

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

Tobias Klenze, Christoph Sprenger

March 17, 2025

Contents

1	Verification Infrastructure	4
1.1	Event Systems	5
1.1.1	Reachable states and invariants	5
1.1.2	Traces	5
1.1.3	Simulation	8
1.1.4	Simulation up to simulation preorder	10
1.2	Atomic messages	11
1.2.1	Agents	11
1.2.2	Nonces and keys	11
1.3	Symmetric and Asymmetric Keys	12
1.3.1	Asymmetric Keys	12
1.3.2	Basic properties of <i>pubK</i> and <i>priK</i>	12
1.3.3	"Image" equations that hold for injective functions	13
1.3.4	Symmetric Keys	13
1.4	Theory of ASes and Messages for Security Protocols	15
1.4.1	keysFor operator	16
1.4.2	Inductive relation "parts"	17
1.4.3	Inductive relation "analz"	21
1.4.4	Inductive relation "synth"	25
1.4.5	HPair: a combination of Hash and MPair	29
1.5	Tools	33
1.5.1	Prefixes, suffixes, and fragments	33
1.5.2	Fragments	33
1.5.3	Pair Fragments	34
1.5.4	Head and Tails	35
1.6	takeW, holds and extract: Applying context-sensitive checks on list elements	36
1.6.1	Definitions	36
1.6.2	Lemmas	37
1.7	Extending <i>Take-While</i> with an additional, mutable parameter	41
1.7.1	Definitions	41
1.7.2	Lemmas	42
2	Abstract, and Concrete Parametrized Models	45
2.1	Network model	46
2.1.1	Interface check	46
2.2	Abstract Model	48

2.2.1	Events	49
2.2.2	Transition system	51
2.2.3	Path authorization property	51
2.2.4	Detectability property	53
2.3	Intermediate Model	54
2.3.1	Events	54
2.3.2	Transition system	55
2.3.3	Auxilliary definitions	56
2.4	Concrete Parametrized Model	58
2.4.1	Hop validation check, authorized segments, and path extraction. . . .	59
2.4.2	Intruder Knowledge definition	62
2.4.3	Events	63
2.4.4	Transition system	65
2.4.5	Assumptions of the parametrized model	65
2.4.6	Mapping dp2 state to dp1 state	66
2.4.7	Invariant: Derivable Intruder Knowledge is constant under <i>dp2-trans</i> .	67
2.4.8	Refinement proof	68
2.4.9	Property preservation	68
2.5	Network Assumptions used for authorized segments.	70
2.6	Parametrized dataplane protocol for directed protocols	71
2.6.1	Hop validation check, authorized segments, and path extraction. . . .	71
2.6.2	Conditions of the parametrized model	73
2.6.3	Lemmas that are needed for the refinement proof	75
2.7	Parametrized dataplane protocol for undirected protocols	79
2.7.1	Hop validation check, authorized segments, and path extraction. . . .	79
2.7.2	Conditions of the parametrized model	81
3	Instances	83
3.1	SCION	84
3.1.1	Hop validation check and extract functions	84
3.1.2	Definitions and properties of the added intruder knowledge	86
3.1.3	Properties of the intruder knowledge, including <i>ik-add</i> and <i>ik-oracle</i> .	86
3.1.4	Direct proof goals for interpretation of <i>dataplane-3-directed</i>	87
3.1.5	Instantiation of <i>dataplane-3-directed</i> locale	88
3.2	SCION Variant	89
3.3	SCION	90
3.3.1	Hop validation check and extract functions	90
3.3.2	Definitions and properties of the added intruder knowledge	92
3.3.3	Properties of the intruder knowledge, including <i>ik-add</i> and <i>ik-oracle</i> .	92
3.3.4	Direct proof goals for interpretation of <i>dataplane-3-directed</i>	93
3.3.5	Instantiation of <i>dataplane-3-directed</i> locale	94
3.4	EPIC Level 1 in the Basic Attacker Model	95
3.4.1	Hop validation check and extract functions	95
3.4.2	Definitions and properties of the added intruder knowledge	97
3.4.3	Properties of the intruder knowledge, including <i>ik-add</i> and <i>ik-oracle</i> .	98
3.4.4	Direct proof goals for interpretation of <i>dataplane-3-directed</i>	100
3.4.5	Instantiation of <i>dataplane-3-directed</i> locale	100

3.5	EPIC Level 1 in the Strong Attacker Model	102
3.5.1	Hop validation check and extract functions	102
3.5.2	Definitions and properties of the added intruder knowledge	104
3.5.3	Properties of the intruder knowledge, including <i>ik-add</i> and <i>ik-oracle</i>	105
3.5.4	Direct proof goals for interpretation of <i>dataplane-3-directed</i>	107
3.5.5	Instantiation of <i>dataplane-3-directed</i> locale	108
3.6	EPIC Level 1 Example instantiation of locale	109
3.6.1	Left segment	109
3.6.2	Right segment	109
3.6.3	Executability	111
3.7	EPIC Level 2 in the Strong Attacker Model	115
3.7.1	Hop validation check and extract functions	115
3.7.2	Definitions and properties of the added intruder knowledge	117
3.7.3	Properties of the intruder knowledge, including <i>ik-add</i> and <i>ik-oracle</i>	118
3.7.4	Direct proof goals for interpretation of <i>dataplane-3-directed</i>	120
3.7.5	Instantiation of <i>dataplane-3-directed</i> locale	121
3.8	Abstract XOR	122
3.8.1	Abstract XOR definition and lemmas	122
3.8.2	Lemmas referring to XOR and msgterm	123
3.9	Anapaya-SCION	124
3.9.1	Hop validation check and extract functions	124
3.9.2	Definitions and properties of the added intruder knowledge	126
3.9.3	Properties of the intruder knowledge, including <i>fset</i>	126
3.9.4	Lemmas helping with conditions relating to extract	128
3.9.5	Direct proof goals for interpretation of <i>dataplane-3-directed</i>	129
3.9.6	Instantiation of <i>dataplane-3-directed</i> locale	130
3.9.7	Normalization of terms	130
3.10	ICING	131
3.10.1	Hop validation check and extract functions	131
3.10.2	Definitions and properties of the added intruder knowledge	133
3.10.3	Properties of the intruder knowledge, including <i>ik-add</i> and <i>ik-oracle</i>	134
3.10.4	Direct proof goals for interpretation of <i>dataplane-3-undirected</i>	135
3.10.5	Instantiation of <i>dataplane-3-undirected</i> locale	135
3.11	ICING variant	137
3.11.1	Hop validation check and extract functions	137
3.11.2	Definitions and properties of the added intruder knowledge	139
3.11.3	Properties of the intruder knowledge, including <i>ik-add</i> and <i>ik-oracle</i>	139
3.11.4	Direct proof goals for interpretation of <i>dataplane-3-undirected</i>	140
3.11.5	Instantiation of <i>dataplane-3-undirected</i> locale	140
3.12	ICING variant	142
3.12.1	Hop validation check and extract functions	142
3.12.2	Definitions and properties of the added intruder knowledge	143
3.12.3	Properties of the intruder knowledge, including <i>ik-add</i> and <i>ik-oracle</i>	144
3.12.4	Direct proof goals for interpretation of <i>dataplane-3-undirected</i>	145
3.12.5	Instantiation of <i>dataplane-3-undirected</i> locale	145
3.13	All Protocols	146

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.

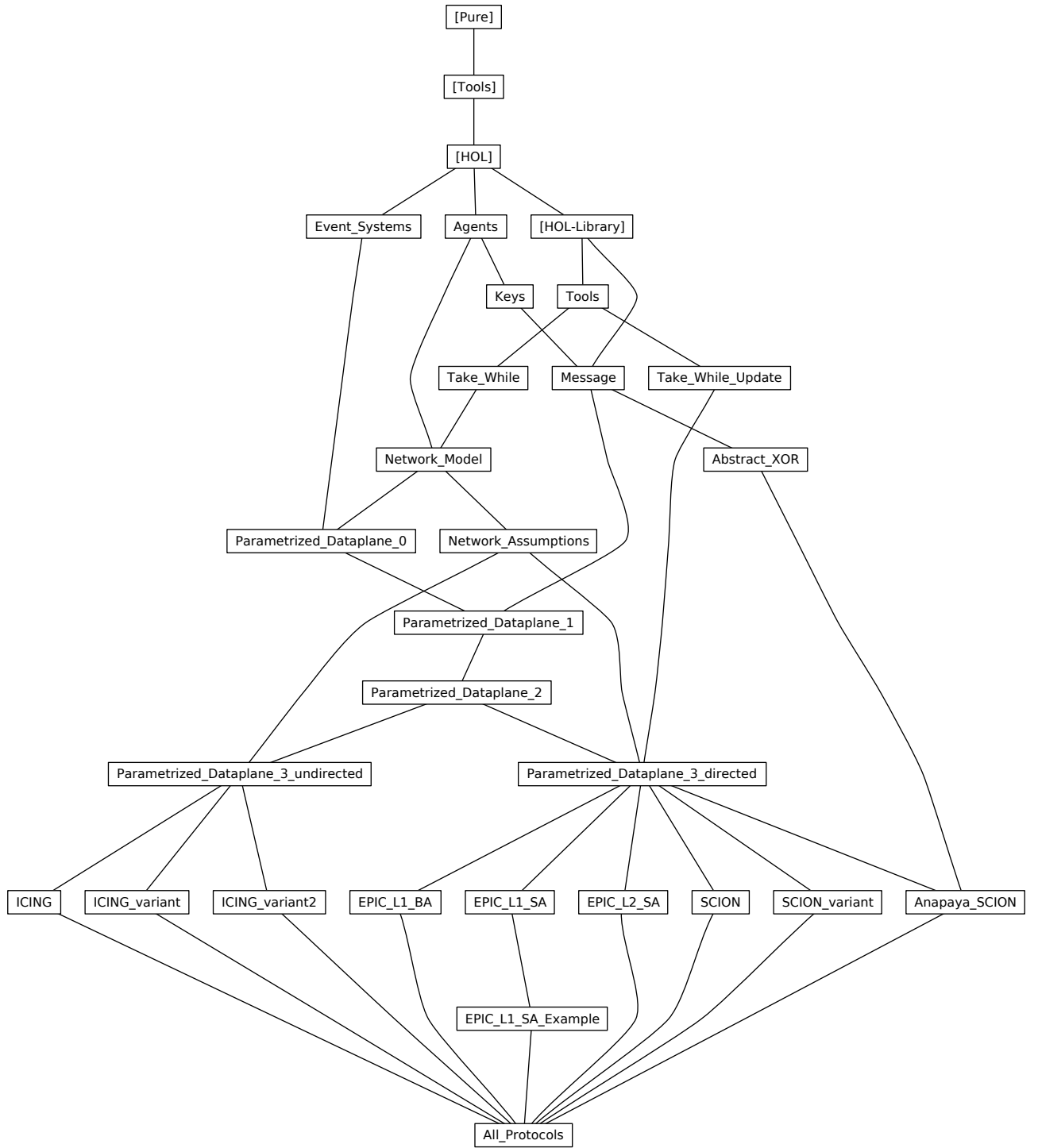


Figure 1: Theory dependencies

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  $\Rightarrow$  bool
  trans :: 's  $\Rightarrow$  'e  $\Rightarrow$  's  $\Rightarrow$  bool  ( $\langle (4\text{-} \text{---} \rightarrow -) \rangle$  [50, 50, 50] 90)
```

1.1.1 Reachable states and invariants

```
inductive
  reach :: ('e, 's) ES  $\Rightarrow$  's  $\Rightarrow$  bool for E
  where
    reach-init [simp, intro]: init E s  $\Longrightarrow$  reach E s
    | reach-trans [intro]:  $\llbracket E: s \text{--}e\rightarrow s'; \text{reach } E s \rrbracket \Longrightarrow \text{reach } E s'$ 
```

```
thm reach.induct
```

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

```
definition Inv :: ('e, 's) ES  $\Rightarrow$  ('s  $\Rightarrow$  bool)  $\Rightarrow$  bool where
  Inv E I  $\longleftrightarrow (\forall s. \text{reach } E s \longrightarrow I s)$ 
```

```
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  $\bigwedge s0. \text{init } E s0 \Longrightarrow I s0$ 
  and  $\bigwedge s e s'. \llbracket E: s \text{--}e\rightarrow s'; \text{reach } E s; I s \rrbracket \Longrightarrow I s'$ 
  shows Inv E I
   $\langle \text{proof} \rangle$ 
```

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  $\bigwedge s0. \text{init } E s0 \Longrightarrow I s0$ 
  and  $\bigwedge s e s'. \llbracket E: s \text{--}e\rightarrow s'; \text{reach } E s; I s; J s; J s' \rrbracket \Longrightarrow I s'$ 
  shows Inv E I
   $\langle \text{proof} \rangle$ 
```

1.1.2 Traces

```
type-synonym 'e trace = 'e list
```

```
inductive
  trace :: ('e, 's) ES  $\Rightarrow$  's  $\Rightarrow$  'e trace  $\Rightarrow$  's  $\Rightarrow$  bool  ( $\langle (4\text{-} - \text{--} \langle - \rangle \rightarrow -) \rangle$  [50, 50, 50] 90)
  for E s
  where
```

$\text{trace-nil } [\text{simp}, \text{intro!}]:$
 $E: s - \langle [] \rangle \rightarrow s$
 $| \text{trace-snoc } [\text{intro}]:$
 $\llbracket E: s - \langle \tau \rangle \rightarrow s'; E: s' - e \rightarrow s'' \rrbracket \implies E: s - \langle \tau @ [e] \rangle \rightarrow s''$

thm *trace.induct*

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

lemma *trace-init-independence* $[\text{elim}]:$
assumes $E: s - \langle \tau \rangle \rightarrow s'$ *trans* $E = \text{trans } F$
shows $F: s - \langle \tau \rangle \rightarrow s'$
 $\langle \text{proof} \rangle$

lemma *trace-single* $[\text{simp}, \text{intro!}]: \llbracket E: s - e \rightarrow s' \rrbracket \implies E: s - \langle [e] \rangle \rightarrow s'$
 $\langle \text{proof} \rangle$

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
 $E: s'' - \langle \tau \rangle \rightarrow s' \quad E: s - e \rightarrow s''$
shows
 $E: s - \langle e \# \tau \rangle \rightarrow s'$
 $\langle \text{proof} \rangle$

lemma *trace-cases-cons*:
assumes
 $E: s - \langle \tau \rangle \rightarrow s'$
 $\llbracket \tau = []; s' = s \rrbracket \implies P$
 $\bigwedge e \tau' s''. \llbracket \tau = e \# \tau'; E: s - e \rightarrow s''; E: s'' - \langle \tau' \rangle \rightarrow s' \rrbracket \implies P$
shows P
 $\langle \text{proof} \rangle$

lemma *trace-consD*: $(E: s - \langle e \# \tau \rangle \rightarrow s') \implies \exists s''. (E: s - e \rightarrow s'') \wedge (E: s'' - \langle \tau \rangle \rightarrow s')$
 $\langle \text{proof} \rangle$

We show how a trace can be appended to another.

lemma *trace-append*: $(E: s - \langle \tau_1 \rangle \rightarrow s') \wedge (E: s' - \langle \tau_2 \rangle \rightarrow s'') \implies E: s - \langle \tau_1 @ \tau_2 \rangle \rightarrow s''$
 $\langle \text{proof} \rangle$

lemma *trace-append-invert*: $(E: s - \langle \tau_1 @ \tau_2 \rangle \rightarrow s'') \implies \exists s'. (E: s - \langle \tau_1 \rangle \rightarrow s') \wedge (E: s' - \langle \tau_2 \rangle \rightarrow s'')$
 $\langle \text{proof} \rangle$

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

lemma *trace-induct2* $[\text{consumes } 3, \text{case-names Nil Snoc}]:$
 $\llbracket E: s - \langle \tau \rangle \rightarrow s''; F: t - \langle \sigma \rangle \rightarrow t''; \text{length } \tau = \text{length } \sigma;$
 $P \llbracket s \rrbracket t;$
 $\bigwedge \tau s' e s'' \sigma t' f t''.$
 $\llbracket E: s - \langle \tau \rangle \rightarrow s'; E: s' - e \rightarrow s''; F: t - \langle \sigma \rangle \rightarrow t'; F: t' - f \rightarrow t''; P \tau s' \sigma t' \rrbracket$

$\implies P (\tau @ [e]) s'' (\sigma @ [f]) t''$
 $\implies P \tau s'' \sigma t''$
 $\langle \text{proof} \rangle$

Relate traces to reachability and invariants

lemma *reach-trace-equiv*: $\text{reach } E s \longleftrightarrow (\exists s0 \tau. \text{init } E s0 \wedge E: s0 -\langle \tau \rangle \rightarrow s) \text{ (is } ?A \longleftrightarrow ?B)$
 $\langle \text{proof} \rangle$

lemma *reach-traceI*: $\llbracket \text{init } E s0; E: s0 -\langle \tau \rangle \rightarrow s \rrbracket \implies \text{reach } E s$
 $\langle \text{proof} \rangle$

lemma *reach-trace-extend*: $\llbracket E: s -\langle \tau \rangle \rightarrow s'; \text{reach } E s \rrbracket \implies \text{reach } E s'$
 $\langle \text{proof} \rangle$

lemma *Inv-trace*: $\llbracket \text{Inv } E I; \text{init } E s0; E: s0 -\langle \tau \rangle \rightarrow s' \rrbracket \implies I s'$
 $\langle \text{proof} \rangle$

Trace semantics of event systems

We define the set of traces of an event system.

