wedge$ ($\exists$ *uinfo* . (*ainfo*, *hfs*) $\in$ *auth-seg2 uinfo*)
proof−
obtain *hfs ainfo* **where** *hfs-def*: *hf* $\in$ *set hfs* (*ainfo*, *hfs*) $\in$ *auth-seg2 uinfo*
using *assms* **apply**(*auto 3 4 simp add*: *K-i-def hf-valid-invert ik-hfs-simp ik-def dest*: *ahi-eq*
dest!: *ik-oracle-is-oracle ik-add-form*)
using *MAC-synth-oracle assms*(*1*) **by** *force*+
then obtain *hfs ainfo* **where** *hfs-def*: *hf* $\in$ *set hfs* (*ainfo*, *hfs*) $\in$ *auth-seg2 uinfo* **by** *auto*
show *?thesis*
using *hfs-def* **apply** (*auto simp add*: *auth-seg2-def dest*!: *TWu.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 l nxt*; *nxt* = *None*⟧ $\Longrightarrow$ *prefix* (*extr-from-hd l*) (*AHIS l*)
by(*induction l nxt rule*: *TWu.holds.induct*[**where** *?upd*=*upd-uinfo*])
(*auto simp add*: *K-i-def TWu.holds-split-tail TWu.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 l nxt*))
(*extr-from-hd l*)
apply(*induction l nxt rule*: *TWu.takeW.induct*[**where** *?Pa*=*hf-valid ainfo*,**where** *?upd*=*upd-uinfo*])
by(*auto simp add*: *TWu.takeW-split-tail TWu.takeW.simps*(*1*))
(*auto 3 4 simp add*: *hf-valid-invert intro*!: *ahi-eq elim*: *ASIF.elims*)

lemma *COND-hf-valid-uinfo*:
⟦*hf-valid ainfo uinfo hf nxt*; *hf-valid ainfo′ uinfo′ hf nxt′*⟧ $\Longrightarrow$ *uinfo′* = *uinfo*

by(*auto dest*: *info-hvf*)

lemma *COND-upd-uinfo-ik*:
⟦*terms-uinfo uinfo* ⊆ *synth* (*analz ik*); *terms-hf hf* ⊆ *synth* (*analz ik*)⟧
⟹ *terms-uinfo* (*upd-uinfo uinfo hf*) ⊆ *synth* (*analz ik*)
by (*auto*)

lemma *COND-upd-uinfo-no-oracle*:
no-oracle ainfo uinfo ⟹ *no-oracle ainfo* (*upd-uinfo uinfo fld*)
by (*auto*)

lemma *COND-auth-restrict-upd*:
auth-restrict ainfo uinfo (*x*#*y*#*hfs*)
⟹ *auth-restrict ainfo* (*upd-uinfo uinfo y*) (*y*#*hfs*)
by (*auto simp add*: *auth-restrict-def*)

3.7.5 Instantiation of *dataplane-3-directed* locale

print-locale *dataplane-3-directed*
sublocale
dataplane-3-directed - - - auth-seg0 terms-uinfo terms-hf hf-valid auth-restrict extr extr-ainfo term-ainfo

upd-uinfo ik-add
ik-oracle no-oracle
apply *unfold-locales*
using *COND-terms-hf COND-honest-hf-analz COND-extr-prefix-path*
COND-path-prefix-extr COND-hf-valid-uinfo COND-upd-uinfo-ik COND-upd-uinfo-no-oracle
COND-auth-restrict-upd **by** *auto*

end
end

3.8 Abstract XOR

theory *Abstract-XOR*
  imports
    HOL.Finite-Set HOL−Library.FSet Message

begin

3.8.1 Abstract XOR definition and lemmas

We model xor as an operation on finite sets (fset). {||} is defined as the identity element.

xor of two fsets is the symmetric difference

definition *xor* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **where**
  xor xs ys = (*xs* |∪| *ys*) |−| (*xs* |∩| *ys*)

lemma *xor-singleton*:
  xor xs {| *z* |} = (*if z* |∈| *xs then xs* |−| {| *z* |} *else finsert z xs*)
  xor {| *z* |} *xs* = (*if z* |∈| *xs then xs* |−| {| *z* |} *else finsert z xs*)
  by (*auto simp add*: *xor-def*)

declare *finsertCI*[*rule del*]
declare *finsertCI*[*intro*]

lemma *xor-assoc*: *xor* (*xor xs ys*) *zs* = *xor xs* (*xor ys zs*)
  by (*auto simp add*: *xor-def*)

lemma *xor-commut*: *xor xs ys* = *xor ys xs*
  by (*auto simp add*: *xor-def*)

lemma *xor-self-inv*: ⟦*xor xs ys* = *zs*; *xs* = *ys*⟧ ⟹ *zs* = {||}
  by (*auto simp add*: *xor-def*)

lemma *xor-self-inv'*: *xor xs xs* = {||}
  by (*auto simp add*: *xor-def*)

lemma *xor-self-inv''*[*dest!*]: *xor xs ys* = {||} ⟹ *xs* = *ys*
  by (*auto simp add*: *xor-def*)

lemma *xor-identity1*[*simp*]: *xor xs* {||} = *xs*
  by (*auto simp add*: *xor-def*)

lemma *xor-identity2*[*simp*]: *xor* {||} *xs* = *xs*
  by (*auto simp add*: *xor-def*)

lemma *xor-in*: *z* |∈| *xs* ⟹ *z* |∉| (*xor xs* {| *z* |})
  by (*auto simp add*: *xor-singleton*)

lemma *xor-out*: *z* |∉| *xs* ⟹ *z* |∈| (*xor xs* {| *z* |})
  by (*auto simp add*: *xor-singleton*)

lemma *xor-elem1*[*dest*]: ⟦*x* ∈ *fset* (*xor X Y*); *x* |∉| *X*⟧ ⟹ *x* |∈| *Y*

by(*auto simp add*: *xor-def*)

lemma *xor-elem2*[*dest*]: $[\![x \in fset\ (xor\ X\ Y);\ x\ |\notin|\ Y]\!] \Longrightarrow x\ |\in|\ X$
by(*auto simp add*: *xor-def*)

lemma *xor-finsert-self*: $xor\ (finsert\ x\ xs)\ \{|x|\} = xs - \{|\ x\ |\}$
by(*auto simp add*: *xor-def*)

3.8.2 Lemmas refering to XOR and msgterm

lemma *FS-contains-elem*:
assumes $elem = f\ (FS\ zs\text{-}s)\ zs\text{-}s = xor\ zs\text{-}b\ \{|\ elem\ |\}\ \bigwedge x.\ size\ (f\ x) > size\ x$
shows $elem \in fset\ zs\text{-}b$
using *assms*(*1*)
apply(*auto simp add*: *xor-def*)
using *FS-mono assms xor-singleton*(*1*)
by (*metis*)

lemma *FS-is-finsert-elem*:
assumes $elem = f\ (FS\ zs\text{-}s)\ zs\text{-}s = xor\ zs\text{-}b\ \{|\ elem\ |\}\ \bigwedge x.\ size\ (f\ x) > size\ x$
shows $zs\text{-}b = finsert\ elem\ zs\text{-}s$
using *assms FS-contains-elem finsert-fminus xor-singleton*(*1*) *FS-mono*
by (*metis FS-mono*)