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

lemma *tracesI* [intro]: $\llbracket \text{init } E s; E: s -\langle \tau \rangle \rightarrow s' \rrbracket \implies \tau \in \text{traces } E$
 $\langle \text{proof} \rangle$

lemma *tracesE* [elim]: $\llbracket \tau \in \text{traces } E; \bigwedge s s'. \llbracket \text{init } E s; E: s -\langle \tau \rangle \rightarrow s' \rrbracket \implies P \rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *traces-nil* [simp, intro!]: $\text{init } E s \implies [] \in \text{traces } E$
 $\langle \text{proof} \rangle$

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* :: $(e, s) ES \Rightarrow e \text{ trace set} \Rightarrow \text{bool}$ (**infix** $\langle \models_{ES} \rangle$ 90) **where**
 $E \models_{ES} \varphi \longleftrightarrow \text{traces } E \subseteq \varphi$

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

$\tau \in \text{traces } E$

$\bigwedge s s'. \llbracket \text{init } E s; E: s -\langle \tau \rangle \rightarrow s' \rrbracket \implies I \tau s'$

$\bigwedge s. I \tau s \implies \tau \in \varphi$

shows $\tau \in \varphi$ $\langle \text{proof} \rangle$

lemma *trace-property-rule*:

assumes

$\bigwedge s0. \text{init } E \ s0 \implies I \ [] \ s0$
 $\bigwedge s \ s' \ \tau \ e \ s''.$
 $\llbracket \text{init } E \ s; E: s - \langle \tau \rangle \rightarrow s'; E: s' - e \rightarrow s''; I \ \tau \ s'; \text{reach } E \ s' \rrbracket \implies I \ (\tau @ [e]) \ s''$
 $\bigwedge \tau \ s. \llbracket I \ \tau \ s; \text{reach } E \ s \rrbracket \implies \tau \in \varphi$
shows $E \models_{ES} \varphi$
 $\langle \text{proof} \rangle$

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

lemma *Inv-trace-property*:

assumes $\text{Inv } E \ I$ **and** $[] \in \varphi$
and $(\bigwedge s \ \tau \ s' \ e \ s'').$
 $\llbracket \text{init } E \ s; E: s - \langle \tau \rangle \rightarrow s'; E: s' - e \rightarrow s''; I \ s; I \ s'; \text{reach } E \ s'; \tau \in \varphi \rrbracket \implies \tau @ [e] \in \varphi$
shows $E \models_{ES} \varphi$
 $\langle \text{proof} \rangle$

1.1.3 Simulation

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

coinductive

$\text{sim} :: ('e, 's) \text{ ES} \Rightarrow ('f, 't) \text{ ES} \Rightarrow ('e \Rightarrow 'f) \Rightarrow 's \Rightarrow 't \Rightarrow \text{bool}$
for $E \ F \ \pi$
where
 $\llbracket \bigwedge e \ s'. (E: s - e \rightarrow s') \implies \exists t'. (F: t - \pi \ e \rightarrow t') \wedge \text{sim } E \ F \ \pi \ s' \ t' \rrbracket \implies \text{sim } E \ F \ \pi \ s \ t$

abbreviation

$\text{simS} :: ('e, 's) \text{ ES} \Rightarrow ('f, 't) \text{ ES} \Rightarrow 's \Rightarrow ('e \Rightarrow 'f) \Rightarrow 't \Rightarrow \text{bool}$
 $(\langle (5, -, - \sqsubseteq -) \rangle [50, 50, 50, 60, 50] \ 90)$

where

$\text{simS } E \ F \ s \ \pi \ t \equiv \text{sim } E \ F \ \pi \ s \ t$

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

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
 $R \ s \ t$
 $\bigwedge s \ t \ a \ s'. \llbracket R \ s \ t; E: s - a \rightarrow s' \rrbracket \implies (\exists t'. (F: t - \pi \ a \rightarrow t') \wedge R \ s' \ t')$
shows
 $E, F: s \sqsubseteq_{\pi} t$
 $\langle \text{proof} \rangle$

lemma *sim-refl*: $E, E: s \sqsubseteq_i d \ s$

$\langle \text{proof} \rangle$

lemma *sim-trans*: $\llbracket E, F: s \sqsubseteq_{\pi} 1 \ t; F, G: t \sqsubseteq_{\pi} 2 \ u \rrbracket \implies E, G: s \sqsubseteq_{(\pi 2 \circ \pi 1)} u$
 $\langle \text{proof} \rangle$

Extend transition simulation to traces.

lemma *trace-sim*:
assumes $E: s \rightarrow \langle \tau \rangle s' \ E, F: s \sqsubseteq_{\pi} t$
shows $\exists t'. (F: t \rightarrow \langle \text{map } \pi \ \tau \rangle t') \wedge (E, F: s' \sqsubseteq_{\pi} t')$
 $\langle \text{proof} \rangle$

Simulation for event systems

definition

sim-ES :: $(e, s) \text{ ES} \Rightarrow (e \Rightarrow f) \Rightarrow (f, t) \text{ ES} \Rightarrow \text{bool}$ $\langle (\exists \sqsubseteq -) \rangle [50, 60, 50] \ 95$

where

$E \sqsubseteq_{\pi} F \iff (\exists R.$
 $(\forall s0. \text{init } E \ s0 \longrightarrow (\exists t0. \text{init } F \ t0 \wedge R \ s0 \ t0)) \wedge$
 $(\forall s \ t. R \ s \ t \longrightarrow E, F: s \sqsubseteq_{\pi} t))$

lemma *sim-ES-I*:

assumes
 $\bigwedge s0. \text{init } E \ s0 \implies (\exists t0. \text{init } F \ t0 \wedge R \ s0 \ t0)$ **and**
 $\bigwedge s \ t. R \ s \ t \implies E, F: s \sqsubseteq_{\pi} t$
shows $E \sqsubseteq_{\pi} F$
 $\langle \text{proof} \rangle$

lemma *sim-ES-E*:

assumes
 $E \sqsubseteq_{\pi} F$
 $\bigwedge R. \llbracket \bigwedge s0. \text{init } E \ s0 \implies (\exists t0. \text{init } F \ t0 \wedge R \ s0 \ t0); \bigwedge s \ t. R \ s \ t \implies E, F: s \sqsubseteq_{\pi} t \rrbracket \implies P$
shows P
 $\langle \text{proof} \rangle$

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

lemma *simulate-ES*:

assumes
init: $\bigwedge s0. \text{init } E \ s0 \implies (\exists t0. \text{init } F \ t0 \wedge R \ s0 \ t0)$ **and**
step: $\bigwedge s \ t \ a \ s'. \llbracket R \ s \ t; \text{reach } E \ s; \text{reach } F \ t; E: s \xrightarrow{a} s' \rrbracket \implies (\exists t'. (F: t \xrightarrow{\pi \ a} t') \wedge R \ s' \ t')$
shows $E \sqsubseteq_{\pi} F$
 $\langle \text{proof} \rangle$

lemma *simulate-ES-with-invariants*:

assumes
init: $\bigwedge s0. \text{init } E \ s0 \implies (\exists t0. \text{init } F \ t0 \wedge R \ s0 \ t0)$ **and**
step: $\bigwedge s \ t \ a \ s'. \llbracket R \ s \ t; I \ s; J \ t; E: s \xrightarrow{a} s' \rrbracket \implies (\exists t'. (F: t \xrightarrow{\pi \ a} t') \wedge R \ s' \ t')$ **and**
invE: $\bigwedge s. \text{reach } E \ s \longrightarrow I \ s$ **and**
invF: $\bigwedge t. \text{reach } F \ t \longrightarrow J \ t$
shows $E \sqsubseteq_{\pi} F$ $\langle \text{proof} \rangle$

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

Variants with a functional simulation relation, aka refinement mapping.

lemma *simulate-ES-fun*:

assumes

init: $\bigwedge s0. \text{init } E \ s0 \implies \text{init } F \ (h \ s0)$ **and**

step: $\bigwedge s \ a \ s'. \llbracket E: s \dashv a \rightarrow s'; \text{reach } E \ s; \text{reach } F \ (h \ s) \rrbracket \implies F: h \ s \dashv \pi \ a \rightarrow h \ s'$

shows $E \sqsubseteq_{\pi} F$

$\langle \text{proof} \rangle$

lemma *simulate-ES-fun-with-invariants*:

assumes

init: $\bigwedge s0. \text{init } E \ s0 \implies \text{init } F \ (h \ s0)$ **and**

step: $\bigwedge s \ a \ s'. \llbracket E: s \dashv a \rightarrow s'; I \ s; J \ (h \ s) \rrbracket \implies F: h \ s \dashv \pi \ a \rightarrow h \ s'$ **and**

invE: $\bigwedge s. \text{reach } E \ s \longrightarrow I \ s$ **and**

invF: $\bigwedge t. \text{reach } F \ t \longrightarrow J \ t$

shows $E \sqsubseteq_{\pi} F$

$\langle \text{proof} \rangle$

lemmas *simulate-ES-fun-with-invariant* =

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

Reflexivity and transitivity for ES simulation.

lemma *sim-ES-reft*: $E \sqsubseteq_i d \ E$

$\langle \text{proof} \rangle$

lemma *sim-ES-trans*:

assumes $E \sqsubseteq_{\pi 1} F$ **and** $F \sqsubseteq_{\pi 2} G$ **shows** $E \sqsubseteq_{(\pi 2 \circ \pi 1)} G$

$\langle \text{proof} \rangle$

Soundness for trace inclusion and property preservation

lemma *simulation-soundness*: $E \sqsubseteq_{\pi} F \implies (\text{map } \pi) \text{'traces } E \subseteq \text{traces } F$

$\langle \text{proof} \rangle$

lemmas *simulation-rule* = *simulate-ES* [*THEN simulation-soundness*]

lemmas *simulation-rule-id* = *simulation-rule*[**where** $\pi = \text{id}$, *simplified*]

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

corollary *property-preservation*:

$\llbracket E \sqsubseteq_{\pi} F; F \models_{ES} P; \bigwedge \tau. \text{map } \pi \ \tau \in P \implies \tau \in Q \rrbracket \implies E \models_{ES} Q$

$\langle \text{proof} \rangle$

1.1.4 Simulation up to simulation preorder

lemma *sim-coinduct-upto-sim* [*consumes 1, case-names sim*]:

assumes

major: $R \ s \ t$ **and**

$S: \bigwedge s \ t \ a \ s'. \llbracket R \ s \ t; E: s \dashv a \rightarrow s' \rrbracket \implies$

$\exists t'. (F: t \dashv \pi \ a \rightarrow t') \wedge ((\text{sim } E \ E \ \text{id}) \text{ OO } R \text{ OO } (\text{sim } F \ F \ \text{id})) \ s' \ t'$

shows

$E, F: s \sqsubseteq_{\pi} t$

$\langle \text{proof} \rangle$

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 $\equiv$ $\neg$ *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.

typeddecl *fid-t*

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

fun *fid* :: *fresh-t* $\Rightarrow$ *fid-t* **where**

fid (*f* $\$$ *n*) = *f*

fun *num* :: *fresh-t* $\Rightarrow$ *nat* **where**

num (*f* $\$$ *n*) = *n*

Nonces

type-synonym

nonce = *fresh-t*

end

1.3 Symmetric and Asymmetric 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* $\Rightarrow$ *key* **where**
 invKey (*pubK* *A*) = *priK* *A*
 | *invKey* (*priK* *A*) = *pubK* *A*
 | *invKey* *K* = *K*

definition

symKeys :: *key* set **where**
symKeys $\equiv$ {*K*. *invKey* *K* = *K*}

lemma *invKey-K*: $K \in \text{symKeys} \implies \text{invKey } K = K$
 $\langle \text{proof} \rangle$

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

lemma *invKey-invKey-id* [*simp*]: $\text{invKey } (\text{invKey } K) = K$
 $\langle \text{proof} \rangle$

lemma *invKey-eq* [*simp*]: $(\text{invKey } K = \text{invKey } K') = (K = K')$
 $\langle \text{proof} \rangle$

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*: $\text{priK } A \neq \text{pubK } A'$
 $\langle \text{proof} \rangle$

lemma *publicKey-neq-privateKey*: $\text{pubK } A \neq \text{priK } A'$
 $\langle \text{proof} \rangle$

1.3.2 Basic properties of *pubK* and *priK*

lemma *publicKey-inject* [*iff*]: $(\text{pubK } A = \text{pubK } A') = (A = A')$
 $\langle \text{proof} \rangle$

lemma *not-symKeys-pubK* [*iff*]: $\text{pubK } A \notin \text{symKeys}$

$\langle \text{proof} \rangle$

lemma *not-symKeys-priK* [iff]: $\text{priK } A \notin \text{symKeys}$

$\langle \text{proof} \rangle$

lemma *symKey-neq-priK*: $K \in \text{symKeys} \implies K \neq \text{priK } A$

$\langle \text{proof} \rangle$

lemma *symKeys-neq-imp-neq*: $(K \in \text{symKeys}) \neq (K' \in \text{symKeys}) \implies K \neq K'$

$\langle \text{proof} \rangle$

lemma *symKeys-invKey-iff* [iff]: $(\text{invKey } K \in \text{symKeys}) = (K \in \text{symKeys})$

$\langle \text{proof} \rangle$

1.3.3 "Image" equations that hold for injective functions

lemma *invKey-image-eq* [simp]: $(\text{invKey } x \in \text{invKey } A) = (x \in A)$

$\langle \text{proof} \rangle$

lemma *invKey-pubK-image-priK-image* [simp]: $\text{invKey } ' \text{ pubK } ' AS = \text{priK } ' AS$

$\langle \text{proof} \rangle$

lemma *publicKey-notin-image-privateKey*: $\text{pubK } A \notin \text{priK } ' AS$

$\langle \text{proof} \rangle$

lemma *privateKey-notin-image-publicKey*: $\text{priK } x \notin \text{pubK } ' AA$

$\langle \text{proof} \rangle$

lemma *publicKey-image-eq* [simp]: $(\text{pubK } x \in \text{pubK } ' AA) = (x \in AA)$

$\langle \text{proof} \rangle$

lemma *privateKey-image-eq* [simp]: $(\text{priK } A \in \text{priK } ' AS) = (A \in AS)$

$\langle \text{proof} \rangle$

1.3.4 Symmetric Keys

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

lemma *sym-shrK*: $\text{macK } X \in \text{symKeys}$ — All shared keys are symmetric

$\langle \text{proof} \rangle$

Symmetric keys and inversion

lemma *symK-eq-invKey*: $\llbracket SK = \text{invKey } K; SK \in \text{symKeys} \rrbracket \implies K = SK$

$\langle \text{proof} \rangle$

Image-related lemmas.

lemma *publicKey-notin-image-shrK*: $\text{pubK } x \notin \text{macK } ' AA$

$\langle \text{proof} \rangle$

lemma *privateKey-notin-image-shrK*: $\text{priK } x \notin \text{macK } ' AA$

$\langle \text{proof} \rangle$

lemma *shrK-notin-image-publicKey*: $\text{mac}K\ x \notin \text{pub}K\ 'AA$
<proof>

lemma *shrK-notin-image-privateKey*: $\text{mac}K\ x \notin \text{pri}K\ 'AA$
<proof>

lemma *shrK-image-eq* [simp]: $(\text{mac}K\ x \in \text{mac}K\ 'AA) = (x \in AA)$
<proof>

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
 $-MTuple :: [a, args] \Rightarrow 'a * 'b \quad (\langle (\langle indent=2 notation=\langle mixfix message tuple \rangle \langle -, / \rangle) \rangle)$

syntax-consts

$-MTuple \equiv MPair$

translations

$\langle x, y, z \rangle \equiv \langle x, \langle y, z \rangle \rangle$
 $\langle x, y \rangle \equiv CONST MPair x y$

syntax

$-MHF :: [a, 'b, 'c, 'd, 'e] \Rightarrow 'a * 'b * 'c * 'd * 'e \quad (\langle (5HF \langle -, / -, / -, / - \rangle) \rangle)$

abbreviation

$Mac :: [msgterm, msgterm] \Rightarrow msgterm \quad (\langle (4Mac[-] /-) \rangle [0, 1000])$

where

— Message Y paired with a MAC computed with the help of X
 $Mac[X] Y \equiv Hash \langle X, Y \rangle$

abbreviation *macKey* **where** $macKey a \equiv Key (macK a)$

definition

$keysFor :: msgterm set \Rightarrow key set$

where

— Keys useful to decrypt elements of a message set
 $keysFor H \equiv invKey \{ K. \exists X. Crypt K X \in H \}$

Inductive Definition of "All Parts" of a Message

inductive-set

$parts :: msgterm set \Rightarrow msgterm set$

for $H :: msgterm set$

where

$Inj [intro]: X \in H \implies X \in parts H$
| *Fst*: $\langle X, - \rangle \in parts H \implies X \in parts H$
| *Snd*: $\langle -, Y \rangle \in parts H \implies Y \in parts H$

| *Lst*: $\llbracket L \text{ } xs \in \text{parts } H; X \in \text{set } xs \rrbracket \implies X \in \text{parts } H$
 | *FSt*: $\llbracket FS \text{ } xs \in \text{parts } H; X \in xs \rrbracket \implies X \in \text{parts } H$

 | *Body*: $\text{Crypt } K \text{ } X \in \text{parts } H \implies X \in \text{parts } H$

Monotonicity

lemma *parts-mono*: $G \subseteq H \implies \text{parts } G \subseteq \text{parts } H$
 $\langle \text{proof} \rangle$

Equations hold because constructors are injective.

lemma *Other-image-eq* [simp]: $(AS \text{ } x \in AS'A) = (x:A)$
 $\langle \text{proof} \rangle$

lemma *Key-image-eq* [simp]: $(Key \text{ } x \in Key'A) = (x:A)$
 $\langle \text{proof} \rangle$

lemma *AS-Key-image-eq* [simp]: $(AS \text{ } x \notin Key'A)$
 $\langle \text{proof} \rangle$

lemma *Num-Key-image-eq* [simp]: $(Num \text{ } x \notin Key'A)$
 $\langle \text{proof} \rangle$

1.4.1 keysFor operator

lemma *keysFor-empty* [simp]: $\text{keysFor } \{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *keysFor-Un* [simp]: $\text{keysFor } (H \cup H') = \text{keysFor } H \cup \text{keysFor } H'$
 $\langle \text{proof} \rangle$

lemma *keysFor-UN* [simp]: $\text{keysFor } (\bigcup_{i \in A}. H \text{ } i) = (\bigcup_{i \in A}. \text{keysFor } (H \text{ } i))$
 $\langle \text{proof} \rangle$

Monotonicity

lemma *keysFor-mono*: $G \subseteq H \implies \text{keysFor } G \subseteq \text{keysFor } H$
 $\langle \text{proof} \rangle$

lemma *keysFor-insert-AS* [simp]: $\text{keysFor } (\text{insert } (AS \text{ } A) \text{ } H) = \text{keysFor } H$
 $\langle \text{proof} \rangle$

lemma *keysFor-insert-Num* [simp]: $\text{keysFor } (\text{insert } (Num \text{ } N) \text{ } H) = \text{keysFor } H$
 $\langle \text{proof} \rangle$

lemma *keysFor-insert-Key* [simp]: $\text{keysFor } (\text{insert } (Key \text{ } K) \text{ } H) = \text{keysFor } H$
 $\langle \text{proof} \rangle$

lemma *keysFor-insert-Nonce* [simp]: $\text{keysFor } (\text{insert } (Nonce \text{ } n) \text{ } H) = \text{keysFor } H$
 $\langle \text{proof} \rangle$

lemma *keysFor-insert-L* [simp]: $\text{keysFor } (\text{insert } (L \text{ } X) \text{ } H) = \text{keysFor } H$
 $\langle \text{proof} \rangle$

lemma *keysFor-insert-FS* [simp]: $\text{keysFor } (\text{insert } (FS \ X) \ H) = \text{keysFor } H$
 $\langle \text{proof} \rangle$

lemma *keysFor-insert-Hash* [simp]: $\text{keysFor } (\text{insert } (Hash \ X) \ H) = \text{keysFor } H$
 $\langle \text{proof} \rangle$

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

lemma *keysFor-insert-Crypt* [simp]:
 $\text{keysFor } (\text{insert } (Crypt \ K \ X) \ H) = \text{insert } (\text{invKey } K) (\text{keysFor } H)$
 $\langle \text{proof} \rangle$

lemma *keysFor-image-Key* [simp]: $\text{keysFor } (Key' E) = \{\}$
 $\langle \text{proof} \rangle$

lemma *Crypt-imp-invKey-keysFor*: $Crypt \ K \ X \in H \implies \text{invKey } K \in \text{keysFor } H$
 $\langle \text{proof} \rangle$

1.4.2 Inductive relation "parts"

lemma *MPair-parts*:

$$\begin{aligned} & \llbracket \\ & \quad \langle X, Y \rangle \in \text{parts } H; \\ & \quad \llbracket X \in \text{parts } H; Y \in \text{parts } H \rrbracket \implies P \\ & \rrbracket \implies P \end{aligned}$$
 $\langle \text{proof} \rangle$

lemma *L-parts*:

$$\begin{aligned} & \llbracket \\ & \quad L \ l \in \text{parts } H; \\ & \quad \llbracket \text{set } l \subseteq \text{parts } H \rrbracket \implies P \\ & \rrbracket \implies P \end{aligned}$$
 $\langle \text{proof} \rangle$

lemma *FS-parts*:

$$\begin{aligned} & \llbracket \\ & \quad FS \ l \in \text{parts } H; \\ & \quad \llbracket \text{fset } l \subseteq \text{parts } H \rrbracket \implies P \\ & \rrbracket \implies P \end{aligned}$$
 $\langle \text{proof} \rangle$

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*: $H \subseteq \text{parts } H$
 $\langle \text{proof} \rangle$

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

lemma *parts-empty* [simp]: $\text{parts}\{\} = \{\}$
 $\langle \text{proof} \rangle$

lemma *parts-emptyE* [elim!]: $X \in \text{parts}\{\} \implies P$
 $\langle \text{proof} \rangle$

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

lemma *parts-singleton*: $X \in \text{parts } H \implies \exists Y \in H. X \in \text{parts } \{Y\}$
 $\langle \text{proof} \rangle$

lemma *parts-singleton-set*: $x \in \text{parts } \{s . P s\} \implies \exists Y. P Y \wedge x \in \text{parts } \{Y\}$
 $\langle \text{proof} \rangle$

lemma *parts-singleton-set-rev*: $\llbracket x \in \text{parts } \{Y\}; P Y \rrbracket \implies x \in \text{parts } \{s . P s\}$
 $\langle \text{proof} \rangle$

lemma *parts-Hash*: $\llbracket \bigwedge t . t \in H \implies \exists t' . t = \text{Hash } t' \rrbracket \implies \text{parts } H = H$
 $\langle \text{proof} \rangle$

Unions

lemma *parts-Un-subset1*: $\text{parts } G \cup \text{parts } H \subseteq \text{parts}(G \cup H)$
 $\langle \text{proof} \rangle$

lemma *parts-Un-subset2*: $\text{parts}(G \cup H) \subseteq \text{parts } G \cup \text{parts } H$
 $\langle \text{proof} \rangle$

lemma *parts-Un* [simp]: $\text{parts}(G \cup H) = \text{parts } G \cup \text{parts } H$
 $\langle \text{proof} \rangle$

lemma *parts-insert*: $\text{parts } (\text{insert } X H) = \text{parts } \{X\} \cup \text{parts } H$
 $\langle \text{proof} \rangle$

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

lemma *parts-insert2*:
 $\text{parts } (\text{insert } X (\text{insert } Y H)) = \text{parts } \{X\} \cup \text{parts } \{Y\} \cup \text{parts } H$
 $\langle \text{proof} \rangle$

lemma *parts-two*: $\llbracket x \in \text{parts } \{e1, e2\}; x \notin \text{parts } \{e1\} \rrbracket \implies x \in \text{parts } \{e2\}$
 $\langle \text{proof} \rangle$

Added to simplify arguments to parts, analz and synth.

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

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

lemma *parts-insert-subset*: $\text{insert } X (\text{parts } H) \subseteq \text{parts}(\text{insert } X H)$
 $\langle \text{proof} \rangle$

Idempotence

lemma *parts-partsD* [*dest!*]: $X \in \text{parts } (\text{parts } H) \implies X \in \text{parts } H$
<proof>

lemma *parts-idem* [*simp*]: $\text{parts } (\text{parts } H) = \text{parts } H$
<proof>

lemma *parts-subset-iff* [*simp*]: $(\text{parts } G \subseteq \text{parts } H) = (G \subseteq \text{parts } H)$
<proof>

Transitivity

lemma *parts-trans*: $\llbracket X \in \text{parts } G; \ G \subseteq \text{parts } H \rrbracket \implies X \in \text{parts } H$
<proof>

Unions, revisited

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

lemma *parts-split*: $\text{parts } H = \bigcup \{ \text{parts } \{h\} \mid h . h \in H \}$
<proof>

Cut

lemma *parts-cut*:
 $\llbracket Y \in \text{parts } (\text{insert } X \ G); \ X \in \text{parts } H \rrbracket \implies Y \in \text{parts } (G \cup H)$
<proof>

lemma *parts-cut-eq* [*simp*]: $X \in \text{parts } H \implies \text{parts } (\text{insert } X \ H) = \text{parts } H$
<proof>

Rewrite rules for pulling out atomic messages

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

lemma *parts-insert-AS* [*simp*]:
 $\text{parts } (\text{insert } (AS \ agt) \ H) = \text{insert } (AS \ agt) (\text{parts } H)$
<proof>

lemma *parts-insert-Epsilon* [*simp*]:
 $\text{parts } (\text{insert } \varepsilon \ H) = \text{insert } \varepsilon (\text{parts } H)$
<proof>

lemma *parts-insert-Num* [*simp*]:
 $\text{parts } (\text{insert } (Num \ N) \ H) = \text{insert } (Num \ N) (\text{parts } H)$
<proof>

lemma *parts-insert-Key* [*simp*]:
 $\text{parts } (\text{insert } (Key \ K) \ H) = \text{insert } (Key \ K) (\text{parts } H)$
<proof>

lemma *parts-insert-Nonce* [*simp*]:

$parts (insert (Nonce\ n)\ H) = insert (Nonce\ n) (parts\ H)$
 $\langle proof \rangle$

lemma *parts-insert-Hash* [simp]:
 $parts (insert (Hash\ X)\ H) = insert (Hash\ X) (parts\ H)$
 $\langle proof \rangle$

lemma *parts-insert-Crypt* [simp]:
 $parts (insert (Crypt\ K\ X)\ H) = insert (Crypt\ K\ X) (parts (insert\ X\ H))$
 $\langle proof \rangle$

lemma *parts-insert-MPair* [simp]:
 $parts (insert \langle X, Y \rangle H) =$
 $insert \langle X, Y \rangle (parts (insert\ X (insert\ Y\ H)))$
 $\langle proof \rangle$

lemma *parts-insert-L* [simp]:
 $parts (insert (L\ xs)\ H) =$
 $insert (L\ xs) (parts ((set\ xs) \cup H))$
 $\langle proof \rangle$

lemma *parts-insert-FS* [simp]:
 $parts (insert (FS\ xs)\ H) =$
 $insert (FS\ xs) (parts ((fset\ xs) \cup H))$
 $\langle proof \rangle$

lemma *parts-image-Key* [simp]: $parts (Key'N) = Key'N$
 $\langle proof \rangle$

Parts of lists and finite sets.

lemma *parts-list-set* :
 $parts (L'ls) = (L'ls) \cup (\bigcup l \in ls. parts (set\ l))$
 $\langle proof \rangle$

lemma *parts-insert-list-set* :
 $parts ((L'ls) \cup H) = (L'ls) \cup (\bigcup l \in ls. parts ((set\ l))) \cup parts\ H$
 $\langle proof \rangle$

lemma *parts-fset-set* :
 $parts (FS'ls) = (FS'ls) \cup (\bigcup l \in ls. parts (fset\ l))$
 $\langle proof \rangle$

suffix of parts

lemma *suffix-in-parts*:
 $suffix (x\#xs)\ ys \implies x \in parts\ \{L\ ys\}$
 $\langle proof \rangle$

lemma *parts-L-set*:
 $\llbracket x \in parts\ \{L\ ys\}; ys \in St \rrbracket \implies x \in parts (L'St)$
 $\langle proof \rangle$

lemma *suffix-in-parts-set*:

$\llbracket \text{suffix } (x \# xs) \text{ } ys; ys \in St \rrbracket \implies x \in \text{parts } (L \cdot St)$
 $\langle \text{proof} \rangle$

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* $\Rightarrow$ *msgterm set*

for *H* :: *msgterm set*

where

Inj [*intro,simp*] : $X \in H \implies X \in \text{analz } H$
 $| \text{Fst}: \quad \langle X, Y \rangle \in \text{analz } H \implies X \in \text{analz } H$
 $| \text{Snd}: \quad \langle X, Y \rangle \in \text{analz } H \implies Y \in \text{analz } H$
 $| \text{Lst}: \quad (L \ y) \in \text{analz } H \implies x \in \text{set } (y) \implies x \in \text{analz } H$
 $| \text{FSt}: \quad \llbracket FS \ xs \in \text{analz } H; X \mid \in \ xs \rrbracket \implies X \in \text{analz } H$
 $| \text{Decrypt } [dest]: \quad \llbracket \text{Crypt } K \ X \in \text{analz } H; \text{Key } (\text{invKey } K) \in \text{analz } H \rrbracket \implies X \in \text{analz } H$

Monotonicity; Lemma 1 of Lowe's paper

lemma *analz-mono*: $G \subseteq H \implies \text{analz}(G) \subseteq \text{analz}(H)$

$\langle \text{proof} \rangle$

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

Making it safe speeds up proofs

lemma *MPair-analz* [*elim!*]:

$\llbracket \langle X, Y \rangle \in \text{analz } H; \llbracket X \in \text{analz } H; Y \in \text{analz } H \rrbracket \implies P \rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *L-analz* [*elim!*]:

$\llbracket L \ l \in \text{analz } H; \llbracket \text{set } l \subseteq \text{analz } H \rrbracket \implies P \rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *FS-analz* [*elim!*]:

$\llbracket FS \ l \in \text{analz } H; \llbracket \text{fset } l \subseteq \text{analz } H \rrbracket \implies P \rrbracket \implies P$
 $\langle \text{proof} \rangle$

thm *parts.FSt subsetI*

lemma *analz-increasing*: $H \subseteq \text{analz}(H)$

$\langle \text{proof} \rangle$

lemma *analz-subset-parts*: $\text{analz } H \subseteq \text{parts } H$
 $\langle \text{proof} \rangle$

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

lemma *no-crypt-analz-is-parts*:
 $\neg (\exists K X . \text{Crypt } K X \in \text{parts } A) \implies \text{analz } A = \text{parts } A$
 $\langle \text{proof} \rangle$

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

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

lemma *parts-analz* [simp]: $\text{parts } (\text{analz } H) = \text{parts } H$
 $\langle \text{proof} \rangle$

lemma *analz-parts* [simp]: $\text{analz } (\text{parts } H) = \text{parts } H$
 $\langle \text{proof} \rangle$

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

General equational properties

lemma *analz-empty* [simp]: $\text{analz } \{\} = \{\}$
 $\langle \text{proof} \rangle$

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*: $\text{analz}(G) \cup \text{analz}(H) \subseteq \text{analz}(G \cup H)$
 $\langle \text{proof} \rangle$

lemma *analz-insert*: $\text{insert } X (\text{analz } H) \subseteq \text{analz}(\text{insert } X H)$
 $\langle \text{proof} \rangle$

Rewrite rules for pulling out atomic messages

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

lemma *analz-insert-AS* [simp]:
 $\text{analz } (\text{insert } (AS \text{ agt}) H) = \text{insert } (AS \text{ agt}) (\text{analz } H)$
 $\langle \text{proof} \rangle$

lemma *analz-insert-Num* [simp]:
 $\text{analz } (\text{insert } (Num N) H) = \text{insert } (Num N) (\text{analz } H)$
 $\langle \text{proof} \rangle$

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

lemma *analz-insert-Key* [simp]:
 $K \notin \text{keysFor } (\text{analz } H) \implies$
 $\text{analz } (\text{insert } (Key K) H) = \text{insert } (Key K) (\text{analz } H)$
 $\langle \text{proof} \rangle$

lemma *analz-insert-LEmpty* [simp]:
 $\text{analz } (\text{insert } (L \ []) H) = \text{insert } (L \ []) (\text{analz } H)$
 $\langle \text{proof} \rangle$

lemma *analz-insert-L* [simp]:
 $\text{analz } (\text{insert } (L \ l) H) = \text{insert } (L \ l) (\text{analz } (\text{set } l \cup H))$
 $\langle \text{proof} \rangle$

lemma *analz-insert-FS* [simp]:
 $\text{analz } (\text{insert } (FS \ l) H) = \text{insert } (FS \ l) (\text{analz } (fset \ l \cup H))$
 $\langle \text{proof} \rangle$

lemma $L \ [] \in \text{analz } \{L[L \ []]\}$
 $\langle \text{proof} \rangle$

lemma *analz-insert-Hash* [simp]:
 $\text{analz } (\text{insert } (Hash \ X) H) = \text{insert } (Hash \ X) (\text{analz } H)$
 $\langle \text{proof} \rangle$

lemma *analz-insert-MPair* [simp]:
 $\text{analz } (\text{insert } \langle X, Y \rangle H) =$
 $\text{insert } \langle X, Y \rangle (\text{analz } (\text{insert } X (\text{insert } Y H)))$
 $\langle \text{proof} \rangle$

Can pull out enCrypted message if the Key is not known

lemma *analz-insert-Crypt*:
 $\text{Key } (invKey \ K) \notin \text{analz } H \implies$
 $\text{analz } (\text{insert } (Crypt \ K \ X) H) = \text{insert } (Crypt \ K \ X) (\text{analz } H)$
 $\langle \text{proof} \rangle$

lemma *analz-insert-Decrypt1*:
 $\text{Key } (invKey \ K) \in \text{analz } H \implies$
 $\text{analz } (\text{insert } (Crypt \ K \ X) H) \subseteq$
 $\text{insert } (Crypt \ K \ X) (\text{analz } (\text{insert } X H))$
 $\langle \text{proof} \rangle$

lemma *analz-insert-Decrypt2*:
 $\text{Key } (invKey \ K) \in \text{analz } H \implies$
 $\text{insert } (Crypt \ K \ X) (\text{analz } (\text{insert } X H)) \subseteq$
 $\text{analz } (\text{insert } (Crypt \ K \ X) H)$
 $\langle \text{proof} \rangle$

lemma *analz-insert-Decrypt*:
 $\text{Key } (invKey \ K) \in \text{analz } H \implies$
 $\text{analz } (\text{insert } (Crypt \ K \ X) H) =$
 $\text{insert } (Crypt \ K \ X) (\text{analz } (\text{insert } X H))$
 $\langle \text{proof} \rangle$

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 $\text{analz } (\text{insert } (Crypt \ K \ X) H)$

lemma *analz-Crypt-if* [simp]:

$$\text{analz } (\text{insert } (\text{Crypt } K \ X) \ H) =$$

$$(\text{if } (\text{Key } (\text{invKey } K) \in \text{analz } H)$$

$$\text{then insert } (\text{Crypt } K \ X) (\text{analz } (\text{insert } X \ H))$$

$$\text{else insert } (\text{Crypt } K \ X) (\text{analz } H))$$

$$\langle \text{proof} \rangle$$

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

lemma *analz-insert-Crypt-subset*:

$$\text{analz } (\text{insert } (\text{Crypt } K \ X) \ H) \subseteq$$

$$\text{insert } (\text{Crypt } K \ X) (\text{analz } (\text{insert } X \ H))$$

$$\langle \text{proof} \rangle$$

lemma *analz-image-Key* [simp]: $\text{analz } (\text{Key}'N) = \text{Key}'N$

$$\langle \text{proof} \rangle$$

Idempotence and transitivity

lemma *analz-analzD* [dest!]: $X \in \text{analz } (\text{analz } H) \implies X \in \text{analz } H$

$$\langle \text{proof} \rangle$$

lemma *analz-idem* [simp]: $\text{analz } (\text{analz } H) = \text{analz } H$

$$\langle \text{proof} \rangle$$

lemma *analz-subset-iff* [simp]: $(\text{analz } G \subseteq \text{analz } H) = (G \subseteq \text{analz } H)$

$$\langle \text{proof} \rangle$$

lemma *analz-trans*: $\llbracket X \in \text{analz } G; \ G \subseteq \text{analz } H \rrbracket \implies X \in \text{analz } H$

$$\langle \text{proof} \rangle$$

Cut; Lemma 2 of Lowe

lemma *analz-cut*: $\llbracket Y \in \text{analz } (\text{insert } X \ H); \ X \in \text{analz } H \rrbracket \implies Y \in \text{analz } H$

$$\langle \text{proof} \rangle$$

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*: $X \in \text{analz } H \implies \text{analz } (\text{insert } X \ H) = \text{analz } H$

$$\langle \text{proof} \rangle$$

A congruence rule for "analz"

lemma *analz-subset-cong*:

$$\llbracket \text{analz } G \subseteq \text{analz } G'; \ \text{analz } H \subseteq \text{analz } H' \rrbracket$$

$$\implies \text{analz } (G \cup H) \subseteq \text{analz } (G' \cup H')$$

$$\langle \text{proof} \rangle$$

lemma *analz-cong*:

$$\llbracket \text{analz } G = \text{analz } G'; \ \text{analz } H = \text{analz } H' \rrbracket$$

$$\implies \text{analz } (G \cup H) = \text{analz } (G' \cup H')$$

$$\langle \text{proof} \rangle$$

lemma *analz-insert-cong*:

$$\text{analz } H = \text{analz } H' \implies \text{analz}(\text{insert } X \ H) = \text{analz}(\text{insert } X \ H')$$

$\langle proof \rangle$

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

lemma *analz-trivial*:

$$\begin{aligned} & \llbracket \\ & \quad \forall X Y. \langle X, Y \rangle \notin H; \forall xs. L\ xs \notin H; \forall xs. FS\ xs \notin H; \\ & \quad \forall X K. Crypt\ K\ X \notin H \\ & \rrbracket \implies analz\ H = H \end{aligned}$$
 $\langle proof \rangle$

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

lemma *analz-UN-analz-lemma*:

$$X \in analz\ (\bigcup i \in A. analz\ (H\ i)) \implies X \in analz\ (\bigcup i \in A. H\ i)$$
 $\langle proof \rangle$

lemma *analz-UN-analz [simp]*: $analz\ (\bigcup i \in A. analz\ (H\ i)) = analz\ (\bigcup i \in A. H\ i)$ $\langle proof \rangle$