lemma *FS-update-eq*:
assumes $xs = f\ (FS\ (xor\ zs\ \{|xs|\}))$
and $ys = g\ (FS\ (xor\ zs\ \{|ys|\}))$
and $\bigwedge x.\ size\ (f\ x) > size\ x$
and $\bigwedge x.\ size\ (g\ x) > size\ x$
shows $xs = ys$
proof(*rule ccontr*)
assume *elem-neq*: $xs \neq ys$
obtain *zs-s1 zs-s2* **where** *zs-defs*:
$zs\text{-}s1 = xor\ zs\ \{|xs|\}\ zs\text{-}s2 = xor\ zs\ \{|ys|\}$ **by** *simp*
have *elems-contained-zs*: $xs \in fset\ zs\ ys \in fset\ zs$
using *assms FS-contains-elem* **by** *blast*+
then have *elems-elem*: $ys\ |\in|\ zs\text{-}s1\ xs\ |\in|\ zs\text{-}s2$
using *elem-neq* **by**(*auto simp add*: *xor-def zs-defs*)
have *zs-finsert*: $finsert\ xs\ zs\text{-}s2 = zs\text{-}s2\ finsert\ ys\ zs\text{-}s1 = zs\text{-}s1$
using *elems-elem* **by** *fastforce*+
have *f1*: $\forall m\ f\ fa.\ \neg\ sum\ fa\ (fset\ (finsert\ (m{::}msgterm)\ f)) < (fa\ m{::}nat)$
by (*simp add*: *sum.insert-remove*)
from *assms*(*1*−*2*) **have** $size\ xs > size\ (f\ (FS\ \{|\ ys\ |\}))\ size\ ys > size\ (g\ (FS\ \{|\ xs\ |\}))$
apply(*simp-all add*: *zs-defs*[*symmetric*])
using *zs-finsert f1* **by** (*metis* (*no-types*) *add-Suc-right assms*(*3*−*4*) *dual-order.strict-trans less-add-Suc1 msgterm.size*(*17*) *not-less-eq size-fset-simps*)+
then show *False* **using** *assms*(*3*,*4*) *elems-elem*
by (*metis add.right-neutral add-Suc-right f1 less-add-Suc1 msgterm.size*(*17*) *not-less-eq not-less-iff-gr-or-eq order.strict-trans size-fset-simps*)
qed

declare *fminusE*[*rule del*]
declare *finsertCI*[*rule del*]

declare *fminusE*[*elim*]
declare *finsertCI*[*intro*]

lemma *fset-size-le*:
  assumes $x \in$ *fset xs*
  shows *size* $x <$ *Suc* ($\sum x{\in}$*fset xs*. *Suc* (*size* x))
proof−
  have *size* $x \leq$ ($\sum x{\in}$*fset xs*. *size* x) **using** *assms*
    by (*auto intro*: *member-le-sum*)
  moreover have ($\sum x{\in}$*fset xs*. *size* x) $<$ ($\sum x{\in}$*fset xs*. *Suc* (*size* x))
    by (*metis assms empty-iff finite-fset lessI sum-strict-mono*)
  ultimately show *?thesis* **by** *auto*
qed

We can show that xor is a commutative function.

locale *abstract-xor*
begin
sublocale *comp-fun-commute xor*
  by(*auto simp add*: *comp-fun-commute-def xor-def*)

end
end

3.9 Anapaya-SCION

This is the "new" SCION protocol, as specified on the website of Anapaya: https://scion.docs.anapaya.net/en/latest/protocols/scion-header.html (Accessed 2021-03-02). It does not use the next hop field in its MAC computation, but instead refers uses a mutable uinfo field which acts as an XOR-based accumulator for all upstream MACs.

This protocol instance requires the use of the extensions of our formalization that provide mutable uinfo field and an XOR abstraction.

theory *Anapaya-SCION*
  imports
    ../Parametrized-Dataplane-3-directed
    ../infrastructure/Abstract-XOR
begin

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

sublocale *comp-fun-commute xor*
  by(*auto simp add*: *comp-fun-commute-def xor-def*)

3.9.1 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 unauthenticated info field and the hop field to be validated. The next hop field is not used in this instance.

fun *hf-valid* :: *msgterm* ⇒ *msgterm fset*
    ⇒ *SCION-HF*
    ⇒ *SCION-HF option* ⇒ *bool* **where**
  hf-valid (*Num ts*) *uinfo* (|*AHI* = *ahi*, *UHI* = -, *HVF* = *x*|) *nxt* ⟷
    (∃ *upif downif*. *x* = *Mac*[*macKey* (*ASID ahi*)] (*L* [*Num ts*, *upif*, *downif*, *FS uinfo*]) ∧
        ASIF (*DownIF ahi*) *downif* ∧ *ASIF* (*UpIF ahi*) *upif*)
| *hf-valid* - - - - = *False*

Updating the uinfo field involves XORin the current hop validation field onto it. Note that in all authorized segments, the hvf will already have been contained in segid, hence this operation only removes terms from the fset in the forwarding of honestly created packets.

definition *upd-uinfo* :: *msgterm fset* ⇒ *SCION-HF* ⇒ *msgterm fset* **where**
  upd-uinfo segid hf = *xor segid* {| *HVF hf* |}

declare *upd-uinfo-def*[*simp*]

The following lemma is needed to show the termination of extr, defined below.

lemma *extr-helper*:
⟦*x* = *Mac*[*macKey asid'a*] (*L* [*ts*, *upif'a*, *downif'a*, *FS segid'*]);
  fcard segid' = *fcard* (*xor segid* {|*x*|}); *x* |∈| *segid*⟧

```
      ⟹ (case x of Hash ⟨Key (macK asid), L []⟩ ⇒ 0 | Hash ⟨Key (macK asid), L [ts]⟩ ⇒ 0
| Hash ⟨Key (macK asid), L [ts, upif]⟩ ⇒ 0 | Hash ⟨Key (macK asid), L [ts, upif, downif]⟩ ⇒ 0
          | Hash ⟨Key (macK asid), L [ts, upif, downif, FS segid]⟩ ⇒ Suc (fcard segid)
| Hash ⟨Key (macK asid), L (ts # upif # downif # FS segid # ac # lista)⟩ ⇒ 0
          | Hash ⟨Key (macK asid), L (ts # upif # downif # - # list)⟩ ⇒ 0
| Hash ⟨Key (macK asid), -⟩ ⇒ 0 | Hash ⟨Key -, msgterm2⟩ ⇒ 0 | Hash ⟨-, msgterm2⟩ ⇒ 0
| Hash - ⇒ 0 | - ⇒ 0)
         < Suc (fcard segid)
  apply auto
  using fcard-fminus1-less xor-singleton(1) by (metis)
```

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.

```
function (sequential) extr :: msgterm ⇒ ahi list where
  extr (Mac[macKey asid] (L [ts, upif, downif, FS segid]))
 = (|UpIF = term2if upif, DownIF = term2if downif, ASID = asid|) # (if (∃ nextmac asid' upif'
downif' segid'.
        segid' = xor segid {| nextmac |} ∧
        nextmac = Mac[macKey asid'] (L [ts, upif', downif', FS segid']))
    then extr (THE nextmac. (∃ asid' upif' downif' segid'.
                        segid' = xor segid {| nextmac |} ∧
                        nextmac = Mac[macKey asid'] (L [ts, upif', downif', FS segid'])))
    else [])
| extr - = []
  by pat-completeness auto
termination
  apply (relation measure (λx. (case x of Mac[macKey asid] (L [ts, upif, downif, FS segid])
                 ⇒ Suc (fcard segid)
              | - ⇒ 0)))
  apply auto
  apply(rule theI2)
  by(auto elim: FS-update-eq elim!: FS-contains-elem intro!: extr-helper)
```

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 term-ainfo :: msgterm ⇒ msgterm where
  term-ainfo ≡ id
```

The ainfo field must be a Num, since it represents the timestamp (this is only needed for authorized segments (ainfo, []), since for all other segments, *hf-valid* enforces this.

Furthermore, we require that the last hop field on l has a MAC that is computed with the empty uinfo field. This restriction cannot be introduced via *hf-valid*, since it is not a check performed by the on-path routers, but rather results from the way that authorized paths are set up on the control plane. We need this restriction to ensure that the uinfo field of the top node does not contain extra terms (e.g. secret keys).

```
definition auth-restrict where
  auth-restrict ainfo uinfo l ≡
   (∃ ts. ainfo = Num ts)
```

$\wedge$ (*case l of* [] $\Rightarrow$ (*uinfo* = {||}) |
\- $\Rightarrow$ *hf-valid ainfo* {||} (*last l*) *None*)

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 *terms-hf* :: *SCION-HF* $\Rightarrow$ *msgterm set* **where**
terms-hf hf = {*HVF hf*}

When analyzing a uinfo field (which is an fset of message terms), the attacker learns all elements of the fset.

abbreviation *terms-uinfo* :: *msgterm fset* $\Rightarrow$ *msgterm set* **where**
terms-uinfo $\equiv$ *fset*

abbreviation *no-oracle* :: *'ainfo* $\Rightarrow$ *msgterm fset* $\Rightarrow$ *bool* **where** *no-oracle* $\equiv$ (λ - -. *True*)

Properties following from definitions

We now define useful properties of the above definition.

lemma *hf-valid-invert*:
hf-valid tsn uinfo hf nxt $\longleftrightarrow$
((∃ *ahi ts upif downif asid x*.
hf = (|*AHI* = *ahi*, *UHI* = (), *HVF* = *x*|) $\wedge$
ASID ahi = *asid* $\wedge$ *ASIF* (*DownIF ahi*) *downif* $\wedge$ *ASIF* (*UpIF ahi*) *upif* $\wedge$
x = *Mac*[*macKey asid*] (*L* [*tsn*, *upif*, *downif*, *FS uinfo*]) $\wedge$
tsn = *Num ts*)
)
by(*auto elim*!: *hf-valid.elims*)

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

3.9.2 Definitions and properties of the added intruder knowledge

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

print-locale *dataplane-3-directed-defs*
sublocale *dataplane-3-directed-defs - - - auth-seg0 hf-valid auth-restrict extr extr-ainfo term-ainfo terms-hf terms-uinfo upd-uinfo no-oracle*
by *unfold-locales*
declare *TWu.holds-set-list*[*dest*]
declare *TWu.holds-takeW-is-identity*[*simp*]
declare *parts-singleton*[*dest*]

abbreviation *ik-add* :: *msgterm set* **where** *ik-add* $\equiv$ {}

abbreviation *ik-oracle* :: *msgterm set* **where** *ik-oracle* $\equiv$ {}

3.9.3 Properties of the intruder knowledge, including *fset*.

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

print-locale *dataplane-3-directed-ik-defs*
sublocale
dataplane-3-directed-ik-defs - - - auth-seg0 terms-uinfo no-oracle hf-valid auth-restrict extr extr-ainfo term-ainfo
terms-hf upd-uinfo ik-add ik-oracle
by *unfold-locales*

For this instance model, the neighboring hop field is irrelevant. Hence, if we are interested in establishing the first hop field's validity given *hfs-valid*, we do not need to make a case distinction on the rest of the hop fields (which would normally be required by *TWu*.

lemma *hfs-valid-first*[*elim*]: *hfs-valid ainfo uinfo* (*hf* # *post*) *nxt* ⟹ *hf-valid ainfo uinfo hf nxt′*
by(*cases post*, *auto simp add*: *hf-valid-invert TWu.holds.simps*)

Properties of HVF of valid hop fields that fulfill the restriction.

lemma *auth-properties*:
assumes *hf* ∈ *set hfs hfs-valid ainfo uinfo hfs nxt auth-restrict ainfo uinfo hfs*
t = *HVF hf*
shows (∃ *t′* . *t* = *Hash t′*)
∧ (∃ *uinfo′*. *auth-restrict ainfo uinfo′ hfs*
∧ (∃ *nxt*. *hf-valid ainfo uinfo′ hf nxt*))
using *assms*
proof(*induction uinfo hfs nxt arbitrary*: *hf rule*: *TWu.holds.induct*[**where** *?upd=upd-uinfo*])
case (*1 info x y ys nxt*)
then show *?case*
proof(*cases hf* = *x*)
case *True*
then show *?thesis*

using *1*(*2*−*5*) **by** (*auto simp add*: *TWu.holds.simps*(*1*) *hf-valid-invert*)
next
case *False*
then have *hf* ∈ *set* (*y* # *ys*) **using** *1* **by** *auto*
then show *?thesis*
apply− **apply**(*drule 1*)
subgoal using *assms 1*(*2*−*5*) **by** (*simp add*: *TWu.holds.simps*(*1*))
using *assms*(*3*) *1*(*2*−*5*) *False*
by(*auto simp add*: *auth-restrict-def hf-valid-invert*)
qed
qed(*auto simp add*: *auth-restrict-def hf-valid-invert intro*!: *exI*)

lemma *ik-hfs-form*: *t* ∈ *parts ik-hfs* ⟹ ∃ *t′* . *t* = *Hash t′*
by(*auto 3 4 simp add*: *auth-seg2-def dest*: *auth-properties*)

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

lemma *parts-ik-hfs*[*simp*]: *parts ik-hfs* = *ik-hfs*
by (*auto intro*!: *parts-Hash ik-hfs-form*)

This lemma allows us not only to expand the definition of *ik-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-hfs-simp*:
  $t \in$ *ik-hfs* $\longleftrightarrow$ ($\exists$ *t'* . *t* = *Hash t'*) $\wedge$ ($\exists$ *hf* . *t* = *HVF hf*
      $\wedge$ ($\exists$ *hfs uinfo*. *hf* $\in$ *set hfs* $\wedge$ ($\exists$ *ainfo* . (*ainfo*, *hfs*) $\in$ (*auth-seg2 uinfo*)
      $\wedge$ ($\exists$ *nxt uinfo'*. *hf-valid ainfo uinfo' hf nxt*)))) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
  assume *asm*: *?lhs*
  then obtain *ainfo uinfo hf hfs* **where**
    dfs: *hf* $\in$ *set hfs* (*ainfo*, *hfs*) $\in$ (*auth-seg2 uinfo*) *t* = *HVF hf*
    by(*auto simp add*: *ik-hfs-def*)
  then have *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-hfs-form*)
qed(*auto simp add*: *ik-hfs-def*)

The following lemma is one of the conditions. We already prove it here, since it is helpful elsewhere.

lemma *auth-restrict-upd*:
    auth-restrict ainfo uinfo (*x#y#hfs*)
  $\Longrightarrow$ *auth-restrict ainfo* (*upd-uinfo uinfo y*) (*y#hfs*)
  by (*auto simp add*: *auth-restrict-def*)

We now show that *ik-uinfo* is redundant, since all of its terms are already contained in *ik-hfs*. To this end, we first show that a term contained in the uinfo field of an authorized paths is also contained in the HVF of the same path.

lemma *uinfo-contained-in-HVF*:
  assumes $t \in$ *fset uinfo* (*ainfo*, *hfs*) $\in$ (*auth-seg2 uinfo*)
  shows $\exists$ *hf*. *t* = *HVF hf* $\wedge$ *hf* $\in$ *set hfs*
proof−
  from *assms*(*2*) **have** *hfs-defs*: *hfs-valid-None ainfo uinfo hfs auth-restrict ainfo uinfo hfs*
    by(*auto simp add*: *auth-seg2-def*)
  obtain *nxt*::*SCION-HF option* **where** *nxt-None*[*intro*]: *nxt* = *None* **by** *simp*
  then show *?thesis* **using** *hfs-defs assms*(*1*)
  proof(*induction uinfo hfs nxt rule*: *TWu.holds.induct*[**where** *?upd*=*upd-uinfo*])
    case (*1 info x y ys nxt*)
    then have *hf-valid-x*: *hf-valid ainfo info x* (*Some y*) **by**(*auto simp only*: *TWu.holds.simps*)
    from *1*(*2*−*3*,*5*) **show** *?case*