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*:

$$\begin{aligned} & \neg(\exists K. Key\ K \in analz\ H) \\ & \implies analz\ H = \bigcup \{ analz\ \{h\} \mid h. h \in H \} \end{aligned}$$
 $\langle proof \rangle$

lemma *analz-Un-eq*:

assumes $\neg(\exists K. Key\ K \in analz\ H)$ **and** $\neg(\exists K. Key\ K \in analz\ G)$
shows $analz\ (H \cup G) = analz\ H \cup analz\ G$ $\langle proof \rangle$

lemma *analz-Un-eq-Crypt*:

assumes $\neg(\exists K. Key\ K \in analz\ G)$ **and** $\neg(\exists K X. Crypt\ K\ X \in analz\ G)$
shows $analz\ (H \cup G) = analz\ H \cup analz\ G$ $\langle proof \rangle$

lemma *analz-list-set* :

$$\begin{aligned} & \neg(\exists K. Key\ K \in analz\ (L'ls)) \\ & \implies analz\ (L'ls) = (L'ls) \cup (\bigcup l \in ls. analz\ (set\ l)) \end{aligned}$$
 $\langle proof \rangle$

lemma *analz-fset-set* :

$$\begin{aligned} & \neg(\exists K. Key\ K \in analz\ (FS'ls)) \\ & \implies analz\ (FS'ls) = (FS'ls) \cup (\bigcup l \in ls. analz\ (fset\ l)) \end{aligned}$$
 $\langle proof \rangle$

1.4.4 Inductive relation "synth"

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

inductive-set

synth :: msgterm set $\Rightarrow$ msgterm set

for *H* :: msgterm set

where

Inj [intro]: $X \in H \Longrightarrow X \in \text{synth } H$
 $|\ \varepsilon$ [simp,intro]: $\varepsilon \in \text{synth } H$
 $|\ AS$ [simp,intro]: $AS\ agt \in \text{synth } H$
 $|\ Num$ [simp,intro]: $Num\ n \in \text{synth } H$
 $|\ Lst$ [intro]: $\llbracket \bigwedge x . x \in \text{set } xs \Longrightarrow x \in \text{synth } H \rrbracket \Longrightarrow L\ xs \in \text{synth } H$
 $|\ FSt$ [intro]: $\llbracket \bigwedge x . x \in \text{fset } xs \Longrightarrow x \in \text{synth } H;$
 $\quad \bigwedge x\ ys . x \in \text{fset } xs \Longrightarrow x \neq FS\ ys \rrbracket$
 $\quad \Longrightarrow FS\ xs \in \text{synth } H$
 $|\ Hash$ [intro]: $X \in \text{synth } H \Longrightarrow Hash\ X \in \text{synth } H$
 $|\ MPair$ [intro]: $\llbracket X \in \text{synth } H; Y \in \text{synth } H \rrbracket \Longrightarrow \langle X, Y \rangle \in \text{synth } H$
 $|\ Crypt$ [intro]: $\llbracket X \in \text{synth } H; Key\ K \in H \rrbracket \Longrightarrow Crypt\ K\ X \in \text{synth } H$

Monotonicity

lemma *synth-mono*: $G \subseteq H \Longrightarrow \text{synth}(G) \subseteq \text{synth}(H)$

<proof>

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

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

lemma *synth-increasing*: $H \subseteq \text{synth}(H)$

<proof>

lemma *synth-analz-self*: $x \in H \Longrightarrow x \in \text{synth } (\text{analz } H)$

<proof>

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*: $\text{synth}(G) \cup \text{synth}(H) \subseteq \text{synth}(G \cup H)$

<proof>

lemma *synth-insert*: $\text{insert } X\ (\text{synth } H) \subseteq \text{synth}(\text{insert } X\ H)$

<proof>

Idempotence and transitivity

lemma *synth-synthD* [dest!]: $X \in \text{synth } (\text{synth } H) \Longrightarrow X \in \text{synth } H$

<proof>

lemma *synth-idem*: $\text{synth } (\text{synth } H) = \text{synth } H$

<proof>

lemma *synth-subset-iff* [simp]: $(\text{synth } G \subseteq \text{synth } H) = (G \subseteq \text{synth } H)$
 $\langle \text{proof} \rangle$

lemma *synth-trans*: $\llbracket X \in \text{synth } G; G \subseteq \text{synth } H \rrbracket \implies X \in \text{synth } H$
 $\langle \text{proof} \rangle$

Cut; Lemma 2 of Lowe

lemma *synth-cut*: $\llbracket Y \in \text{synth } (\text{insert } X \ H); X \in \text{synth } H \rrbracket \implies Y \in \text{synth } H$
 $\langle \text{proof} \rangle$

lemma *Nonce-synth-eq* [simp]: $(\text{Nonce } N \in \text{synth } H) = (\text{Nonce } N \in H)$
 $\langle \text{proof} \rangle$

lemma *Key-synth-eq* [simp]: $(\text{Key } K \in \text{synth } H) = (\text{Key } K \in H)$
 $\langle \text{proof} \rangle$

lemma *Crypt-synth-eq* [simp]:
 $\text{Key } K \notin H \implies (\text{Crypt } K \ X \in \text{synth } H) = (\text{Crypt } K \ X \in H)$
 $\langle \text{proof} \rangle$

lemma *keysFor-synth* [simp]:
 $\text{keysFor } (\text{synth } H) = \text{keysFor } H \cup \text{invKey} \{K. \text{Key } K \in H\}$
 $\langle \text{proof} \rangle$

lemma *L-cons-synth* [simp]:
 $(\text{set } xs \subseteq H) \implies (L \ xs \in \text{synth } H)$
 $\langle \text{proof} \rangle$

lemma *FS-cons-synth* [simp]:
 $\llbracket \text{fset } xs \subseteq H; \bigwedge x \ ys. x \in \text{fset } xs \implies x \neq \text{FS } ys; \text{fcard } xs \neq \text{Suc } 0 \rrbracket \implies (\text{FS } xs \in \text{synth } H)$
 $\langle \text{proof} \rangle$

Combinations of parts, analz and synth

lemma *parts-synth* [simp]: $\text{parts } (\text{synth } H) = \text{parts } H \cup \text{synth } H$
 $\langle \text{proof} \rangle$

lemma *analz-analz-Un* [simp]: $\text{analz } (\text{analz } G \cup H) = \text{analz } (G \cup H)$
 $\langle \text{proof} \rangle$

lemma *analz-synth-Un* [simp]: $\text{analz } (\text{synth } G \cup H) = \text{analz } (G \cup H) \cup \text{synth } G$
 $\langle \text{proof} \rangle$

lemma *analz-synth* [simp]: $\text{analz } (\text{synth } H) = \text{analz } H \cup \text{synth } H$
 $\langle \text{proof} \rangle$

lemma *analz-Un-analz* [simp]: $\text{analz } (G \cup \text{analz } H) = \text{analz } (G \cup H)$
 $\langle \text{proof} \rangle$

lemma *analz-synth-Un2* [simp]: $\text{analz } (G \cup \text{synth } H) = \text{analz } (G \cup H) \cup \text{synth } H$

$\langle proof \rangle$

For reasoning about the Fake rule in traces

lemma *parts-insert-subset-Un*: $X \in G \implies \text{parts}(\text{insert } X \ H) \subseteq \text{parts } G \cup \text{parts } H$
 $\langle proof \rangle$

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

lemma *Fake-parts-insert*:
 $X \in \text{synth } (\text{analz } H) \implies$
 $\text{parts } (\text{insert } X \ H) \subseteq \text{synth } (\text{analz } H) \cup \text{parts } H$
 $\langle proof \rangle$

lemma *Fake-parts-insert-in-Un*:
 $\llbracket Z \in \text{parts } (\text{insert } X \ H); \ X \in \text{synth } (\text{analz } H) \rrbracket$
 $\implies Z \in \text{synth } (\text{analz } H) \cup \text{parts } H$
 $\langle proof \rangle$

H is sometimes *Key* ‘ $KK \cup \text{spies evs}$ ’, so can’t put $G = H$.

lemma *Fake-analz-insert*:
 $X \in \text{synth } (\text{analz } G) \implies$
 $\text{analz } (\text{insert } X \ H) \subseteq \text{synth } (\text{analz } G) \cup \text{analz } (G \cup H)$
 $\langle proof \rangle$

lemma *analz-conj-parts [simp]*:
 $(X \in \text{analz } H \ \& \ X \in \text{parts } H) = (X \in \text{analz } H)$
 $\langle proof \rangle$

lemma *analz-disj-parts [simp]*:
 $(X \in \text{analz } H \mid X \in \text{parts } H) = (X \in \text{parts } H)$
 $\langle proof \rangle$

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

lemma *MPair-synth-analz [iff]*:
 $(\langle X, Y \rangle \in \text{synth } (\text{analz } H)) =$
 $(X \in \text{synth } (\text{analz } H) \ \& \ Y \in \text{synth } (\text{analz } H))$
 $\langle proof \rangle$

lemma *L-cons-synth-analz [iff]*:
 $(L \ xs \in \text{synth } (\text{analz } H)) =$
 $(\text{set } xs \subseteq \text{synth } (\text{analz } H))$
 $\langle proof \rangle$

lemma *L-cons-synth-parts [iff]*:
 $(L \ xs \in \text{synth } (\text{parts } H)) =$
 $(\text{set } xs \subseteq \text{synth } (\text{parts } H))$
 $\langle proof \rangle$

lemma *FS-cons-synth-analz [iff]*:
 $\llbracket \bigwedge x \ ys . x \in \text{fset } xs \implies x \neq \text{FS } ys; \ \text{fcard } xs \neq \text{Suc } 0 \rrbracket \implies$
 $(\text{FS } xs \in \text{synth } (\text{analz } H)) =$

$(fset\ xs \subseteq synth\ (analz\ H))$
 $\langle proof \rangle$

lemma *FS-cons-synth-parts* [iff]:
 $\llbracket \bigwedge x\ ys.\ x \in fset\ xs \implies x \neq FS\ ys; fcard\ xs \neq Suc\ 0 \rrbracket \implies$
 $(FS\ xs \in synth\ (parts\ H)) =$
 $(fset\ xs \subseteq synth\ (parts\ H))$
 $\langle proof \rangle$

lemma *Crypt-synth-analz*:
 $\llbracket Key\ K \in analz\ H; Key\ (invKey\ K) \in analz\ H \rrbracket$
 $\implies (Crypt\ K\ X \in synth\ (analz\ H)) = (X \in synth\ (analz\ H))$
 $\langle proof \rangle$

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

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]: $\text{Crypt } K \ X \notin \text{Key } A$
 $\langle \text{proof} \rangle$

lemma *Hash-notin-image-Key* [simp]: $\text{Hash } X \notin \text{Key } A$
 $\langle \text{proof} \rangle$

lemma *synth-analz-mono*: $G \subseteq H \implies \text{synth } (\text{analz } (G)) \subseteq \text{synth } (\text{analz } (H))$
 $\langle \text{proof} \rangle$

lemma *synth-parts-mono*: $G \subseteq H \implies \text{synth } (\text{parts } G) \subseteq \text{synth } (\text{parts } H)$
 $\langle \text{proof} \rangle$

lemma *Fake-analz-eq* [simp]:
 $X \in \text{synth } (\text{analz } H) \implies \text{synth } (\text{analz } (\text{insert } X \ H)) = \text{synth } (\text{analz } H)$
 $\langle \text{proof} \rangle$

Two generalizations of *analz-insert-eq*

lemma *gen-analz-insert-eq* [rule-format]:
 $X \in \text{analz } H \implies \text{ALL } G. H \subseteq G \longrightarrow \text{analz } (\text{insert } X \ G) = \text{analz } G$
 $\langle \text{proof} \rangle$

lemma *Fake-parts-sing*:
 $X \in \text{synth } (\text{analz } H) \implies \text{parts } \{X\} \subseteq \text{synth } (\text{analz } H) \cup \text{parts } H$
 $\langle \text{proof} \rangle$

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 $\text{analz } H \subseteq \text{synth } (\text{analz } H')$
shows $\text{analz } (\text{insert } X \ H) \subseteq \text{synth } (\text{analz } (\text{insert } X \ H'))$
 $\langle \text{proof} \rangle$

lemma *synth-parts-insert*:
assumes $\text{parts } H \subseteq \text{synth } (\text{parts } H')$
shows $\text{parts } (\text{insert } X \ H) \subseteq \text{synth } (\text{parts } (\text{insert } X \ H'))$
 $\langle \text{proof} \rangle$

lemma *parts-insert-subset-impl*:
 $\llbracket x \in \text{parts } (\text{insert } a \ G); x \in \text{parts } G \implies x \in \text{synth } (\text{parts } H); a \in \text{synth } (\text{parts } H) \rrbracket$
 $\implies x \in \text{synth } (\text{parts } H)$
 $\langle \text{proof} \rangle$

lemma *synth-parts-subset-elem*:
 $\llbracket A \subseteq \text{synth } (\text{parts } B); x \in \text{parts } A \rrbracket \implies x \in \text{synth } (\text{parts } B)$
 $\langle \text{proof} \rangle$

lemma *synth-parts-subset*:

$A \subseteq \text{synth } (\text{parts } B) \implies \text{parts } A \subseteq \text{synth } (\text{parts } B)$
 $\langle \text{proof} \rangle$

lemma *parts-synth-parts[simp]*: $\text{parts } (\text{synth } (\text{parts } H)) = \text{synth } (\text{parts } H)$
 $\langle \text{proof} \rangle$

lemma *synth-parts-trans*:
assumes $A \subseteq \text{synth } (\text{parts } B)$ **and** $B \subseteq \text{synth } (\text{parts } C)$
shows $A \subseteq \text{synth } (\text{parts } C)$
 $\langle \text{proof} \rangle$

lemma *synth-parts-trans-elem*:
assumes $x \in A$ **and** $A \subseteq \text{synth } (\text{parts } B)$ **and** $B \subseteq \text{synth } (\text{parts } C)$
shows $x \in \text{synth } (\text{parts } C)$
 $\langle \text{proof} \rangle$

lemma *synth-un-parts-split*:
assumes $x \in \text{synth } (\text{parts } A \cup \text{parts } B)$
and $\bigwedge x . x \in A \implies x \in \text{synth } (\text{parts } C)$
and $\bigwedge x . x \in B \implies x \in \text{synth } (\text{parts } C)$
shows $x \in \text{synth } (\text{parts } C)$
 $\langle \text{proof} \rangle$

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* $\Rightarrow$ *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]: $\llbracket \bigwedge x . x \in \text{set } xs \implies \text{normalized } x \rrbracket \implies \text{normalized } (L \text{ } xs)$
 FSt [intro]: $\llbracket \bigwedge x . x \in \text{fset } xs \implies \text{normalized } x; \bigwedge x \text{ } ys . x \in \text{fset } xs \implies x \neq FS \text{ } ys \rrbracket \implies \text{normalized } (FS \text{ } xs)$
 $Hash$ [intro]: *normalized* $X \implies \text{normalized } (Hash \text{ } X)$
 $MPair$ [intro]: $\llbracket \text{normalized } X; \text{normalized } Y \rrbracket \implies \text{normalized } \langle X, Y \rangle$
 $Crypt$ [intro]: $\llbracket \text{normalized } X \rrbracket \implies \text{normalized } (Crypt \text{ } K \text{ } X)$

thm *normalized.simps*

find-theorems *normalized*

Examples

lemma *normalized* ($FS \{ |Hash (FS \{ |Num 0| \}), Num 0| \}$) $\langle \text{proof} \rangle$

lemma $\neg \text{normalized } (FS \{ |FS \{ |Num 0| \}, Num 0| \})$ $\langle \text{proof} \rangle$

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

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

lemma *normalized-synth*[elim!]: $\llbracket t \in \text{synth } H; \bigwedge t. t \in H \implies \text{normalized } t \rrbracket \implies \text{normalized } t$
 <proof>

lemma *normalized-parts*[elim!]: $\llbracket t \in \text{parts } H; \bigwedge t. t \in H \implies \text{normalized } t \rrbracket \implies \text{normalized } t$
 <proof>

lemma *normalized-analz*[elim!]: $\llbracket t \in \text{analz } H; \bigwedge t. t \in H \implies \text{normalized } t \rrbracket \implies \text{normalized } t$
 <proof>

Properties of *normalized*

lemma *normalized-FS*[elim]: $\llbracket \text{normalized } (FS \text{ } xs); x \in xs \rrbracket \implies \text{normalized } x$
 <proof>

lemma *normalized-FS-FS*[elim]: $\llbracket \text{normalized } (FS \text{ } xs); x \in xs; x = FS \text{ } ys \rrbracket \implies \text{False}$
 <proof>

lemma *normalized-subset*: $\llbracket \text{normalized } (FS \text{ } xs); ys \subseteq xs \rrbracket \implies \text{normalized } (FS \text{ } ys)$
 <proof>

lemma *normalized-insert*[elim!]: $\text{normalized } (FS \text{ } (\text{fininsert } x \text{ } xs)) \implies \text{normalized } (FS \text{ } xs)$
 <proof>

lemma *normalized-union*:
assumes $\text{normalized } (FS \text{ } xs) \text{ normalized } (FS \text{ } ys) \text{ } zs \subseteq xs \cup ys$
shows $\text{normalized } (FS \text{ } zs)$
 <proof>

lemma *normalized-minus*[elim]:
assumes $\text{normalized } (FS \text{ } (ys \mid - \mid xs)) \text{ normalized } (FS \text{ } xs)$
shows $\text{normalized } (FS \text{ } ys)$
 <proof>

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

lemma *FS-mono*: $\llbracket zs-s = \text{fininsert } (f \text{ } (FS \text{ } zs-s)) \text{ } zs-b; \bigwedge x. \text{size } (f \text{ } x) > \text{size } x \rrbracket \implies \text{False}$
 <proof>

lemma *FS-contr*: $\llbracket zs = f \text{ } (FS \text{ } \{|zs|\}); \bigwedge x. \text{size } (f \text{ } x) > \text{size } x \rrbracket \implies \text{False}$
 <proof>

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*: $\llbracket \text{prefix } xs \text{ } ys; zs = x \# ys; \text{prefix } xs' (x \# xs) \rrbracket \implies \text{prefix } xs' \text{ } zs$
 $\langle \text{proof} \rangle$

lemma *suffix-nonempty-extendable*:

$\llbracket \text{suffix } xs \text{ } l; xs \neq l \rrbracket \implies \exists x. \text{suffix } (x \# xs) \text{ } l$
 $\langle \text{proof} \rangle$

lemma *set-suffix*:

$\llbracket x \in \text{set } l'; \text{suffix } l' \text{ } l \rrbracket \implies x \in \text{set } l$
 $\langle \text{proof} \rangle$

lemma *set-prefix*:

$\llbracket x \in \text{set } l'; \text{prefix } l' \text{ } l \rrbracket \implies x \in \text{set } l$
 $\langle \text{proof} \rangle$

lemma *set-suffix-elem*: $\text{suffix } (x \# xs) \text{ } p \implies x \in \text{set } p$
 $\langle \text{proof} \rangle$

lemma *set-prefix-elem*: $\text{prefix } (x \# xs) \text{ } p \implies x \in \text{set } p$
 $\langle \text{proof} \rangle$

lemma *Cons-suffix-set*: $x \in \text{set } y \implies \exists xs. \text{suffix } (x \# xs) \text{ } y$
 $\langle \text{proof} \rangle$

1.5.2 Fragments

definition *fragment* :: $'a \text{ list} \Rightarrow 'a \text{ list set} \Rightarrow \text{bool}$

where $\text{fragment } xs \text{ } St \longleftrightarrow (\exists zs1 \text{ } zs2. zs1 @ xs @ zs2 \in St)$

lemma *fragmentI*: $\llbracket zs1 @ xs @ zs2 \in St \rrbracket \implies \text{fragment } xs \text{ } St$
 $\langle \text{proof} \rangle$

lemma *fragmentE [elim]*: $\llbracket \text{fragment } xs \text{ } St; \bigwedge zs1 \text{ } zs2. \llbracket zs1 @ xs @ zs2 \in St \rrbracket \implies P \rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *fragment-Nil [simp]*: $\text{fragment } [] \text{ } St \longleftrightarrow St \neq \{\}$
 $\langle \text{proof} \rangle$

lemma *fragment-subset*: $\llbracket St \subseteq St'; \text{fragment } l \text{ } St \rrbracket \implies \text{fragment } l \text{ } St'$
 $\langle \text{proof} \rangle$

lemma *fragment-prefix*: $\llbracket \text{prefix } l' \text{ } l; \text{fragment } l \text{ } St \rrbracket \implies \text{fragment } l' \text{ } St$
 $\langle \text{proof} \rangle$

lemma *fragment-suffix*: $\llbracket \text{suffix } l' \text{ } l; \text{fragment } l \text{ } St \rrbracket \implies \text{fragment } l' \text{ } St$

$\langle \text{proof} \rangle$

lemma *fragment-self* [*simp*, *intro*]: $\llbracket l \in St \rrbracket \implies \text{fragment } l \text{ } St$
 $\langle \text{proof} \rangle$

lemma *fragment-prefix-self* [*simp*, *intro*]:
 $\llbracket l \in St; \text{prefix } l' \text{ } l \rrbracket \implies \text{fragment } l' \text{ } St$
 $\langle \text{proof} \rangle$

lemma *fragment-suffix-self* [*simp*, *intro*]:
 $\llbracket l \in St; \text{suffix } l' \text{ } l \rrbracket \implies \text{fragment } l' \text{ } St$
 $\langle \text{proof} \rangle$

lemma *fragment-is-prefix-suffix*:
 $\text{fragment } l \text{ } St \implies \exists \text{pre } \text{suffix} . \text{prefix } l \text{ } \text{pre} \wedge \text{suffix } \text{pre } \text{suffix} \wedge \text{suffix} \in St$
 $\langle \text{proof} \rangle$

1.5.3 Pair Fragments

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

lemma *pfragmentI*: $\llbracket (\text{ainf}, \text{zs1} @ xs @ \text{zs2}) \in St \rrbracket \implies \text{pfragment } \text{ainf } xs \text{ } St$
 $\langle \text{proof} \rangle$

lemma *pfragmentE* [*elim*]: $\llbracket \text{pfragment } \text{ainf } xs \text{ } St; \bigwedge \text{zs1 } \text{zs2}. \llbracket (\text{ainf}, \text{zs1} @ xs @ \text{zs2}) \in St \rrbracket \implies P \rrbracket \implies P$
 $\langle \text{proof} \rangle$

lemma *pfragment-prefix*:
 $\text{pfragment } \text{ainf } (xs @ ys) \text{ } St \implies \text{pfragment } \text{ainf } xs \text{ } St$
 $\langle \text{proof} \rangle$

lemma *pfragment-prefix'*:
 $\llbracket \text{pfragment } \text{ainf } ys \text{ } St; \text{prefix } xs \text{ } ys \rrbracket \implies \text{pfragment } \text{ainf } xs \text{ } St$
 $\langle \text{proof} \rangle$

lemma *pfragment-suffix*: $\llbracket \text{suffix } l' \text{ } l; \text{pfragment } \text{ainf } l \text{ } St \rrbracket \implies \text{pfragment } \text{ainf } l' \text{ } St$
 $\langle \text{proof} \rangle$

lemma *pfragment-self* [*simp*, *intro*]: $\llbracket (\text{ainf}, l) \in St \rrbracket \implies \text{pfragment } \text{ainf } l \text{ } St$
 $\langle \text{proof} \rangle$

lemma *pfragment-suffix-self* [*simp*, *intro*]:
 $\llbracket (\text{ainf}, l) \in St; \text{suffix } l' \text{ } l \rrbracket \implies \text{pfragment } \text{ainf } l' \text{ } St$
 $\langle \text{proof} \rangle$

lemma *pfragment-self-eq*:
 $\llbracket \text{pfragment } \text{ainf } l \text{ } S; \bigwedge \text{zs1 } \text{zs2} . (\text{ainf}, \text{zs1} @ l @ \text{zs2}) \in S \implies (\text{ainf}, \text{zs1} @ l' @ \text{zs2}) \in S \rrbracket \implies \text{pfragment } \text{ainf } l' \text{ } S$
 $\langle \text{proof} \rangle$

lemma *pfragment-self-eq-nil*:

$\llbracket \text{pfragment ainf } l \ S; \bigwedge \text{zs1 zs2} . (\text{ainf}, \text{zs1} @ l @ \text{zs2}) \in S \implies (\text{ainf}, l' @ \text{zs2}) \in S \rrbracket \implies \text{pfragment ainf } l' \ S$
 $\langle \text{proof} \rangle$

lemma *pfragment-cons*: $\text{pfragment ainfo } (x \# \text{fut}) \ S \implies \text{pfragment ainfo fut } S$
 $\langle \text{proof} \rangle$

1.5.4 Head and Tails

fun *head* **where** $\text{head } [] = \text{None} \mid \text{head } (x \# xs) = \text{Some } x$
fun *ifhead* **where** $\text{ifhead } [] \ n = n \mid \text{ifhead } (x \# xs) \ - = \text{Some } x$
fun *tail* **where** $\text{tail } [] = \text{None} \mid \text{tail } xs = \text{Some } (\text{last } xs)$

lemma *head-cons*: $xs \neq [] \implies \text{head } xs = \text{Some } (\text{hd } xs) \langle \text{proof} \rangle$

lemma *tail-cons*: $xs \neq [] \implies \text{tail } xs = \text{Some } (\text{last } xs) \langle \text{proof} \rangle$

lemma *tail-snoc*: $\text{tail } (xs @ [x]) = \text{Some } x \langle \text{proof} \rangle$

lemma $\forall y \ ys . l \neq ys @ [y] \implies l = []$
 $\langle \text{proof} \rangle$

lemma *tl-append2*: $\text{tl } (\text{pref} @ [a, b]) = \text{tl } (\text{pref} @ [a]) @ [b]$
 $\langle \text{proof} \rangle$

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
```

| $\neg P \text{ pre } x (\text{Some } y) \implies \text{takeW pre } (x \# y \# xs) \text{ } xo = []$
 $\langle \text{proof} \rangle$

termination

$\langle \text{proof} \rangle$

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

fun *extract* :: 'a option $\Rightarrow$ 'a list $\Rightarrow$ 'a option $\Rightarrow$ '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 [])

$\langle \text{proof} \rangle$

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 [])

$\langle \text{proof} \rangle$

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

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 []))

$\langle \text{proof} \rangle$

lemma *extract-split-tail*:

extract pre (x # xs) *nxt* =

(case xs of

[] $\Rightarrow$ (if P pre x *nxt* then *nxt* else (Some x))

| (y # ys) $\Rightarrow$ (if P pre x (Some y) then *extract* (Some x) (y # ys) *nxt* else Some x))

$\langle \text{proof} \rangle$

lemma *holds-split-tail*:

holds pre (x # xs) *nxt* $\longleftrightarrow$

(case xs of

[] $\Rightarrow$ P pre x *nxt*

| (y # ys) $\Rightarrow$ P pre x (Some y) $\wedge$ *holds* (Some x) (y # ys) *nxt*)

$\langle \text{proof} \rangle$

lemma *holds-Cons-P*:

holds pre (x # xs) *nxt* $\implies \exists y . P \text{ pre } x \ y$

$\langle \text{proof} \rangle$

lemma *holds-Cons-holds*:

holds pre (x # xs) nxt $\implies$ holds (Some x) xs nxt
 $\langle \text{proof} \rangle$

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)*

$\langle \text{proof} \rangle$

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

lemma *takeW-append*:

takeW pre (xs @ ys) nxt =
(let y = case ys of [] $\Rightarrow$ nxt | x # - $\Rightarrow$ Some x in
(let new-pre = case xs of [] $\Rightarrow$ pre | - $\Rightarrow$ (Some (last xs)) in
if holds pre xs y then xs @ takeW new-pre ys nxt
else takeW pre xs y))

$\langle \text{proof} \rangle$

lemma *holds-append*:

holds pre (xs @ ys) nxt =
(let y = case ys of [] $\Rightarrow$ nxt | x # - $\Rightarrow$ Some x in
(let new-pre = case xs of [] $\Rightarrow$ pre | - $\Rightarrow$ (Some (last xs)) in
holds pre xs y $\wedge$ holds new-pre ys nxt))

$\langle \text{proof} \rangle$

corollary *holds-cutoff*:

holds pre (l1 @ l2) nxt $\implies$ $\exists$ nxt' . holds pre l1 nxt'
 $\langle \text{proof} \rangle$

lemma *extract-append*:

extract pre (xs @ ys) nxt =
(let y = case ys of [] $\Rightarrow$ nxt | x # - $\Rightarrow$ Some x in
(let new-pre = case xs of [] $\Rightarrow$ pre | - $\Rightarrow$ (Some (last xs)) in
if holds pre xs y then extract new-pre ys nxt else extract pre xs y))

$\langle \text{proof} \rangle$

lemma *takeW-prefix*:

prefix (takeW pre l nxt) l
 $\langle \text{proof} \rangle$

lemma *takeW-set*: *t $\in$ set (TW.takeW P pre l nxt) $\implies$ t $\in$ set l*

$\langle \text{proof} \rangle$

lemma *holds-implies-takeW-is-identity*:

holds pre l nxt $\implies$ takeW pre l nxt = l
 $\langle \text{proof} \rangle$

lemma *holds-takeW-is-identity[simp]*:
 $takeW\ pre\ l\ next = l \longleftrightarrow holds\ pre\ l\ next$
 $\langle proof \rangle$

lemma *takeW-takeW-extract*:
 $takeW\ pre\ (takeW\ pre\ l\ next)\ (extract\ pre\ l\ next)$
 $= takeW\ pre\ l\ next$
 $\langle proof \rangle$

Show the equivalence of two takeW with different pres

lemma *takeW-pre-eqI*:
 $\llbracket \bigwedge x . l = [x] \implies P\ pre\ x\ next \longleftrightarrow P\ pre'\ x\ next;$
 $\bigwedge x1\ x2\ l' . l = x1 \# x2 \# l' \implies P\ pre\ x1\ (Some\ x2) \longleftrightarrow P\ pre'\ x1\ (Some\ x2) \rrbracket \implies$
 $takeW\ pre\ l\ next = takeW\ pre'\ l\ next$
 $\langle proof \rangle$

lemma *takeW-replace-pre*:
 $\llbracket P\ pre\ x1\ n; n = ifhead\ xs\ next \rrbracket \implies prefix\ (TW.takeW\ P\ pre'\ (x1 \# xs)\ next)\ (TW.takeW\ P\ pre\ (x1 \# xs)\ next)$
 $\langle proof \rangle$

Holds unfolding

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

lemma *holds-set-list*: $\llbracket holds\ pre\ l\ next; x \in set\ l \rrbracket \implies \exists\ p\ y . P\ p\ x\ y$
 $\langle proof \rangle$

lemma *holds-unfold*: $holds\ pre\ l\ None \implies$
 $l = [] \vee$
 $(\exists\ x . l = [x] \wedge P\ pre\ x\ None) \vee$
 $(\exists\ x\ y\ ys . l = (x \# y \# ys) \wedge P\ pre\ x\ (Some\ y) \wedge holds\ (Some\ x)\ (y \# ys)\ None)$
 $\langle proof \rangle$

lemma *holds-unfold-prexnext*:
 $\llbracket suffix\ (x0 \# x1 \# x2 \# xs)\ l; holds\ pre\ l\ next \rrbracket$
 $\implies P\ (Some\ x0)\ x1\ (Some\ x2)$
 $\langle proof \rangle$

lemma *holds-unfold-prexnext'*:
 $\llbracket holds\ pre\ l\ next; l = (zs @ (x0 \# x1 \# x2 \# xs)) \rrbracket$
 $\implies P\ (Some\ x0)\ x1\ (Some\ x2)$
 $\langle proof \rangle$

lemma *holds-unfold-xz*:
 $\llbracket suffix\ (x1 \# x2 \# xs)\ l; holds\ pre\ l\ next \rrbracket \implies \exists\ pre' . P\ pre'\ x1\ (Some\ x2)$
 $\langle proof \rangle$

lemma *holds-unfold-prex*:
 $\llbracket suffix\ (x1 \# x2 \# xs)\ l; holds\ pre\ l\ next \rrbracket \implies \exists\ next' . P\ (Some\ x1)\ x2\ next'$
 $\langle proof \rangle$

lemma *holds-suffix*:
 $\llbracket \text{holds } pre \ l \ next; \text{suffix } l' \ l \rrbracket \implies \exists \ pre'. \text{holds } pre' \ l' \ next$
 $\langle proof \rangle$

lemma *holds-unfold-prelnil*:
 $\llbracket \text{holds } pre \ l \ next; \ l = (zs@(x0 \# x1 \# [])) \rrbracket$
 $\implies P \ (Some \ x0) \ x1 \ next$
 $\langle proof \rangle$

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  $\Rightarrow$  'a  $\Rightarrow$  'a option  $\Rightarrow$  bool)
  fixes upd :: ('b  $\Rightarrow$  'a  $\Rightarrow$  'b)
begin
```

1.7.1 Definitions

Apply *upds* on a sequence

```
abbreviation upds :: 'b  $\Rightarrow$  'a list  $\Rightarrow$  'b where
  upds  $\equiv$  foldl upd
```

```
fun upd-opt :: ('b  $\Rightarrow$  'a option  $\Rightarrow$  '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  $\Rightarrow$  'a list  $\Rightarrow$  'a option  $\Rightarrow$  bool
where
  holds info (x # y # ys) nxt  $\longleftrightarrow$  P info x (Some y)  $\wedge$  holds (upd info y) (y # ys) nxt
| holds info [x] nxt  $\longleftrightarrow$  P info x nxt
| holds info [] nxt  $\longleftrightarrow$  True
```

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

```
function takeW :: 'b  $\Rightarrow$  'a list  $\Rightarrow$  'a option  $\Rightarrow$  'a list where
  takeW - [] - = []
| P info x xo  $\Longrightarrow$  takeW info [x] xo = [x]
|  $\neg$  P info x xo  $\Longrightarrow$  takeW info [x] xo = []
| P info x (Some y)  $\Longrightarrow$  takeW info (x # y # xs) xo = x # takeW (upd info y) (y # xs) xo
|  $\neg$  P info x (Some y)  $\Longrightarrow$  takeW info (x # y # xs) xo = []
<proof>
termination
  <proof>
```

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

```
fun extract :: 'b  $\Rightarrow$  'a list  $\Rightarrow$  'a option  $\Rightarrow$  '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 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 [])
<proof>

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 [])
<proof>

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 []))
<proof>

lemma *extract-split-tail*:

extract info (x # xs) nxt =
(case xs of
[] $\Rightarrow$ (if P info x nxt then nxt else (Some x))
| (y # ys) $\Rightarrow$ (if P info x (Some y) then extract (upd info y) (y # ys) nxt else Some x))
<proof>

lemma *holds-split-tail*:

holds info (x # xs) nxt $\longleftrightarrow$
(case xs of
[] $\Rightarrow$ P info x nxt
| (y # ys) $\Rightarrow$ P info x (Some y) $\wedge$ holds (upd info y) (y # ys) nxt)
<proof>

lemma *holds-Cons-P*:

holds info (x # xs) nxt $\implies \exists y . P \text{ info } x y$
<proof>

lemma *holds-Cons-holds*:

holds info (x # xs) nxt $\implies$ holds (upd-opt info (head xs)) xs nxt
<proof>

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)*

<proof>

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))
 <proof>

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

<proof>

lemma *takeW-prefix*:

prefix (takeW info l nxt) l
 <proof>

lemma *takeW-set*: *t ∈ set (TWu.takeW P upd info l nxt) ==> t ∈ set l*

<proof>

lemma *holds-implies-takeW-is-identity*:

holds info l nxt ==> takeW info l nxt = l
 <proof>

lemma *holds-takeW-is-identity[simp]*:

takeW info l nxt = l <==> holds info l nxt
 <proof>

lemma *takeW-takeW-extract*:

takeW info (takeW info l nxt) (extract info l nxt)
= takeW info l nxt
 <proof>

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*

<proof>

lemma *holds-set-list-no-update*: *[[holds info l nxt; x ∈ set l; ^a b. upd a b = a]] ==> ∃ y . P info x y*

<proof>

lemma *holds-unfold*: *holds info l None ==>*

l = [] ∨
(∃ x . l = [x] ^ P info x None) ∨
(∃ x y ys . l = (x#y#ys) ^ P info x (Some y) ^ holds (upd info y) (y#ys) None)
 <proof>

lemma *holds-unfold-prexnxt'*:

$\llbracket \text{holds info } l \text{ next}; l = (zs @ (x1 \# x2 \# xs)); zs \neq [] \rrbracket$
 $\implies P (\text{upds info } ((tl \text{ } zs) @ [x1])) \ x1 \ (\text{Some } x2)$
 $\langle \text{proof} \rangle$

lemma *holds-suffix*:

$\llbracket \text{holds info } l \text{ next}; \text{suffix } l' \ l \rrbracket \implies \exists \text{ info}'. \text{ holds info}' \ l' \text{ next}$
 $\langle \text{proof} \rangle$

lemma *holds-unfold-prelnil*:

$\llbracket \text{holds info } l \text{ next}; l = (zs @ (x1 \# [])); zs \neq [] \rrbracket$
 $\implies P (\text{upds info } ((tl \text{ } zs) @ [x1])) \ x1 \text{ next}$
 $\langle \text{proof} \rangle$

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**
 $\text{upd-shifted uinfo } (x \# xs) \text{ next} = \text{upds uinfo } (xs @ [next])$
 $| \text{upd-shifted uinfo } [] \text{ next} = \text{uinfo}$

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

lemma *holds-intermediate*:

assumes *holds uinfo p next* $p = \text{pre} @ \text{hf} \# \text{post}$
shows *holds (upd-shifted uinfo pre hf) (hf # post) next*
 $\langle \text{proof} \rangle$

lemma *holds-intermediate-ex*:

assumes *holds uinfo hfs next* $\text{hf} \in \text{set hfs}$
shows $\exists \text{pre post} . \text{ holds } (\text{upd-shifted uinfo pre hf}) \ (\text{hf} \# \text{post}) \text{ next} \wedge \text{hfs} = \text{pre} @ \text{hf} \# \text{post}$
 $\langle \text{proof} \rangle$

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 × '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 $\equiv$ *TW.holds if-valid pre l nxt*

abbreviation *ifs-valid-prefix* **where**

ifs-valid-prefix pre l nxt $\equiv$ *TW.takeW if-valid pre l nxt*

abbreviation *ifs-valid-None* **where**

ifs-valid-None l $\equiv$ *ifs-valid None l None*

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

ifs-valid-None-prefix l $\equiv$ *ifs-valid-prefix None l None*

lemma *strip-ifs-valid-prefix*:

pfragment ainfo l auth-seg0 $\implies$ *pfragment ainfo (ifs-valid-prefix pre l nxt) auth-seg0*
<proof>

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* $\Rightarrow$ *ifs* $\Rightarrow$ *as option* $\times$ *ifs option* **where**

rev-link a1 i1 $\equiv$ (*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 × ifs × as × ifs) ⇒ ('aahi, 'ainfo) pkt0 set
  loc  :: as ⇒ ('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 ⇒ ('aahi, 'ainfo) dp0-state
  ⇒ as ⇒ ('aahi, 'ainfo) pkt0 ⇒ bool