    proof(*cases t* = *HVF y*)
      case *False*
      then have $t \in$ *fset* (*upd-uinfo info y*) **using** *1*(*2*,*5*) **by** (*simp add*: *xor-singleton*(*1*))
      moreover have *hfs-valid-None ainfo* (*upd-uinfo info y*) (*y # ys*)
          auth-restrict ainfo (*upd-uinfo info y*) (*y # ys*)
        using *1*(*3*−*4*) **by**(*auto simp only*: *TWu.holds.simps elim*: *auth-restrict-upd*)
      ultimately have $\exists$ *hf*. *t* = *HVF hf* $\wedge$ *hf* $\in$ *set* (*y # ys*) **using** *1*(*1*) *1.prems*(*1*) **by** *blast*
      then show *?thesis* **by** *auto*
    qed(*auto*)
  next
    case (*2 info x nxt*)
    then show *?case*
      apply(*auto simp only*: *TWu.holds.simps auth-restrict-def*)

    by (*auto simp add*: *hf-valid-invert*)
  next
    case (*3 info nxt*)
    then show *?case* **by**(*auto simp add*: *auth-restrict-def*)
  qed
qed

The following lemma allows us to ignore *ik-uinfo* when we unfold *ik*.

lemma *ik-uinfo-in-ik-hfs*: $t \in$ *ik-uinfo* $\Longrightarrow t \in$ *ik-hfs*
  by(*auto simp add*: *ik-hfs-def dest*!: *uinfo-contained-in-HVF*)

Properties of Intruder Knowledge

lemma *auth-ainfo*[*dest*]: ⟦(*ainfo*, *hfs*) ∈ (*auth-seg2 uinfo*)⟧ ⟹ ∃ *ts* . *ainfo* = *Num ts*
  by(*auto simp add*: *auth-seg2-def auth-restrict-def*)

This lemma unfolds the definition of the intruder knowledge but also already applies some simplifications, such as ignoring *ik-uinfo*.

lemma *ik-simpler*:
  ik = *ik-hfs*
    ∪ {*term-ainfo ainfo* | *ainfo hfs uinfo*. (*ainfo*, *hfs*) ∈ (*auth-seg2 uinfo*)}
    ∪ *Key*‘(*macK*‘*bad*)
  by(*auto simp add*: *ik-def simp del*: *ik-uinfo-def dest*: *ik-uinfo-in-ik-hfs*)

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-simpler auth-seg2-def ik-hfs-simp auth-restrict-def*)

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

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

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

3.9.4 Lemmas helping with conditions relating to extract

Resolve the definite descriptor operator THE.

lemma *THE-nextmac*:
assumes *hvf* = *Mac*[*macKey askey*] (*L* [*Num ts*, *upif*, *downif*, *FS* (*xor info* {|*hvf*|})])
shows (*THE nextmac*. ∃ *asid' upif' downif'*.
nextmac = *Mac*[*macKey asid'*] (*L* [*Num ts*, *upif'*, *downif'*, *FS* (*xor info* {|*nextmac*|})]))
= *hvf*
apply(*rule theI2*[*of - hvf*])
using *assms*
by(*auto elim*!: *FS-update-eq*[*of - - info hvf*])

lemma *hf-valid-uinfo*:
assumes *hf-valid ainfo* (*upd-uinfo uinfo y*) *y nxt hvfy* = *HVF y*
shows *hvfy* ∈ *fset uinfo*
apply (*cases y*)
using *assms* **by**(*auto simp add*: *hf-valid-invert elim*!: *FS-contains-elem*)

A single step of extract. Extract on a single valid hop field is equivalent to that hop field's hop info field concat extract on the next hop field, where the next hop field has to be valid with uinfo updated.

lemma *extr-hf-valid*:
assumes *hf-valid ainfo uinfo x nxt hf-valid ainfo* (*upd-uinfo uinfo y*) *y nxt'*
shows *extr* (*HVF x*) = *AHI x* # *extr* (*HVF y*)
proof−
obtain *uinfo'* **where** *info'-def*: *uinfo'* = *xor uinfo* {|*HVF y*|} **by** *simp*
obtain *ts ahi upif downif hvfx ahi' upif' downif' hvfy* **where** *unfolded-defs*:
x = (|*AHI* = *ahi*, *UHI* = (), *HVF* = *hvfx*|)
ASIF (*UpIF ahi*) *upif*
ASIF (*DownIF ahi*) *downif*
hvfx = *Mac*[*macKey* (*ASID ahi*)] (*L* [*Num ts*, *upif*, *downif*, *FS uinfo*])
y = (|*AHI* = *ahi'*, *UHI* = (), *HVF* = *hvfy*|)
ASIF (*UpIF ahi'*) *upif'*
ASIF (*DownIF ahi'*) *downif'*
hvfy = *Mac*[*macKey* (*ASID ahi'*)] (*L* [*Num ts*, *upif'*, *downif'*, *FS* (*uinfo'*)])
using *assms* **apply**(*auto simp only*: *hf-valid-invert*) **by** (*auto simp add*: *info'-def*)
have *hvfy-in-uinfo*: *hvfy* ∈ *fset uinfo*
using *assms*(*2*) **apply**(*auto intro*!: *hf-valid-uinfo*) **using** *unfolded-defs* **by** *simp*
then obtain *fcard-uinfo-minus1* **where** *fcard uinfo* = *Suc fcard-uinfo-minus1*

by (*metis fcard-Suc-fminus1*)
then show *?thesis*
apply(*cases y*)
using *unfolded-defs*(*1−7*) **apply** (*auto intro*!: *ahi-eq*)
subgoal for *nextmac*
apply(*auto simp add*: *THE-nextmac dest*!: *FS-update-eq*[*of nextmac -*
- hvfy (*λx. Mac*[*macKey* (*ASID ahi′*)] (*L* [*Num ts*, *upif′*, *downif′*, *x*]))])
using *unfolded-defs*(*8*) *info′-def* **by** *fastforce+*
using *hvfy-in-uinfo unfolded-defs*(*8*) *info′-def*
by (*fastforce dest*!: *fcard-Suc-fminus1*[*simplified*] *elim*!: *allE*[*of - HVF y*])
qed

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

lemma *COND-honest-hf-analz*:
assumes *ASID* (*AHI hf*) ∉ *bad hf-valid ainfo uinfo hf nxt terms-hf hf* ⊆ *synth* (*analz ik*)
no-oracle ainfo uinfo
shows *terms-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*
by(*auto simp add*: *mac-format-def hf-valid-invert*)
then obtain *hfs uinfo′* **where**
hf ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo′*
using *assms*(*1*,*2*) *hf-synth-ik* **by**(*auto dest*!: *MAC-synth*)
then have *HVF hf* ∈ *ik*
using *assms*(*2*)
by(*auto simp add*: *ik-hfs-def intro*!: *ik-ik-hfs intro*!: *exI*)
then show *?thesis* **by** *auto*
qed

lemma *COND-terms-hf*:
assumes *hf-valid ainfo uinfo hf nxt terms-hf hf* ⊆ *analz ik*
no-oracle ainfo uinfo
shows ∃*hfs. hf* ∈ *set hfs* ∧ (∃*uinfo′* . (*ainfo*, *hfs*) ∈ (*auth-seg2 uinfo′*))
proof−
obtain *hfs ainfo uinfo* **where** *hfs-def*: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*
using *assms*
by(*simp only*: *analz-parts-ik parts-ik*)
(*auto 3 4 simp add*: *hf-valid-invert ik-hfs-simp ik-simpler dest*: *ahi-eq*)
show *?thesis*
using *hfs-def* **apply** (*auto simp add*: *auth-seg2-def dest*!: *TWu.holds-set-list*)
using *hfs-def assms*(*1*) **by** (*auto simp add*: *auth-seg2-def dest*: *info-hvf*)
qed

lemmas *COND-auth-restrict-upd* = *auth-restrict-upd*

lemma *COND-extr-prefix-path*:
⟦*hfs-valid ainfo uinfo l nxt*; *auth-restrict ainfo uinfo l*⟧ ⟹ *prefix* (*extr-from-hd l*) (*AHIS l*)
proof(*induction l nxt rule*: *TWu.holds.induct*[**where** *?upd*=*upd-uinfo*])
case (*1 info x y ys nxt*)
from *1*(*2−3*) **have** *hfs-valid*:

hfs-valid ainfo (*upd-uinfo info y*) (*y* # *ys*) *nxt*
auth-restrict ainfo (*upd-uinfo info y*) (*y* # *ys*)
by(*auto simp only*: *TWu.holds.simps intro*!: *auth-restrict-upd*)
then have *prefix-y*: *prefix* (*extr-from-hd* (*y* # *ys*)) (*AHIS* (*y* # *ys*)) **by**(*rule 1*(*1*))
have *extr-from-hd* (*x* # *y* # *ys*) = *AHI x* # *extr-from-hd* (*y* # *ys*)
apply(*cases ys*)
using *1*(*2*) **by**(*auto simp only*: *extr-from-hd.simps TWu.holds.simps intro*!: *extr-hf-valid*)
then show *?case*
using *prefix-y* **by** (*auto*)
qed(*auto simp only*: *TWu.holds.simps hf-valid-invert*,
auto simp add: *fcard-fempty auth-restrict-def intro*!: *ahi-eq dest*!: *FS-contr*)

lemma *COND-path-prefix-extr*:
prefix (*AHIS* (*hfs-valid-prefix ainfo uinfo l nxt*))
(*extr-from-hd l*)
proof(*induction l nxt rule*: *TWu.takeW.induct*[**where** *?Pa=hf-valid ainfo*, **where** *?upd=upd-uinfo*])
case (*2 info x xo*)
then show *?case*
apply(*cases fcard info*)
by(*auto 3 4 intro*!: *ahi-eq simp add*: *fcard-fempty TWu.takeW-split-tail TWu.takeW.simps*(*1*) *hf-valid-invert*)
next
case (*4 info x y xs xo*)
from *4*(*1*) **show** *?case*
proof (*cases hf-valid ainfo* (*upd-uinfo info y*) *y* (*head xs*))
case *hf-valid-y*: *True*
obtain *info′* **where** *info′-def*: *info′* = *xor info* {|*HVF y*|} **by** *simp*
show *?thesis*
proof(*rule prefix-cons*[**where** *?ys=extr-from-hd* (*y* # *xs*), **where** *?x* = *AHI x*])
show *extr-from-hd* (*x* # *y* # *xs*) = *AHI x* # *extr-from-hd* (*y* # *xs*)
using *hf-valid-y 4*(*1*)
by(*auto simp del*: *upd-uinfo-def elim*!: *extr-hf-valid*[*rotated*])
next
have *prefix* (*map AHI* (*hfs-valid-prefix ainfo* (*upd-uinfo info y*) (*y* # *xs*) *xo*)) (*extr* (*HVF y*))
using *4*(*2*) **by** (*auto simp del*: *upd-uinfo-def*)
then show *prefix* (*AHIS* (*hfs-valid-prefix ainfo info* (*x* # *y* # *xs*) *xo*)) (*AHI x* # *extr* (*HVF y*))
by (*auto simp add*: *TWu.takeW-split-tail*[**where** *?x* = *x*] *TWu.takeW.simps*(*1*) *simp del*: *upd-uinfo-def*)
qed(*auto*)
next
case *False*
then show *?thesis*
by(*auto simp add*: *TWu.takeW-split-tail hf-valid-invert*)
(*auto simp add*: *fcard-fempty intro*!: *ahi-eq*)
qed
qed(*auto simp add*: *TWu.takeW-split-tail TWu.takeW.simps*(*1*))

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

lemma *COND-upd-uinfo-ik*:
⟦*terms-uinfo uinfo* ⊆ *synth* (*analz ik*); *terms-hf hf* ⊆ *synth* (*analz ik*)⟧

⟹ *terms-uinfo* (*upd-uinfo uinfo hf*) ⊆ *synth* (*analz ik*)
by *fastforce*

lemma *COND-upd-uinfo-no-oracle*:
no-oracle ainfo uinfo ⟹ *no-oracle ainfo* (*upd-uinfo uinfo fld*)
by (*auto*)

3.9.6 Instantiation of *dataplane-3-directed* locale

print-locale *dataplane-3-directed*
sublocale
dataplane-3-directed - - - auth-seg0 terms-uinfo terms-hf hf-valid auth-restrict extr extr-ainfo term-ainfo

upd-uinfo ik-add
ik-oracle no-oracle
apply *unfold-locales*
using *COND-terms-hf COND-honest-hf-analz COND-extr-prefix-path*
COND-path-prefix-extr COND-hf-valid-uinfo COND-upd-uinfo-ik COND-upd-uinfo-no-oracle
COND-auth-restrict-upd **by** *auto*

3.9.7 Normalization of terms

We now show that all terms that occur in reachable states are normalized, meaning that they do not have directly nested FSets. For instance, a term *FS* {|*FS* {|*Num 0*|}, *Num 0*|} is not normalized, whereas *FS* {|*Hash* (*FS* {|*Num 0*|}), *Num 0*|} is normalized.

lemma *normalized-upd*:
⟦*normalized* (*FS* (*upd-uinfo info y*)); *normalized* (*FS* {| *HVF y* |})⟧
⟹ *normalized* (*FS info*)
by(*auto simp add*: *xor-singleton*)

declare *normalized.Lst*[*intro*!] *normalized.FSt*[*intro*!] *normalized.Hash*[*intro*!] *normalized.MPair*[*intro*!]

lemma *auth-uinfo-normalized*:
⟦*hfs-valid ainfo uinfo hfs nxt*; *auth-restrict ainfo uinfo hfs*⟧ ⟹ *normalized* (*FS uinfo*)
proof(*induction hfs nxt arbitrary*: *rule*: *TWu.holds.induct*[**where** *?upd=upd-uinfo*])
case (*1 info x y ys nxt*)
from *1* **have** *hfs-valid*: *hf-valid ainfo info x* (*Some y*)
hfs-valid ainfo (*upd-uinfo info y*) (*y # ys*) *nxt*
auth-restrict ainfo (*upd-uinfo info y*) (*y # ys*)
by(*auto simp only*: *TWu.holds.simps intro*!: *auth-restrict-upd*)
then have *normalized-upd-y*: *normalized* (*FS* (*upd-uinfo info y*)) **by** (*elim 1*(*1*))
obtain *z* **where** *hfy-valid*: *hf-valid ainfo* (*upd-uinfo info y*) *y z*
using *hfs-valid*(*2*) **by**(*auto dest*: *hfs-valid-first*)
obtain *info-s* **where** *info-s-def*[*simp*]: *xor info* {|*HVF y*|} = *info-s* **by** *simp*
from *normalized-upd-y* **show** *?case*
apply(*auto elim*!: *normalized-upd simp only*:)
using *hfy-valid info-s-def normalized-upd-y*
by (*auto 3 4 simp add*: *hf-valid-invert elim*: *ASIF.elims*(*2*))
qed(*auto simp only*: *TWu.holds.simps auth-restrict-def*,
auto simp add: *hf-valid-invert*)

lemma *auth-normalized-hf*:

assumes *auth-restrict ainfo uinfo* (*pre* @ *hf* # *post*)
hfs-valid ainfo (*upds-uinfo-shifted uinfo pre hf*) (*hf* # *post*) *nxt*
upds-uinfo-shifted uinfo pre hf = *hf-uinfo*
shows *normalized* (*HVF hf*)
using *assms*(*1*−*2*)
apply(*auto dest*!: *hfs-valid-first simp add*: *hf-valid-invert assms*(*3*))
using *assms*(*2*−*3*)
by(*auto dest*!: *auth-uinfo-normalized dest*: *auth-restrict-app elim*: *ASIF.elims*(*2*))

lemma *auth-normalized*:
⟦*hf* ∈ *set hfs*; *hfs-valid ainfo uinfo hfs nxt*; *auth-restrict ainfo uinfo hfs*⟧
⟹ *normalized* (*HVF hf*)
by(*auto dest*: *TWu.holds-intermediate-ex intro*: *auth-normalized-hf*)

All terms derivable by the intruder are normalized. Note that (i) the dynamic intruder knowledge *ik-dyn* contains all terms of messages contained in the state and (ii) the dynamic intruder knowledge remains constant. Hence this lemma suffices to show that all terms contained in *int* and *ext* channels of reachable states are normalized as well.

lemma *ik-synth-normalized*: *t* ∈ *synth* (*analz ik*) ⟹ *normalized t*
by (*auto*, *auto simp add*: *ik-simpler*)
(*auto simp add*: *ik-hfs-def auth-seg2-def hfs-valid-prefix-generic-def*
elim!: *auth-normalized*)

end
end

3.10 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
```