```

where

```

  dp0-add-loc s s' asid pkt ≡ s' = s[|loc := (loc s)(asid := loc 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 dp0-add-chan :: ('aahi, 'ainfo) dp0-state ⇒ ('aahi, 'ainfo) dp0-state
  ⇒ as ⇒ ifs ⇒ ('aahi, 'ainfo) pkt0 ⇒ bool where
  dp0-add-chan s s' a1 i1 pkt ≡
    ∃ a2 i2 . rev-link a1 i1 = (Some a2, Some i2) ∧
    s' = s[|chan := (chan s)((a1, i1, a2, i2) := chan s (a1, i1, a2, i2) ∪ {pkt})|]

```

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

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

lemmas $dp0\text{-}msgs = dp0\text{-}add\text{-}loc\text{-}def\ dp0\text{-}add\text{-}chan\text{-}def\ dp0\text{-}in\text{-}chan\text{-}def$

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

$dp0\text{-}dispatch\text{-}int\ s\ m\ ainfo\ asid\ pas\ fut\ hist\ s' \equiv$

— guard: check that the future path is a fragment of an authorized segment. In reality, honest agents will always choose a path that is a prefix of an authorized segment, but for our models this difference is not significant.

$m = ()\ AInfo = ainfo, past = pas, future = fut, history = hist\ () \wedge$

$hist = [] \wedge$

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

— action: Update the state to include m

$dp0\text{-}add\text{-}loc\ s\ s'\ asid\ m$

definition

dp0-recv

where

$dp0\text{-}recv\ s\ m\ asid\ ainfo\ hf1\ downif\ pas\ fut\ hist\ s' \equiv$

— guard: there are at least two hop fields left, which means we can advance the packet by one hop.

$m = ()\ AInfo = ainfo, past = pas, future = hf1\ \# fut, history = hist\ () \wedge$

$dp0\text{-}in\text{-}chan\ s\ asid\ downif\ m \wedge$

$ASID\ hf1 = asid \wedge$

— action: Update state to include message

$dp0\text{-}add\text{-}loc\ s\ s'\ asid\ ()$

$AInfo = ainfo,$

$past = pas,$

$$\begin{array}{l}
future = hf1 \# fut, \\
history = hist \\
\end{array}
\rangle$$

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' ≡

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

$pfragment\ ainfo\ fut\ auth-seg0 \wedge$

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

2.2.2 Transition system

fun $dp0-trans$ **where**

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

definition $dp0-init :: ('aahi, 'ainfo)\ dp0-state$ **where**

$dp0-init \equiv () \mid chan = (\lambda-. \{\}), loc = (\lambda-. \{\}) \mid$

definition $dp0 :: (('aahi, 'ainfo)\ evt0, ('aahi, 'ainfo)\ dp0-state)\ ES$ **where**

$dp0 \equiv ()$
 $init = (=)\ dp0-init,$
 $trans = dp0-trans$
 $\mid$

lemmas $dp0-trans-defs = dp0-dispatch-int-def\ dp0-recv-def\ dp0-send-def\ dp0-deliver-def\ dp0-dispatch-ext-def$

lemmas $dp0-defs = dp0-def\ dp0-init-def\ dp0-trans-defs$

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

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

declare $soup-def$ $[simp]$

declare $if-split-asm$ $[split]$

lemma $dp0-add-chan-msgs:$

assumes $dp0-add-chan\ s\ s'\ asid\ upif\ m$ **and** $soup\ n\ s'$ **and** $n \neq m$

shows $soup\ n\ s$

$\langle proof \rangle$

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 $\Rightarrow$ bool **where**
auth-path m $\equiv$ pfragment (AInfo m) (rev (history m) @ future m) auth-seg0

definition *inv-auth* :: ('aahi, 'ainfo) dp0-state $\Rightarrow$ bool **where**
inv-auth s $\equiv \forall m . \text{soup } m \ s \longrightarrow \text{auth-path } m$

lemma *inv-authI*:
assumes $\bigwedge m . \text{soup } m \ s \implies \text{pfragment } (AInfo \ m) \ (\text{rev } (\text{history } m) \ @ \ \text{future } m) \ \text{auth-seg0}$
shows *inv-auth* s
 <proof>

lemma *inv-authD*:
assumes *inv-auth* s soup m s
shows pfragment (AInfo m) (rev (history m) @ future m) auth-seg0
 <proof>

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>

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>

lemma *Inv-inv-auth*: Inv dp0 *inv-auth*
 <proof>

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

lemma *tr0-satisfies-pathauthorization*: dp0 $\models_{ES}$ *TR-auth*
 <proof>

Easier to read

definition *inv-authorized* :: ('aahi, 'ainfo) dp0-state $\Rightarrow$ bool **where**
inv-authorized s $\equiv \forall m . \text{soup } m \ s \longrightarrow$
 $(\exists \text{timestamp } \text{auth-path} . (\text{timestamp}, \text{auth-path}) \in \text{auth-seg0} \wedge$
 $(\exists \text{pre } \text{post} . \text{auth-path} = \text{pre} \ @ \ (\text{rev } (\text{history } m)) \ @ \ \text{post}))$

lemma *inv-auth* s $\implies$ *inv-authorized* s
 <proof>

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* $\Rightarrow$ bool **where**
inv-detect *s* $\equiv \forall m . \text{soup } m \text{ } s \longrightarrow \text{prefix } (\text{history } m) (\text{past } m)$

lemma *inv-detectI*:
assumes $\bigwedge m \ x . \text{soup } m \text{ } s \Longrightarrow \text{prefix } (\text{history } m) (\text{past } m)$
shows *inv-detect* *s*
 $\langle \text{proof} \rangle$

lemma *inv-detectD*:
assumes *inv-detect* *s*
shows $\bigwedge m \ x . m \in (\text{loc } s) \ x \Longrightarrow \text{prefix } (\text{history } m) (\text{past } m)$
and $\bigwedge m \ x . m \in (\text{chan } s) \ x \Longrightarrow \text{prefix } (\text{history } m) (\text{past } m)$
 $\langle \text{proof} \rangle$

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'*
 $\langle \text{proof} \rangle$

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'*
 $\langle \text{proof} \rangle$

lemma *Inv-inv-detect*: *Inv dp0 inv-detect*
 $\langle \text{proof} \rangle$

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

lemma *tr0-satisfies-detectability*: *dp0* $\models_{ES}$ *TR-detect*
 $\langle \text{proof} \rangle$

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

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

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

definition

$dp1\text{-recv}$

where

$dp1\text{-recv } s \ m \ asid \ ainfo \ hf1 \ downif \ pas \ fut \ hist \ s' \equiv$
 $DownIF \ hf1 = Some \ downif$
 $\wedge dp0\text{-recv } s \ m \ asid \ ainfo \ hf1 \ downif \ pas \ fut \ hist \ s'$

2.3.2 Transition system

fun $dp1\text{-trans}$ **where**

$dp1\text{-trans } s \ (evt\text{-dispatch-int0 } asid \ m) \ s' \longleftrightarrow$
 $(\exists ainfo \ pas \ fut \ hist. \ dp1\text{-dispatch-int } s \ m \ ainfo \ asid \ pas \ fut \ hist \ s') \mid$
 $dp1\text{-trans } s \ (evt\text{-dispatch-ext0 } asid \ upif \ m) \ s' \longleftrightarrow$
 $(\exists ainfo \ pas \ fut \ hist. \ dp1\text{-dispatch-ext } s \ m \ asid \ ainfo \ upif \ pas \ fut \ hist \ s') \mid$
 $dp1\text{-trans } s \ (evt\text{-recv0 } asid \ downif \ m) \ s' \longleftrightarrow$
 $(\exists ainfo \ hf1 \ pas \ fut \ hist. \ dp1\text{-recv } s \ m \ asid \ ainfo \ hf1 \ downif \ pas \ fut \ hist \ s') \mid$
 $dp1\text{-trans } s \ e \ s' \longleftrightarrow dp0\text{-trans } s \ e \ s'$

definition $dp1\text{-init} :: ('aahi, 'ainfo) \ dp1\text{-state}$ **where**

$dp1\text{-init} \equiv () \ chan = (\lambda-. \ {}), \ loc = (\lambda-. \ {}))$

definition $dp1 :: (('aahi, 'ainfo) \ evt1, ('aahi, 'ainfo) \ dp1\text{-state}) \ ES$ **where**

$dp1 \equiv ()$
 $init = (=) \ dp1\text{-init},$
 $trans = dp1\text{-trans}$
 $()$

lemmas $dp1\text{-trans-defs} = dp0\text{-trans-defs} \ dp1\text{-dispatch-ext-def} \ dp1\text{-recv-def}$

lemmas $dp1\text{-defs} = dp1\text{-def} \ dp1\text{-dispatch-int-def} \ dp1\text{-init-def} \ dp1\text{-trans-defs}$

fun $pkt1\text{to}0chan :: as \Rightarrow ifs \Rightarrow ('aahi, 'ainfo) \ pkt1 \Rightarrow ('aahi, 'ainfo) \ pkt0$ **where**

$pkt1\text{to}0chan \ asid \ upif \ () \ AInfo = ainfo, \ past = pas, \ future = fut, \ history = hist \ () =$
 $() \ pkt0.AInfo = ainfo, \ past = pas, \ future = ifs\text{-valid-prefix } (prev\text{-hf } asid \ upif) \ fut \ None,$
 $history = hist)$

fun $pkt1\text{to}0loc :: ('aahi, 'ainfo) \ pkt1 \Rightarrow ('aahi, 'ainfo) \ pkt0$ **where**

$pkt1\text{to}0loc \ () \ AInfo = ainfo, \ past = pas, \ future = fut, \ history = hist \ () =$
 $() \ pkt0.AInfo = ainfo, \ past = pas, \ future = ifs\text{-valid-prefix } None \ fut \ None, \ history = hist)$

definition $R10 :: ('aahi, 'ainfo) \ dp1\text{-state} \Rightarrow ('aahi, 'ainfo) \ dp0\text{-state}$ **where**

$R10 \ s =$
 $() \ chan = \lambda(a1, i1, a2, i2). \ (pkt1\text{to}0chan \ a1 \ i1) \ '((chan \ s) \ (a1, i1, a2, i2)),$
 $loc = \lambda x. \ pkt1\text{to}0loc \ '((loc \ s) \ x))$

```

fun  $\pi_1$  :: ('aahi, 'ainfo) evt1  $\Rightarrow$  ('aahi, 'ainfo) evt0 where
   $\pi_1$  (evt-dispatch-int0 asid m) = evt-dispatch-int0 asid (pkt1to0loc m)
|  $\pi_1$  (evt-recv0 asid downif m) = evt-recv0 asid downif (pkt1to0loc m)
|  $\pi_1$  (evt-send0 asid upif m) = evt-send0 asid upif (pkt1to0loc m)
|  $\pi_1$  (evt-deliver0 asid m) = evt-deliver0 asid (pkt1to0loc m)
|  $\pi_1$  (evt-dispatch-ext0 asid upif m) = evt-dispatch-ext0 asid upif (pkt1to0chan asid upif m)
|  $\pi_1$  (evt-observe0 s) = evt-observe0 (R10 s)
|  $\pi_1$  evt-skip0 = evt-skip0

```

```

declare TW.takeW.elims[elim]

```

```

lemma dp1-refines-dp0: dp1  $\sqsubseteq_{\pi_1}$  dp0
<proof>

```

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  $\Rightarrow$  msgterm  $\Rightarrow$  bool where
  ASIF (Some a) (AS a') = (a=a')
| ASIF None  $\varepsilon$  = True
| ASIF - = False

```

```

lemma ASIF-None[simp]: ASIF ifopt  $\varepsilon \longleftrightarrow$  ifopt = None <proof>

```

```

lemma ASIF-AS[simp]: ASIF ifopt (AS a)  $\longleftrightarrow$  ifopt = Some a <proof>

```

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

```

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

```

```

lemma ASIF-term2if[intro]: ASIF i mi  $\Longrightarrow$  ASIF (term2if mi) mi
<proof>

```

```

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

```

```

lemma if2term-eq[elim]: if2term a = if2term b  $\Longrightarrow$  a = b
<proof>

```

```

lemma term2if-if2term[simp]: term2if (if2term a) = a <proof>

```

```

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

```

```

fun term2hf :: msgterm  $\Rightarrow$  ahi where

```

$term2hf\ (L\ [upif,\ downif,\ Num\ asid]) = (\UpIF = term2if\ upif,\ DownIF = term2if\ downif,\ ASID = asid)$

lemma $term2hf$ - $hf2term$ [simp]: $term2hf\ (hf2term\ hf) = hf\ \langle proof \rangle$

lemma ahi -eq:

$\llbracket ASID\ ahi' = ASID\ (ahi::ahi);\ ASIF\ (DownIF\ ahi')\ downif;\ ASIF\ (UpIF\ ahi')\ upif;\ ASIF\ (DownIF\ ahi)\ downif;\ ASIF\ (UpIF\ ahi)\ upif \rrbracket \implies ahi = ahi'$
 $\langle proof \rangle$

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  $\Rightarrow$  'uinfo where
  upd-uinfo-pkt ( $\lambda$  AInfo = ainfo, UInfo = uinfo, past = pas, future = hf1 # fut, history = hist)  $\Rightarrow$ 
    = upd-uinfo uinfo hf1
| upd-uinfo-pkt ( $\lambda$  AInfo = ainfo, UInfo = uinfo, past = pas, future = [], history = hist)  $\Rightarrow$  uinfo
```

```
definition upd-pkt :: ('aahi, 'uinfo, 'uhi, 'ainfo) pkt2  $\Rightarrow$  ('aahi, 'uinfo, 'uhi, 'ainfo) pkt2 where
  upd-pkt pkt = pkt ( $\lambda$  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  $\Rightarrow$  'aahi ahi-scheme list where
  AHIS hfs  $\equiv$  map AHI hfs
```

```
declare AHIS-def[simp]
```

```
fun extr-from-hd :: ('aahi, 'uhi) HF list  $\Rightarrow$  '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  $\implies$  prefix (AHIS x1) (AHIS x2)
<proof>
```

```
lemma AHIS-set: hf  $\in$  set (AHIS l)  $\implies$   $\exists$  hfc . hfc  $\in$  set l  $\wedge$  hf = AHI hfc
<proof>
```

```
lemma AHIS-set-rev: ( $\lambda$  AHI = ahi, UHI = uhi, HVF = x)  $\in$  set hfs  $\implies$  ahi  $\in$  set (AHIS hfs)
<proof>
```

```
fun pkt2to1loc :: ('aahi, 'uinfo, 'uhi, 'ainfo) pkt2  $\Rightarrow$  ('aahi, 'ainfo) pkt1 where
  pkt2to1loc ( $\lambda$  AInfo = ainfo, UInfo = uinfo, past = pas, future = fut, history = hist)  $\Rightarrow$ 
    ( $\lambda$  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  $\Rightarrow$  ('aahi, 'ainfo) pkt1 where
  pkt2to1chan ( $\lambda$  AInfo = ainfo, UInfo = uinfo, past = pas, future = fut, history = hist)  $\Rightarrow$ 
    ( $\lambda$  pkt0.AInfo = ainfo,
      past = AHIS pas,
      future = AHIS (hfs-valid-prefix-generic ainfo
        (upd-uinfo-pkt ( $\lambda$  AInfo = ainfo, UInfo = uinfo, past = pas, future = fut, history = hist)  $\Rightarrow$ 
          pas (head pas) fut None),
      history = hist)
```

abbreviation $AHIo :: ('aahi, 'uhi) HF\ option \Rightarrow 'aahi\ ahi\ scheme\ option$ **where**
 $AHIo \equiv map-option\ AHI$

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 $auth-seg2 :: 'uinfo \Rightarrow ('ainfo \times ('aahi, 'uhi) HF\ list) set$ **where**
 $auth-seg2\ uinfo \equiv (\{(ainfo, l) \mid ainfo\ l.\ hfs-valid-prefix-generic\ ainfo\ uinfo \ \square\ None\ l\ None = l$
 $\quad \wedge\ auth-restrict\ ainfo\ uinfo\ l$
 $\quad \wedge\ no-oracle\ ainfo\ uinfo$
 $\quad \wedge\ (ainfo, AHIS\ l) \in auth-seg0\})$

lemma $auth-seg20$:

$$(x, y) \in auth-seg2\ uinfo \implies (x, AHIS\ y) \in auth-seg0\ \langle proof \rangle$$

lemma $pfragment-auth-seg20$:

$$pfragment\ ainfo\ l\ (auth-seg2\ uinfo) \implies pfragment\ ainfo\ (AHIS\ l)\ auth-seg0\ \langle proof \rangle$$

lemma $pfragment-auth-seg20'$:

$$\llbracket pfragment\ ainfo\ l\ (auth-seg2\ uinfo); l' = AHIS\ l \rrbracket \implies pfragment\ ainfo\ l'\ auth-seg0\ \langle proof \rangle$$

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

definition

$dp2-add-loc2 ::$
 $('aahi, 'uinfo, 'uhi, 'ainfo, 'more) dp2-state-scheme \Rightarrow$
 $('aahi, 'uinfo, 'uhi, 'ainfo, 'more) dp2-state-scheme \Rightarrow as \Rightarrow ('aahi, 'uinfo, 'uhi, 'ainfo) pkt2 \Rightarrow$
 $bool$

where

$$dp2-add-loc2\ s\ s'\ asid\ pkt \equiv s' = s \llbracket loc2 := (loc2\ s)(asid := loc2\ s\ asid \cup \{pkt\}) \rrbracket$$