3.10.1 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 *term-ainfo* :: *msgterm* ⇒ *msgterm* **where**
term-ainfo ≡ *id*

An authenticated info field is always a number (corresponding to a timestamp). The unauthenticated info field is set to the empty term ε.

definition *auth-restrict* **where**
auth-restrict ainfo uinfo l ≡ (∃ *ts*. *ainfo* = *Num ts*) ∧ (*uinfo* = ε)

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 *terms-hf* :: *ICING-HF* ⇒ *msgterm set* **where**
terms-hf hf = {*HVF hf*}

abbreviation *terms-uinfo* :: *msgterm* ⇒ *msgterm set* **where**
terms-uinfo x ≡ {*x*}

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 ⟷
(∃ *PoC-i-expire ahi A-i* . *tsn* = *Num PoC-i-expire* ∧ *ahi* = *AHI hf* ∧
UHI hf = () ∧ *uinfo* = ε ∧
HVF hf = *A-i* ∧
A-i = *Hash* (*maccontents ahi hfs PoC-i-expire*))
apply(*cases hf*) **by**(*auto elim*!: *hf-valid.elims*)

lemma *hf-valid-auth-restrict*[*dest*]: *hf-valid ainfo uinfo hfs hf* ⟹ *auth-restrict ainfo uinfo l*
by(*auto simp add*: *hf-valid-invert auth-restrict-def*)

lemma *auth-restrict-ainfo*[*dest*]: *auth-restrict ainfo uinfo l* ⟹ ∃ *ts*. *ainfo* = *Num ts*
by(*auto simp add*: *auth-restrict-def*)
lemma *auth-restrict-uinfo*[*dest*]: *auth-restrict ainfo uinfo l* ⟹ *uinfo* = ε
by(*auto simp add*: *auth-restrict-def*)

lemma *info-hvf*:
assumes *hf-valid ainfo uinfo hfs m hf-valid ainfo′ uinfo′ hfs′ m′*
HVF m = *HVF m′ m* ∈ *set hfs m′* ∈ *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*)

3.10.2 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.

print-locale *dataplane-3-undirected-defs*
sublocale *dataplane-3-undirected-defs - - - auth-seg0 hf-valid auth-restrict extr extr-ainfo*
term-ainfo terms-hf terms-uinfo no-oracle
by *unfold-locales*

declare *parts-singleton*[*dest*]

definition *ik-add* :: *msgterm set* **where**
ik-add ≡ { *PoC* | *ainfo l uinfo hf PoC pkthash*.
(*ainfo*, *l*) ∈ *auth-seg2 uinfo*
∧ *hf* ∈ *set l* ∧ *HVF hf* = *Mac*[*PoC*] *pkthash* }

lemma *ik-addI*:
⟦(*ainfo*, *l*) ∈ *local.auth-seg2 uinfo*; *hf* ∈ *set l*; *HVF hf* = *Mac*[*PoC*] *pkthash*⟧ ⟹ *PoC* ∈ *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 auth-restrict-def*)
by (*meson sntag.elims*)

lemma *elem-eq*: ⟦$x \in xs$; $x = y$; $xs = ys$⟧ $\Longrightarrow y \in ys$
by *simp*

lemma *valid-hf-eq*:
⟦*HVF hf* = *Mac*[*Mac*[*sntag* (*AHI hf*)] ⟨*fullpath hfs*, *ainfo'*⟩] ⟨*Num 0*, *Hash* (*fullpath hfs*)⟩;
HVF hf' = *Mac*[*Mac*[*sntag* (*AHI hf*)] ⟨*fullpath hfs*, *ainfo'*⟩] *pkthash*;
(*ainfo'*, *l*) $\in$ *auth-seg2 uinfo*; *hf'* $\in$ *set l*⟧
$\Longrightarrow$ *hf* = *hf'*
by(*auto simp add*: *auth-seg2-def hf-valid-invert maccontents-def auth-restrict-def dest*!: *sntag-eq*)

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* $\equiv$ {}

lemma *uinfo-empty*[*dest*]: (*ainfo*, *hfs*) $\in$ *auth-seg2 uinfo* $\Longrightarrow$ *uinfo* = ε
by(*auto simp add*: *auth-seg2-def auth-restrict-def*)

3.10.3 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.

print-locale *dataplane-3-undirected-ik-defs*
sublocale
dataplane-3-undirected-ik-defs - - - auth-seg0 terms-uinfo no-oracle hf-valid auth-restrict extr extr-ainfo term-ainfo terms-hf ik-add ik-oracle
by *unfold-locales*

lemma *ik-hfs-form*: $t \in$ *parts ik-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-hfs-def*[*simp del*]

lemma *parts-ik-hfs*[*simp*]: *parts ik-hfs* = *ik-hfs*
by (*auto intro*!: *parts-Hash ik-hfs-form*)

This lemma allows us not only to expand the definition of *ik-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-hfs-simp*:
$t \in$ *ik-hfs* $\longleftrightarrow$ ($\exists\, t'$. t = *Hash* t') $\wedge$ ($\exists$ *hf* . t = *HVF hf*
$\wedge$ ($\exists$ *hfs*. *hf* $\in$ *set hfs* $\wedge$ ($\exists$ *ainfo uinfo*. (*ainfo*, *hfs*) $\in$ *auth-seg2 uinfo*
$\wedge$ *hf-valid ainfo uinfo hfs hf*))) (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *asm*: *?lhs*

then obtain *ainfo uinfo hf hfs* **where**
dfs: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo t* = *HVF hf*
by(*auto simp add*: *ik-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 3 4 simp add*: *auth-seg2-def intro*!: *ik-hfs-form intro*!: *exI*[*of* - *hf*])+
qed(*auto simp add*: *ik-hfs-def*)

lemma *ik-uinfo-empty*[*simp*]: *ik-uinfo* = {ε}
by(*auto simp add*: *ik-uinfo-def auth-seg2-def auth-restrict-def intro*!: *exI*[*of* - []])
declare *ik-uinfo-def*[*simp del*]

Properties of Intruder Knowledge

lemma *auth-ainfo*[*dest*]: ⟦(*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*⟧ ⟹ ∃ *ts* . *ainfo* = *Num ts*
by(*auto simp add*: *auth-seg2-def auth-restrict-def*)

lemma *Num-ik*[*intro*]: *Num ts* ∈ *ik*
by(*auto simp add*: *ik-def auth-seg2-def auth-restrict-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 auth-restrict-def ik-hfs-simp dest*: *ik-add-form*)

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

lemma *sntag-synth-bad*: *sntag ahi* ∈ *synth ik* ⟹ *ASID ahi* ∈ *bad*
by(*cases ahi*)
(*auto simp add*: *ik-def ik-hfs-simp auth-restrict-def auth-seg2-def dest*: *ik-add-form*)

lemma *HF-eq*:
⟦*AHI hf*′ = *AHI hf*; *UHI hf*′ = *UHI hf*; *HVF hf*′ = *HVF hf*⟧ ⟹ *hf*′ = (*hf*::(′*x*, ′*y*)*HF*)
apply(*cases hf*′, *cases hf*)
by(*auto elim*: *HF.cases*)

3.10.4 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 terms-hf hf* ⊆ *synth* (*analz ik*)
no-oracle ainfo uinfo hf ∈ *set hfs*
shows *terms-hf hf* ⊆ *analz ik*
proof−
from *assms*(*3*) **have** *hf-synth-ik*: *HVF hf* ∈ *synth ik* **by** *auto*
then have ∃ *hfs uinfo*. *hf* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*
using *assms*(*1*,*2*,*4*,*5*)
apply(*auto simp add*: *ik-def hf-valid-invert ik-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*)

subgoal by(*auto simp add*: *ik-hfs-simp ik-def hf-valid-invert simp del*: *ik-uinfo-def*)
subgoal by(*auto simp add*: *ik-hfs-simp ik-def hf-valid-invert maccontents-def*
intro: *sntag-synth-bad dest*: *ik-add-form*)
subgoal
apply(*auto simp add*: *ik-hfs-simp ik-def hf-valid-invert maccontents-def auth-restrict-def auth-seg2-def*
intro: *sntag-synth-bad dest*: *ik-add-form simp del*: *ik-uinfo-def*)
subgoal by (*simp add*: *fullpath-def*)
subgoal using *fullpath-def ik-add-form* **by** *auto*
apply (*auto simp add*: *ik-add-def*)
subgoal for *ainfoa l uinfoa hf′ pkthash*
apply(*frule valid-hf-eq*[**where** *?hf′=hf′*])
by(*auto dest*: *valid-hf-eq simp add*: *hf-valid-invert maccontents-def auth-seg2-def auth-restrict-def*)
done
done
then have *HVF hf* ∈ *ik*
using *assms*(*2*)
by(*auto simp add*: *ik-hfs-def intro*!: *ik-ik-hfs intro*!: *exI*)
then show *?thesis* **by** *auto*
qed

lemma *COND-terms-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* ∧ (∃ *uinfo′* . (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo′*)
using *assms* **apply**(*auto 3 4 simp add*: *hf-valid-invert ik-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-hfs-simp dest*: *ik-add-form*)
by (*metis info-hvf*(*1*) *info-hvf*(*2*))

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

3.10.5 Instantiation of *dataplane-3-undirected* locale

print-locale *dataplane-3-undirected*
sublocale
dataplane-3-undirected - - - auth-seg0 hf-valid auth-restrict extr extr-ainfo term-ainfo terms-uinfo ik-add terms-hf
ik-oracle no-oracle
apply *unfold-locales*
using *COND-terms-hf COND-honest-hf-analz COND-extr COND-hf-valid-uinfo* **by** *auto*

end
end

3.11 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

3.11.1 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 *term-ainfo* :: *msgterm* $\Rightarrow$ *msgterm* **where**
term-ainfo $\equiv$ *id*

An authenticated info field is always a number (corresponding to a timestamp). The unauthenticated info field is set to the empty term ε.

definition *auth-restrict* **where**
auth-restrict ainfo uinfo l $\equiv$ ($\exists$ *ts*. *ainfo* = *Num ts*) $\wedge$ (*uinfo* = ε)

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 *terms-hf* :: *ICING-HF* $\Rightarrow$ *msgterm set* **where**
terms-hf hf = {*HVF hf*}

abbreviation *terms-uinfo* :: *msgterm* $\Rightarrow$ *msgterm set* **where**
terms-uinfo x $\equiv$ {*x*}

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-auth-restrict*[*dest*]: *hf-valid ainfo uinfo hfs hf* $\Longrightarrow$ *auth-restrict ainfo uinfo l*
by(*auto simp add*: *hf-valid-invert auth-restrict-def*)

lemma *auth-restrict-ainfo*[*dest*]: *auth-restrict ainfo uinfo l* $\Longrightarrow$ $\exists$ *ts*. *ainfo* = *Num ts*
by(*auto simp add*: *auth-restrict-def*)
lemma *auth-restrict-uinfo*[*dest*]: *auth-restrict ainfo uinfo l* $\Longrightarrow$ *uinfo* = ε
by(*auto simp add*: *auth-restrict-def*)

lemma *info-hvf*:
assumes *hf-valid ainfo uinfo hfs m hf-valid ainfo′ uinfo′ hfs′ m′*
HVF m = HVF m′ m ∈ set hfs m′ ∈ 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*)

3.11.2 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.

print-locale *dataplane-3-undirected-defs*
sublocale *dataplane-3-undirected-defs - - - auth-seg0 hf-valid auth-restrict extr extr-ainfo term-ainfo terms-hf terms-uinfo no-oracle*
by *unfold-locales*

declare *parts-singleton*[*dest*]

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

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

lemma *uinfo-empty*[*dest*]: (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo* ⟹ *uinfo* = *ε*
by(*auto simp add*: *auth-seg2-def auth-restrict-def*)

3.11.3 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.

print-locale *dataplane-3-undirected-ik-defs*
sublocale
dataplane-3-undirected-ik-defs - - - auth-seg0 terms-uinfo no-oracle hf-valid auth-restrict extr extr-ainfo term-ainfo terms-hf ik-add ik-oracle
by *unfold-locales*

lemma *ik-hfs-form*: *t* ∈ *parts ik-hfs* ⟹ ∃ *t′* . *t* = *Hash t′*
apply *auto* **apply**(*drule parts-singleton*)
by(*auto simp add*: *auth-seg2-def hf-valid-invert*)

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

lemma *parts-ik-hfs*[*simp*]: *parts ik-hfs* = *ik-hfs*
by (*auto intro*!: *parts-Hash ik-hfs-form*)

This lemma allows us not only to expand the definition of *ik-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-hfs-simp*:

t ∈ *ik-hfs* ⟷ (∃ *t'* . *t* = *Hash t'*) ∧ (∃ *hf* . *t* = *HVF hf*
∧ (∃ *hfs*. *hf* ∈ *set hfs* ∧ (∃ *ainfo uinfo*. (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*
∧ *hf-valid ainfo uinfo hfs hf*))) (**is** *?lhs* ⟷ *?rhs*)
proof
assume *asm*: *?lhs*
then obtain *ainfo uinfo hf hfs* **where**
dfs: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo t* = *HVF hf*
by(*auto simp add*: *ik-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 3 4 simp add*: *auth-seg2-def intro*!: *ik-hfs-form exI*[*of* - *hf*])
qed(*auto simp add*: *ik-hfs-def*)

lemma *ik-uinfo-empty*[*simp*]: *ik-uinfo* = {*ε*}
by(*auto simp add*: *ik-uinfo-def auth-seg2-def auth-restrict-def intro*!: *exI*[*of* - []])
declare *ik-uinfo-def*[*simp del*]

Properties of Intruder Knowledge

lemma *auth-ainfo*[*dest*]: ⟦(*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*⟧ ⟹ ∃ *ts* . *ainfo* = *Num ts*
by(*auto simp add*: *auth-seg2-def*)

lemma *Num-ik*[*intro*]: *Num ts* ∈ *ik*
by(*auto simp add*: *ik-def auth-seg2-def auth-restrict-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*
by(*rule no-crypt-analz-is-parts*)
(*auto simp add*: *ik-def auth-seg2-def ik-hfs-simp auth-restrict-def*)