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

definition

$dp2-add-chan2 ::$
 $('aahi, 'uinfo, 'uhi, 'ainfo, 'more) dp2-state-scheme \Rightarrow ('aahi, 'uinfo, 'uhi, 'ainfo, 'more) dp2-state-scheme$
 $\Rightarrow as \Rightarrow ifs \Rightarrow ('aahi, 'uinfo, 'uhi, 'ainfo) pkt2 \Rightarrow bool$

where

$$dp2-add-chan2\ s\ s'\ a1\ i1\ pkt \equiv$$

$$\exists a2\ i2.\ rev-link\ a1\ i1 = (Some\ a2, Some\ i2) \wedge$$

$$s' = s \llbracket chan2 := (chan2\ s)((a1, i1, a2, i2) := chan2\ s\ (a1, i1, a2, i2) \cup \{pkt\}) \rrbracket$$

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

definition

$dp2\text{-in-}chan2 :: ('aahi, 'uinfo, 'uhi, 'ainfo, 'more) dp2\text{-state-scheme} \Rightarrow as \Rightarrow ifs \Rightarrow ('aahi, 'uinfo, 'uhi, 'ainfo) pkt2 \Rightarrow bool$

where

$dp2\text{-in-}chan2\ s\ a1\ i1\ pkt \equiv$
 $\exists a2\ i2 . rev\text{-}link\ a1\ i1 = (Some\ a2, Some\ i2) \wedge$
 $pkt \in (chan2\ s)(a2, i2, a1, i1)$

lemmas $dp2\text{-msgs} = dp2\text{-add-loc2-def}\ dp2\text{-add-}chan2\text{-def}\ dp2\text{-in-}chan2\text{-def}$

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* $\Rightarrow$ *'uinfo*

$\Rightarrow ('aahi, 'uhi)\ HF\ list$

$\Rightarrow ('aahi, 'uhi)\ HF\ option$

$\Rightarrow ('aahi, 'uhi)\ HF$

$\Rightarrow ('aahi, 'uhi)\ HF\ option \Rightarrow bool$

and *upd-uinfo* :: *'uinfo* $\Rightarrow ('aahi, 'uhi)\ HF \Rightarrow 'uinfo +$

— *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**

$ik\text{-}hfs = \{t \mid t\ hf\ hfs\ ainfo\ uinfo. t \in terms\text{-}hf\ hf \wedge hf \in set\ hfs \wedge (ainfo, hfs) \in (auth\text{-}seg2\ uinfo)\}$

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

definition *ik-uinfo* :: *msgterm set* **where**

$ik\text{-}uinfo = \{t \mid ainfo\ hfs\ uinfo\ t. t \in terms\text{-}uinfo\ uinfo \wedge (ainfo, hfs) \in (auth\text{-}seg2\ uinfo)\}$

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

definition *ik* :: *msgterm set* **where**

$ik = ik\text{-}hfs$

$\cup \{term\text{-}ainfo\ ainfo \mid ainfo\ hfs\ uinfo. (ainfo, hfs) \in (auth\text{-}seg2\ uinfo)\}$

$\cup ik\text{-}uinfo$

$\cup Key('macK'bad)$

$\cup ik\text{-}add$

$\cup ik\text{-}oracle$

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

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

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 $\Rightarrow$ msgterm set **where**
ik-dyn *s* $\equiv$ *ik* $\cup$ ($\bigcup \{\text{terms-pkt } m \mid m \text{ x. } m \in \text{loc2 } s \text{ x}\} \cup (\bigcup \{\text{terms-pkt } m \mid m \text{ x. } m \in \text{chan2 } s \text{ x}\})$)

Different way of presenting the intruder knowledge

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

lemma *ik-dynamic* *s* = *ik-dyn* *s*
 $\langle \text{proof} \rangle$

lemma *ik-dyn-mono*: $\llbracket x \in \text{ik-dyn } s; \bigwedge m . \text{soup2 } m \text{ s} \implies \text{soup2 } m \text{ s}' \rrbracket \implies x \in \text{ik-dyn } s'$
 $\langle \text{proof} \rangle$

lemma *ik-info[elim]*:
 $(\text{ainfo}, \text{hfs}) \in (\text{auth-seg2 } \text{uinfo}) \implies \text{term-ainfo ainfo} \in \text{synth (analz ik)}$
 $\langle \text{proof} \rangle$

lemma *ik-ik-hfs*: $t \in \text{ik-hfs} \implies t \in \text{ik}$ $\langle \text{proof} \rangle$

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

definition

dp2-recv

where

dp2-recv *s m asid ainfo uinfo hf1 downif pas fut hist s'* $\equiv$

— guard: a packet with valid interfaces and valid validation fields is in the incoming channel.

$$m = () \text{ AInfo} = \text{ainfo}, \text{UInfo} = \text{uinfo}, \text{past} = \text{pas}, \text{future} = \text{hf1} \# \text{fut}, \text{history} = \text{hist} \rangle \wedge$$

$$\text{dp2-in-chan2 } s \text{ (ASID (AHI hf1)) downif } m \wedge$$

$$\text{DownIF (AHI hf1) = Some downif} \wedge$$

$$\text{ASID (AHI hf1) = asid} \wedge$$

$$\text{hf-valid-generic ainfo (upd-uinfo uinfo hf1) (rev(pas)@hf1\#fut) (head pas) hf1 (head fut) \wedge}$$

— action: Update local state to include message

$$\text{dp2-add-loc2 } s \text{ s' asid (upd-pkt } m)$$

definition

dp2-send

where

$$\text{dp2-send } s \text{ m asid ainfo uinfo hf1 upif pas fut hist s' } \equiv$$

— guard: forward the packet on the external channel and advance the path by one hop.

$$m = () \text{ AInfo} = \text{ainfo}, \text{UInfo} = \text{uinfo}, \text{past} = \text{pas}, \text{future} = \text{hf1} \# \text{fut}, \text{history} = \text{hist} \rangle \wedge$$

$$m \in (\text{loc2 } s) \text{ asid} \wedge$$

$$\text{UpIF (AHI hf1) = Some upif} \wedge$$

$$\text{ASID (AHI hf1) = asid} \wedge$$

$$\text{hf-valid-generic ainfo uinfo (rev(pas)@hf1\#fut) (head pas) hf1 (head fut) \wedge}$$

— action: Update state to include modified message

$$\text{dp2-add-chan2 } s \text{ s' asid upif } ()$$

$$\text{AInfo} = \text{ainfo},$$

$$\text{UInfo} = \text{uinfo},$$

$$\text{past} = \text{hf1} \# \text{pas},$$

$$\text{future} = \text{fut},$$

$$\text{history} = \text{AHI hf1} \# \text{hist}$$

$$\rangle$$

definition

dp2-deliver

where

$$\text{dp2-deliver } s \text{ m asid ainfo uinfo hf1 pas fut hist s' } \equiv$$

$$m = () \text{ AInfo} = \text{ainfo}, \text{UInfo} = \text{uinfo}, \text{past} = \text{pas}, \text{future} = \text{hf1} \# \text{fut}, \text{history} = \text{hist} \rangle \wedge$$

$$m \in (\text{loc2 } s) \text{ asid} \wedge$$

$$\text{ASID (AHI hf1) = asid} \wedge$$

$$\text{fut} = [] \wedge$$

$$\text{hf-valid-generic ainfo uinfo (rev(pas)@hf1\#fut) (head pas) hf1 (head fut) \wedge}$$

— action: Update state to include modified message

$$\text{dp2-add-loc2 } s \text{ s' asid}$$

$$()$$

$$\text{AInfo} = \text{ainfo},$$

$$\text{UInfo} = \text{uinfo},$$

$$\text{past} = \text{hf1} \# \text{pas},$$

$$\text{future} = [],$$

$$\text{history} = (\text{AHI hf1}) \# \text{hist}$$

$$\rangle$$

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 $hist$ $s' \equiv$
 $m = () \ AInfo = ainfo, UInfo = uinfo, past = pas, future = fut, history = hist \rangle \wedge$
 $asid \in bad \wedge$
 $hist = [] \wedge$
 $terms-pkt\ m \subseteq synth\ (analz\ (ik-dyn\ s)) \wedge$
 $no-oracle\ ainfo\ uinfo \wedge$
 — action
dp2-add-chan2 $s\ s'\ asid\ upif\ m$

2.4.4 Transition system

fun *dp2-trans* **where**

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

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

dp2-init $\equiv ()\ chan2 = (\lambda-. \{\}), loc2 = (\lambda-. \{\})()$

definition *dp2* :: $(('aahi, 'uinfo, 'uhi, 'ainfo)\ evt2, ('aahi, 'uinfo, 'uhi, 'ainfo)\ dp2-state)\ ES$ **where**

dp2 $\equiv ()$
 $init = (=)\ dp2-init,$
 $trans = dp2-trans$
 $\rangle$

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* :: $'ainfo \Rightarrow 'uinfo$
 $\Rightarrow ('aahi, 'uhi)\ HF\ list$
 $\Rightarrow ('aahi, 'uhi)\ HF\ option$

$\Rightarrow ('aahi, 'uhi) HF$
 $\Rightarrow ('aahi, 'uhi) HF \text{ option} \Rightarrow \text{bool}$
and $\text{upd-ainfo} :: 'uinfo \Rightarrow ('aahi, 'uhi) HF \Rightarrow 'uinfo +$

assumes *ik-seg-is-auth*:

$\llbracket \text{terms-pkt } m \subseteq \text{synth } (\text{analz } ik);$
 $\text{future } m = \text{hfs}; AInfo\ m = \text{ainfo};$
 $\text{nxt} = \text{None}; \text{no-oracle ainfo uinfo} \rrbracket$
 $\Rightarrow \text{pfragment ainfo}$
 $(\text{ifs-valid-prefix prev'}$
 $(AHIS (\text{hfs-valid-prefix-generic ainfo uinfo pas pre hfs nxt}))$
 $\text{None})$
 auth-seg0

and *upd-ainfo-ik*:

$\llbracket \text{terms-uinfo uinfo} \subseteq \text{synth } (\text{analz } ik); \text{terms-hf hf} \subseteq \text{synth } (\text{analz } ik) \rrbracket$
 $\Rightarrow \text{terms-uinfo } (\text{upd-ainfo uinfo hf}) \subseteq \text{synth } (\text{analz } ik)$

and *upd-ainfo-no-oracle*: $\text{no-oracle ainfo uinfo} \Rightarrow \text{no-oracle ainfo } (\text{upd-ainfo uinfo fld})$

— 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*:

$\text{prefix } (\text{hfs-valid-prefix-generic ainfo uinfo pas pre fut nxt}) \text{ fut}$

and *cons-hfs-valid-prefix-generic*:

$\llbracket \text{hf-valid-generic ainfo uinfo hfs } (\text{head pas}) \text{ hf1 } (\text{head fut}); \text{hfs} = (\text{rev pas}) @ \text{hf1} \# \text{fut};$
 $m = (AInfo = \text{ainfo}, UInfo = \text{uinfo}, \text{past} = \text{pas}, \text{future} = \text{hf1} \# \text{fut}, \text{history} = \text{hist}) \rrbracket$
 $\Rightarrow \text{hfs-valid-prefix-generic ainfo uinfo pas } (\text{head pas}) (\text{hf1} \# \text{fut}) \text{ None} =$
 $\text{hf1} \# (\text{hfs-valid-prefix-generic ainfo } (\text{upd-ainfo-pkt } (\text{fwd-pkt } m)) (\text{hf1} \# \text{pas}) (\text{Some hf1}) \text{ fut None})$
begin

2.4.6 Mapping dp2 state to dp1 state

definition $R21 :: ('aahi, 'uinfo, 'uhi, 'ainfo) \text{ dp2-state} \Rightarrow ('aahi, 'ainfo) \text{ dp1-state}$ **where**

$R21\ s = (\llbracket \text{chan} = \lambda x . \text{pkt2to1chan } ' ((\text{chan2 } s) \ x),$
 $\text{loc} = \lambda x . \text{pkt2to1loc } ' ((\text{loc2 } s) \ x) \rrbracket)$

lemma *auth-seg2-pfragment*:

$\llbracket \text{pfragment ainfo } (\text{hf} \# \text{fut}) (\text{auth-seg2 uinfo}); AHIS (\text{hf} \# \text{fut}) = x \# xs \rrbracket$
 $\Rightarrow \text{pfragment ainfo } (x \# xs) \text{ auth-seg0}$
 $\langle \text{proof} \rangle$

lemma *dp2-in-chan2-to-0E[elim]*:

$\llbracket \text{dp2-in-chan2 } s1\ a1\ i1\ \text{pkt2}; \text{pkt2to1chan } \text{pkt2} = \text{pkt0}; s0 = R21\ s1 \rrbracket \Rightarrow$
 $\text{dp0-in-chan } s0\ a1\ i1\ \text{pkt0}$
 $\langle \text{proof} \rangle$

lemma *dp2-in-loc2-to-0E[elim]*:

$\llbracket \text{pkt2} \in (\text{loc2 } s1) \text{ asid}; \text{pkt2to1loc } \text{pkt2} = \text{pkt0}; P = \text{pkt2to1loc } ' \text{ loc2 } s1 \text{ asid} \rrbracket \Rightarrow$
 $\text{pkt0} \in P$
 $\langle \text{proof} \rangle$

lemma *dp2-add-loc20E*:

$\llbracket \text{dp2-add-loc2 } s1\ s1' \text{ asid } p1; p0 = \text{pkt2to1loc } p1; s0 = R21\ s1; s0' = R21\ s1' \rrbracket$

$\implies dp0\text{-add-loc } s0 \ s0' \ asid \ p0$
 $\langle proof \rangle$

lemma *dp2-add-chan20E*:

$\llbracket dp2\text{-add-chan2 } s1 \ s1' \ a1 \ i1 \ p1; p0 = pkt2to1chan \ p1; s0 = R21 \ s1; s0' = R21 \ s1 \rrbracket$
 $\implies dp0\text{-add-chan } s0 \ s0' \ a1 \ i1 \ p0$
 $\langle proof \rangle$

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* $\Rightarrow$ *bool* **where**
inv-ik-dyn *s* $\equiv$ *ik-dyn* *s* $\subseteq$ *synth* (*analz* *ik*)

lemma *inv-ik-dynI*:

assumes $\bigwedge t \ m \ x . \llbracket t \in terms\text{-pkt } m; m \in loc2 \ s \ x \rrbracket \implies t \in synth \ (analz \ ik)$
and $\bigwedge t \ m \ x . \llbracket t \in terms\text{-pkt } m; m \in chan2 \ s \ x \rrbracket \implies t \in synth \ (analz \ ik)$
shows *inv-ik-dyn* *s*
 $\langle proof \rangle$

lemma *inv-ik-dynD*:

assumes *inv-ik-dyn* *s*
shows $\bigwedge t \ m \ x . \llbracket m \in chan2 \ s \ x; t \in terms\text{-pkt } m \rrbracket \implies t \in synth \ (analz \ ik)$
 $\bigwedge t \ m \ x . \llbracket m \in loc2 \ s \ x; t \in terms\text{-pkt } m \rrbracket \implies t \in synth \ (analz \ ik)$
 $\langle proof \rangle$

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

lemma *inv-ik-dyn-add-loc2[elim!]*:

$\llbracket dp2\text{-add-loc2 } s \ s' \ asid \ m; inv\text{-ik-dyn } s; terms\text{-pkt } m \subseteq synth \ (analz \ ik) \rrbracket$
 $\implies inv\text{-ik-dyn } s'$
 $\langle proof \rangle$

lemma *inv-ik-dyn-add-chan2[elim!]*:

$\llbracket dp2\text{-add-chan2 } s \ s' \ a1 \ i1 \ m; inv\text{-ik-dyn } s; terms\text{-pkt } m \subseteq synth \ (analz \ ik) \rrbracket$
 $\implies inv\text{-ik-dyn } s'$
 $\langle proof \rangle$

lemma *inv-ik-dyn-ik-dyn-ik[simp]*:

assumes *inv-ik-dyn* *s* **shows** *synth* (*analz* (*ik-dyn* *s*)) = *synth* (*analz* *ik*)
 $\langle proof \rangle$

lemma *terms-pkt-upd*:

$\llbracket x \in terms\text{-pkt } (upd\text{-pkt } p); \bigwedge x . x \in terms\text{-pkt } p \implies x \in synth \ (analz \ ik) \rrbracket \implies x \in synth \ (analz \ ik)$
 $\langle proof \rangle$

lemma *Inv-inv-ik-dyn: reach dp2 s* $\implies inv\text{-ik-dyn } s$

$\langle proof \rangle$

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 ⊆ synth (analz (ik-dyn s))*

<proof>

2.4.8 Refinement proof

fun $\pi_2 :: ('aahi, 'uinfo, 'uhi, 'ainfo) \text{ evt2} \Rightarrow ('aahi, 'ainfo) \text{ evt0}$ **where**

$\pi_2 (\text{evt-dispatch-int2 asid } m) = \text{evt-dispatch-int0 asid } (\text{pkt2to1loc } m)$
 $\pi_2 (\text{evt-recv2 asid downif } m) = \text{evt-recv0 asid downif } (\text{pkt2to1chan } m)$
 $\pi_2 (\text{evt-send2 asid upif } m) = \text{evt-send0 asid upif } (\text{pkt2to1loc } m)$
 $\pi_2 (\text{evt-deliver2 asid } m) = \text{evt-deliver0 asid } (\text{pkt2to1loc } m)$
 $\pi_2 (\text{evt-dispatch-ext2 asid upif } m) = \text{evt-dispatch-ext0 asid upif } (\text{pkt2to1chan } m)$
 $\pi_2 (\text{evt-observe2 } s) = \text{evt-observe0 } (R21 \ s)$
 $\pi_2 \text{ evt-skip2} = \text{evt-skip0}$

lemma *dp2-refines-dp1*: $\text{dp2} \sqsubseteq_{\pi_2} \text{dp1}$

<proof>

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* :: $('aahi, 'uinfo, 'uhi, 'ainfo) \text{ pkt2} \Rightarrow \text{bool}$ **where**

auth-path2 m ≡ pfragment (AInfo m) (rev (history m)) auth-seg0

abbreviation *TR-auth2-hist* :: $('aahi, 'uinfo, 'uhi, 'ainfo) \text{ evt2 list set}$ **where** *TR-auth2-hist* ≡
 $\{\tau \mid \tau . \forall s \ m . \text{evt-observe2 } s \in \text{set } \tau \wedge \text{soup2 } m \ s \longrightarrow \text{auth-path2 } m\}$

lemma *evt-observe2-0*:

evt-observe2 s ∈ set τ ⇒ evt-observe0 (R10 (R21 s)) ∈ (λx. π₁ (π₂ x)) ‘ set τ

<proof>

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

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

lemma *loc2to0*: $\llbracket mc \in \text{loc2 } sc \ x; sa = R10 \ (R21 \ sc); ma = \text{pkt1to0loc } (\text{pkt2to1loc } mc) \rrbracket \Longrightarrow ma \in \text{loc } sa \ x$