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

3.11.4 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 terms-hf hf* ⊆ *synth* (*analz ik*)
no-oracle ainfo uinfo hf ∈ *set hfs*
shows *terms-hf hf* ⊆ *analz ik*
proof−
from *assms*(*3*) **have** *hf-synth-ik*: *HVF hf* ∈ *synth ik* **by** *auto*
then have ∃ *hfs uinfo*. *hf* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*
using *assms*(*1*,*2*,*4*,*5*)
apply(*auto simp add*: *ik-def hf-valid-invert ik-hfs-simp*)
subgoal for *ts' hf' hfs'*
using *HF.equality* **by** (*fastforce dest*!: *sntag-eq intro*: *exI*[*of* - *hfs'*])
by(*auto simp add*: *ik-hfs-simp ik-def hf-valid-invert sntag-synth-bad*)
then have *HVF hf* ∈ *ik*

using *assms*(*2*)
by(*auto simp add*: *ik-hfs-def intro*!: *ik-ik-hfs intro*!: *exI*)
then show *?thesis* **by** *auto*
qed

lemma *COND-terms-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* $\wedge$ ($\exists$ *uinfo'* . (*ainfo*, *hfs*) $\in$ *auth-seg2 uinfo'*)
using *assms* **apply**(*auto 3 4 simp add*: *hf-valid-invert ik-hfs-simp ik-def dest*: *ahi-eq*)
apply(*frule sntag-eq*)
apply(*auto simp add*: *ik-def ik-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*)

3.11.5 Instantiation of *dataplane-3-undirected* locale

print-locale *dataplane-3-undirected*
sublocale
dataplane-3-undirected - - - auth-seg0 hf-valid auth-restrict extr extr-ainfo term-ainfo terms-uinfo ik-add terms-hf
ik-oracle no-oracle
apply *unfold-locales*
using *COND-terms-hf COND-honest-hf-analz COND-extr COND-hf-valid-uinfo* **by** *auto*

end
end

3.12 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

3.12.1 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 *term-ainfo* :: *msgterm* ⇒ *msgterm* **where**
term-ainfo ≡ *id*

An authenticated info field is always a number (corresponding to a timestamp). The unauthenticated info field is set to the empty term ε.

definition *auth-restrict* **where**
$auth\text{-}restrict\ ainfo\ uinfo\ l \equiv (\exists\, ts.\ ainfo = Num\ ts) \wedge (uinfo = \varepsilon)$

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 *terms-hf* :: *ICING-HF* ⇒ *msgterm set* **where**
terms-hf hf = {*HVF hf*}

abbreviation *terms-uinfo* :: *msgterm* ⇒ *msgterm set* **where**
terms-uinfo $x \equiv \{x\}$

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 = () ∧
$HVF\ hf = Mac[sntag\ ahi]\ (L\ ((Num\ ts)\#(map\ (hf2term\ o\ AHI)\ hfs))) \wedge uinfo = \varepsilon)$
apply(*cases hf*) **by**(*auto elim*!: *hf-valid.elims*)

lemma *hf-valid-auth-restrict*[*dest*]: *hf-valid ainfo uinfo hfs hf* ⟹ *auth-restrict ainfo uinfo l*
by(*auto simp add*: *hf-valid-invert auth-restrict-def*)

lemma *auth-restrict-ainfo*[*dest*]: *auth-restrict ainfo uinfo l* ⟹ ∃ *ts*. *ainfo* = *Num ts*
by(*auto simp add*: *auth-restrict-def*)
lemma *auth-restrict-uinfo*[*dest*]: *auth-restrict ainfo uinfo l* ⟹ *uinfo* = ε
by(*auto simp add*: *auth-restrict-def*)

3.12.2 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.

print-locale *dataplane-3-undirected-defs*
sublocale *dataplane-3-undirected-defs - - - auth-seg0 hf-valid auth-restrict extr extr-ainfo term-ainfo terms-hf terms-uinfo no-oracle*
by *unfold-locales*

declare *parts-singleton*[*dest*]

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

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

lemma *uinfo-empty*[*dest*]: (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo* ⟹ *uinfo* = *ε*
by(*auto simp add*: *auth-seg2-def auth-restrict-def*)

3.12.3 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.

print-locale *dataplane-3-undirected-ik-defs*
sublocale
dataplane-3-undirected-ik-defs - - - auth-seg0 terms-uinfo no-oracle hf-valid auth-restrict extr extr-ainfo term-ainfo terms-hf ik-add ik-oracle
by *unfold-locales*

lemma *ik-hfs-form*: *t* ∈ *parts ik-hfs* ⟹ ∃ *t'* . *t* = *Hash t'*
apply *auto* **apply**(*drule parts-singleton*)
by(*auto simp add*: *auth-seg2-def hf-valid-invert*)

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

lemma *parts-ik-hfs*[*simp*]: *parts ik-hfs* = *ik-hfs*
by (*auto intro*!: *parts-Hash ik-hfs-form*)

This lemma allows us not only to expand the definition of *ik-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-hfs-simp*:
t ∈ *ik-hfs* ⟷ (∃ *t'* . *t* = *Hash t'*) ∧ (∃ *hf* . *t* = *HVF hf*
∧ (∃ *hfs uinfo*. *hf* ∈ *set hfs* ∧ (∃ *ainfo* . (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*
∧ *hf-valid ainfo uinfo hfs hf*))) (**is** *?lhs* ⟷ *?rhs*)
proof
assume *asm*: *?lhs*
then obtain *ainfo uinfo hf hfs* **where**
dfs: *hf* ∈ *set hfs* (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo t* = *HVF hf*
by(*auto simp add*: *ik-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 3 4 simp add*: *auth-seg2-def intro*!: *ik-hfs-form exI*[*of - hf*])
qed(*auto simp add*: *ik-hfs-def*)

lemma *ik-uinfo-empty*[*simp*]: *ik-uinfo* = {*ε*}

by(*auto simp add*: *ik-uinfo-def auth-seg2-def auth-restrict-def intro*!: *exI*[*of* - []])
declare *ik-uinfo-def*[*simp del*]

Properties of Intruder Knowledge

lemma *auth-ainfo*[*dest*]: ⟦(*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*⟧ ⟹ ∃ *ts* . *ainfo* = *Num ts*
by(*auto simp add*: *auth-seg2-def*)

lemma *Num-ik*[*intro*]: *Num ts* ∈ *ik*
by(*auto simp add*: *ik-def auth-seg2-def auth-restrict-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*
by(*rule no-crypt-analz-is-parts*)
(*auto simp add*: *ik-def auth-seg2-def ik-hfs-simp auth-restrict-def*)

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

3.12.4 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 terms-hf hf* ⊆ *synth* (*analz ik*)
no-oracle ainfo uinfo hf ∈ *set hfs*
shows *terms-hf hf* ⊆ *analz ik*
proof−
from *assms*(*3*) **have** *hf-synth-ik*: *HVF hf* ∈ *synth ik* **by** *auto*
then have ∃ *hfs uinfo*. *hf* ∈ *set hfs* ∧ (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo*
using *assms*(*1*,*2*,*4*,*5*)
apply(*auto simp add*: *ik-def hf-valid-invert ik-hfs-simp*)
subgoal for *ts' hf' hfs'*
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-hfs-simp ik-def hf-valid-invert sntag-synth-bad*)
then have *HVF hf* ∈ *ik*

using *assms*(*2*)
by(*auto simp add*: *ik-hfs-def intro*!: *ik-ik-hfs intro*!: *exI*)
then show *?thesis* **by** *auto*
qed

lemma *COND-terms-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* ∧ (∃ *uinfo′* . (*ainfo*, *hfs*) ∈ *auth-seg2 uinfo′*)
using *assms* **apply**(*auto 3 4 simp add*: *hf-valid-invert ik-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*⟧ ⟹ *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*⟧
⟹ *uinfo′* = *uinfo*
by(*auto simp add*: *hf-valid-invert*)

3.12.5 Instantiation of *dataplane-3-undirected* locale

print-locale *dataplane-3-undirected*
sublocale
dataplane-3-undirected - - - auth-seg0 hf-valid auth-restrict extr extr-ainfo term-ainfo terms-uinfo ik-add terms-hf
ik-oracle no-oracle
apply *unfold-locales*
using *COND-terms-hf COND-honest-hf-analz COND-extr COND-hf-valid-uinfo* **by** *auto*

end
end

3.13 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
    instances/Anapaya-SCION
begin

end
```