<proof>

lemma *chan2to0*: $\llbracket mc \in \text{chan2 } sc \ (a1, i1, a2, i2); sa = R10 \ (R21 \ sc); ma = \text{pkt1to0chan } a1 \ i1 \ (\text{pkt2to1chan } mc) \rrbracket$
 $\implies ma \in \text{chan } sa \ (a1, i1, a2, i2)$
 $\langle \text{proof} \rangle$

lemma *loc2to0-auth*:
 $\llbracket mc \in \text{loc2 } sc \ x; sa = R10 \ (R21 \ sc); ma = \text{pkt1to0loc } (\text{pkt2to1loc } mc); \text{auth-path } ma \rrbracket \implies \text{auth-path2 } mc$
 $\langle \text{proof} \rangle$

lemma *chan2to0-auth*:
 $\llbracket mc \in \text{chan2 } sc \ (a1, i1, a2, i2); sa = R10 \ (R21 \ sc); ma = \text{pkt1to0chan } a1 \ i1 \ (\text{pkt2to1chan } mc); \text{auth-path } ma \rrbracket \implies \text{auth-path2 } mc$
 $\langle \text{proof} \rangle$

lemma *tr2-satisfies-pathauthorization*: $dp2 \models_{ES} TR\text{-auth2-hist}$
 $\langle \text{proof} \rangle$

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

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

lemma *tr2-satisfies-detectability*: $dp2 \models_{ES} TR\text{-detect2}$
 $\langle \text{proof} \rangle$

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: $(info, l) \in auth-seg0 \implies if_valid_None\ l$ **and**

— All authorized segments are rooted, i.e., they start with None

ASM-empty [*simp*, *intro!*]: $(info, []) \in auth-seg0$ **and**

ASM-rooted: $(info, l) \in auth-seg0 \implies rooted\ l$ **and**

ASM-terminated: $(info, l) \in auth-seg0 \implies terminated\ l$

locale *network-assums-undirect* = *network-assums-generic* - - +

assumes

ASM-adversary: $\llbracket \bigwedge hf. hf \in set\ hfs \implies ASID\ hf \in bad \rrbracket \implies (info, hfs) \in auth-seg0$

locale *network-assums-direct* = *network-assums-generic* - - +

assumes

ASM-singleton: $\llbracket ASID\ hf \in bad \rrbracket \implies (info, [hf]) \in auth-seg0$ **and**

ASM-extension: $\llbracket (info, hf2 \# ys) \in auth-seg0; ASID\ hf2 \in bad; ASID\ hf1 \in bad \rrbracket$

$\implies (info, hf1 \# hf2 \# ys) \in auth-seg0$ **and**

ASM-modify: $\llbracket (info, hf \# ys) \in auth-seg0; ASID\ hf = a; ASID\ hf' = a; UpIF\ hf' = UpIF\ hf; a \in bad \rrbracket$

$\implies (info, hf' \# ys) \in auth-seg0$ **and**

ASM-cutoff: $\llbracket (info, zs @ hf \# ys) \in auth-seg0; ASID\ hf = a; a \in bad \rrbracket \implies (info, hf \# ys) \in auth-seg0$

begin

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

<proof>

lemma *auth-seg0-non-empty-frag* [*simp*, *intro!*]: $\exists\ info. pfragment\ info \sqsubseteq auth-seg0$

<proof>

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*

<proof>

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

lemma $\llbracket \bigwedge hf. hf \in set\ hfs \implies ASID\ hf \in bad \rrbracket \implies (info, hfs) \in auth-seg0$

<proof>

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 $\Rightarrow$ msgterm set

— *upd-uinfo* returns the updated uinfo field of a packet.

and *upd-uinfo* :: 'uinfo $\Rightarrow$ ('aahi, 'uhi) HF $\Rightarrow$ '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 $\Rightarrow$ 'uinfo $\Rightarrow$ bool

begin

abbreviation *hf-valid-generic* :: 'ainfo $\Rightarrow$ 'uinfo

$\Rightarrow$ ('aahi, 'uhi) HF list

$\Rightarrow$ ('aahi, 'uhi) HF option

$\Rightarrow$ ('aahi, 'uhi) HF

$\Rightarrow$ ('aahi, 'uhi) HF option $\Rightarrow$ bool **where**

hf-valid-generic ainfo uinfo pas pre hf nxt $\equiv$ *hf-valid* ainfo uinfo hf nxt

definition *hfs-valid-prefix-generic* ::

'ainfo $\Rightarrow$ 'uinfo $\Rightarrow$ ('aahi, 'uhi) HF list $\Rightarrow$ ('aahi, 'uhi) HF option $\Rightarrow$ ('aahi, 'uhi) HF list $\Rightarrow$

('aahi, 'uhi) HF option $\Rightarrow$ ('aahi, 'uhi) HF list **where**

hfs-valid-prefix-generic ainfo uinfo pas pre fut nxt $\equiv$

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*

<proof>

abbreviation *hfs-valid* **where**

hfs-valid ainfo uinfo l nxt $\equiv$ *TWu.holds* (*hf-valid* ainfo) *upd-uinfo* uinfo l nxt

abbreviation *hfs-valid-prefix* **where**

hfs-valid-prefix ainfo uinfo l nxt $\equiv$ *TWu.takeW* (*hf-valid* ainfo) *upd-uinfo* uinfo l nxt

abbreviation *hfs-valid-None* **where**

hfs-valid-None ainfo uinfo l $\equiv$ *hfs-valid* ainfo uinfo l None

abbreviation *hfs-valid-None-prefix* **where**

hfs-valid-None-prefix ainfo uinfo l $\equiv$ *hfs-valid-prefix* ainfo uinfo l None

abbreviation *upds-uinfo* **where**

upds-uinfo $\equiv$ *foldl* *upd-uinfo*

abbreviation *upds-uinfo-shifted* **where**

upds-uinfo-shifted uinfo l nxt $\equiv$ *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  $\Rightarrow$  'uinfo  $\Rightarrow$  ('aahi, 'uhi) HF  $\Rightarrow$  ('aahi, 'uhi) HF option  $\Rightarrow$  bool
  and auth-restrict :: 'ainfo  $\Rightarrow$  'uinfo  $\Rightarrow$  ('aahi, 'uhi) HF list  $\Rightarrow$  bool
  and extr :: msgterm  $\Rightarrow$  'aahi ahi-scheme list
  and extr-ainfo :: msgterm  $\Rightarrow$  'ainfo
  and term-ainfo :: 'ainfo  $\Rightarrow$  msgterm
  and terms-hf :: ('aahi, 'uhi) HF  $\Rightarrow$  msgterm set
  and upd-uinfo :: 'uinfo  $\Rightarrow$  ('aahi, 'uhi) HF  $\Rightarrow$  '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:  $\llbracket (ainfo, hfs) \in (auth-seg2\ uinfo); hf \in set\ hfs \rrbracket$ 
 $\Rightarrow \exists\ nxt\ uinfo'. hf\ valid\ ainfo\ uinfo'\ hf\ nxt \wedge auth-restrict\ ainfo\ uinfo\ hfs \wedge (ainfo, AHIS\ hfs) \in$ 
auth-seg0
   $\langle proof \rangle$ 

lemma prefix-hfs-valid-prefix-generic:
  prefix (hfs-valid-prefix-generic ainfo uinfo pas pre fut nxt) fut
   $\langle proof \rangle$ 

lemma cons-hfs-valid-prefix-generic:
   $\llbracket hf\ valid\ generic\ ainfo\ uinfo\ hfs\ (head\ pas)\ hf1\ (head\ fut); hfs = (rev\ pas) @ hf1 \# fut;$ 
 $m = (\backslash AInfo = ainfo, UInfo = uinfo, past = pas, future = hf1 \# fut, history = hist) \rrbracket$ 
 $\Rightarrow hfs\ valid\ prefix\ generic\ ainfo\ uinfo\ pas\ (head\ pas)\ (hf1 \# fut)\ None =$ 
 $hf1 \# (hfs\ valid\ prefix\ generic\ ainfo\ (upd-uinfo-pkt\ (fwd-pkt\ m))\ (hf1 \# pas)\ (Some\ hf1)\ fut\ None)$ 
   $\langle proof \rangle$ 

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
   $\langle proof \rangle$ 
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  $\Rightarrow$  'uinfo
 $\Rightarrow$  ('aahi, 'uhi) HF
 $\Rightarrow$  ('aahi, 'uhi) HF option  $\Rightarrow$  bool
and auth-restrict :: 'ainfo  $\Rightarrow$  'uinfo  $\Rightarrow$  ('aahi, 'uhi) HF list  $\Rightarrow$  bool
and extr :: msgterm  $\Rightarrow$  'aahi ahi-scheme list
and extr-ainfo :: msgterm  $\Rightarrow$  'ainfo

```

and $\text{term-ainfo} :: 'ainfo \Rightarrow \text{msgterm}$
and $\text{upd-uinfo} :: 'uinfo \Rightarrow ('aahi, 'uhi) HF \Rightarrow 'uinfo$
and $\text{ik-add} :: \text{msgterm set}$
and $\text{ik-oracle} :: \text{msgterm set}$
and $\text{no-oracle} :: 'ainfo \Rightarrow 'uinfo \Rightarrow \text{bool} +$

— 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*:

$\llbracket \text{hf-valid ainfo uinfo hf nxt; terms-hf hf} \subseteq \text{analz ik; no-oracle ainfo uinfo} \rrbracket$
 $\implies \exists \text{hfs} . \text{hf} \in \text{set hfs} \wedge (\exists \text{uinfo}' . (\text{ainfo}, \text{hfs}) \in (\text{auth-seg2 uinfo}'))$

— 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*:

$\llbracket \text{ASID (AHI hf)} \notin \text{bad; hf-valid ainfo uinfo hf nxt; terms-hf hf} \subseteq \text{synth (analz ik); no-oracle ainfo uinfo} \rrbracket$
 $\implies \text{terms-hf hf} \subseteq \text{analz ik}$

— 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*:

$\text{prefix (AHIS (hfs-valid-prefix ainfo uinfo l nxt))}$
 (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 $\text{prefix (AHIS (hfs-valid-prefix ?ainfo ?uinfo ?l ?nxt)) (extr-from-hd ?l)}$, 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*:

$\llbracket \text{hfs-valid ainfo uinfo l nxt; auth-restrict ainfo uinfo l; nxt} = \text{None} \rrbracket$
 $\implies \text{prefix (extr-from-hd l) (AHIS l)}$

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

and *COND-hf-valid-uinfo*:

$\llbracket \text{hf-valid ainfo uinfo hf nxt; hf-valid ainfo' uinfo' hf nxt} \rrbracket$
 $\implies \text{uinfo}' = \text{uinfo}$

— Updating a uinfo field does not reveal anything novel to the attacker.

and *COND-upd-uinfo-ik*:

$\llbracket \text{terms-uinfo uinfo} \subseteq \text{synth (analz ik); terms-hf hf} \subseteq \text{synth (analz ik)} \rrbracket$
 $\implies \text{terms-uinfo (upd-uinfo uinfo hf)} \subseteq \text{synth (analz ik)}$

— The determination of whether a packet is an oracle packet is invariant under uinfo field updates.

and *COND-upd-uinfo-no-oracle*:

$\text{no-oracle ainfo uinfo} \implies \text{no-oracle ainfo (upd-uinfo uinfo fld)}$

— The restriction on authorized paths is invariant under uinfo field updates.

and *COND-auth-restrict-upd*:

$\text{auth-restrict ainfo uinfo (hf1 \# hf2 \# xs)} \implies \text{auth-restrict ainfo (upd-uinfo uinfo hf2) (hf2 \# xs)}$

begin

lemma *holds-path-eq-extr*:

$\llbracket \text{hfs-valid ainfo uinfo l nxt; auth-restrict ainfo uinfo l; nxt} = \text{None} \rrbracket \implies \text{extr-from-hd l} = \text{AHIS l}$
 $\langle \text{proof} \rangle$

lemma *upds-uinfo-no-oracle*:

no-oracle ainfo uinfo $\implies$ no-oracle ainfo (upds-uinfo uinfo hfs)
<proof>

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*:

$\llbracket t \in \text{terms-uinfo (upd-uinfo uinfo hf); terms-uinfo uinfo} \subseteq \text{synth (analz ik); terms-hf hf} \subseteq \text{synth (analz ik)} \rrbracket$
 $\implies t \in \text{synth (analz ik)}$
<proof>

lemma *honest-hf-analz-subsetI*:

$\llbracket t \in \text{terms-hf hf; ASID (AHI hf)} \notin \text{bad; hf-valid ainfo uinfo hf next; terms-hf hf} \subseteq \text{synth (analz ik)}; \text{no-oracle ainfo uinfo} \rrbracket$
 $\implies t \in \text{analz ik}$
<proof>

lemma *extr-from-hd-eq*: $(l \neq [] \wedge l' \neq [] \wedge \text{hd } l = \text{hd } l') \vee (l = [] \wedge l' = []) \implies \text{extr-from-hd } l = \text{extr-from-hd } l'$
<proof>

lemma *path-prefix-extr-l*:

$\llbracket \text{hd } l = \text{hd } l'; l' \neq [] \rrbracket \implies \text{prefix (AHIS (hfs-valid-prefix ainfo uinfo l next)) (extr-from-hd } l')$
<proof>

lemma *path-prefix-extr-l'*:

$\llbracket \text{hd } l = \text{hd } l'; l' \neq []; \text{hf} = \text{hd } l' \rrbracket \implies \text{prefix (AHIS (hfs-valid-prefix ainfo uinfo l next)) (extr (HVF hf))}$
<proof>

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)*
<proof>

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)*
<proof>

lemma *pfrag-extr-auth*:

assumes *hf $\in$ set p and (ainfo, p) $\in$ (auth-seg2 uinfo)*
shows *pfragment ainfo (extr (HVF hf)) auth-seg0*
<proof>

lemma *X-in-ik-is-auth*:

assumes $\text{terms-hf hf1} \subseteq \text{analz ik}$ **and** *no-oracle ainfo uinfo*
shows $\text{pfragment ainfo (AHIS (hfs-valid-prefix ainfo uinfo$
 $(\text{hf1} \# \text{fut})$
 $\text{next}))}$
 auth-seg0
 $\langle \text{proof} \rangle$

Fragment is extendable

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

fun $\text{terminated2} :: ('aahi, 'uhi) \text{HF list} \Rightarrow \text{bool}$ **where**
 $\text{terminated2 (hf\#xs)} \longleftrightarrow \text{DownIF (AHI hf)} = \text{None} \vee \text{ASID (AHI hf)} \in \text{bad}$
 $| \text{terminated2 []} = \text{True}$

lemma $\text{terminated20: terminated (AHIS m)} \Longrightarrow \text{terminated2 m}$ $\langle \text{proof} \rangle$

lemma $\text{cons-snoc: } \exists y \text{ ys. } x \# xs = \text{ys} @ [y]$
 $\langle \text{proof} \rangle$

lemma $\text{terminated2-suffix:}$

$\llbracket \text{terminated2 l; l = zs} @ x \# xs; \text{DownIF (AHI x)} \neq \text{None}; \text{ASID (AHI x)} \notin \text{bad} \rrbracket \Longrightarrow \exists y \text{ ys. } zs = \text{ys} @ [y]$
 $\langle \text{proof} \rangle$

lemma $\text{attacker-modify-cutoff: } \llbracket (\text{info, zs} @ \text{hf} \# \text{ys}) \in \text{auth-seg0}; \text{ASID hf} = a;$
 $\text{ASID hf}' = a; \text{UpIF hf}' = \text{UpIF hf}; a \in \text{bad}; \text{ys}' = \text{hf}' \# \text{ys} \rrbracket \Longrightarrow (\text{info, ys}') \in \text{auth-seg0}$
 $\langle \text{proof} \rangle$

lemma $\text{auth-seg2-terms-hf[elim]:}$

$\llbracket x \in \text{terms-hf hf}; \text{hf} \in \text{set hfs}; (\text{ainfo, hfs}) \in (\text{auth-seg2 uinfo}) \rrbracket \Longrightarrow x \in \text{analz ik}$
 $\langle \text{proof} \rangle$

lemma $\llbracket \text{hfs-valid ainfo uinfo hfs next; hfs} = \text{pref} @ [\text{hf}] \rrbracket \Longrightarrow \text{hf-valid ainfo (upds-uinfo uinfo pref) hf}$
 next
 $\langle \text{proof} \rangle$

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 $\text{fragment-with-Eo-Some-extendable:}$

assumes $\text{terms-hf hf2} \subseteq \text{synth (analz ik)}$

```

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⟩

```

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)
    next))
    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)
      next))
      None)
  auth-seg0
⟨proof⟩

```

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 next = None and no-oracle ainfo uinfo
  shows pfragment ainfo
    (ifs-valid-prefix prev'
      (AHIS (hfs-valid-prefix ainfo uinfo hfs next))
      None)
  auth-seg0
⟨proof⟩

```

```

lemma ik-seg-is-auth':
  assumes terms-pkt m  $\subseteq$  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
   $\langle$ proof $\rangle$ 

print-locale dataplane-2
sublocale dataplane-2 - - - - hfs-valid-prefix-generic - - - - - no-oracle - - hf-valid-generic upd-uinfo
   $\langle$ proof $\rangle$ 

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 $\Rightarrow$ 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 $\Rightarrow$ msgterm set

— *terms-uinfo* extracts what msgterms the intruder can learn from analyzing a given uinfo field.

and *terms-uinfo* :: UINFO $\Rightarrow$ 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 $\Rightarrow$ UINFO $\Rightarrow$ bool

begin

abbreviation *upd-uinfo* :: UINFO $\Rightarrow$ ('aahi, 'uhi) HF $\Rightarrow$ UINFO **where**

upd-uinfo u hf $\equiv$ u

abbreviation *hf-valid-generic* :: 'ainfo $\Rightarrow$ msgterm

$\Rightarrow$ ('aahi, 'uhi) HF list

$\Rightarrow$ ('aahi, 'uhi) HF option

$\Rightarrow$ ('aahi, 'uhi) HF

$\Rightarrow$ ('aahi, 'uhi) HF option $\Rightarrow$ bool **where**

hf-valid-generic ainfo uinfo hfs pre hf nxt $\equiv$ *hf-valid* ainfo uinfo hfs hf

abbreviation *hfs-valid-prefix* **where**

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

definition *hfs-valid-prefix-generic* ::

'ainfo $\Rightarrow$ msgterm $\Rightarrow$ ('aahi, 'uhi) HF list $\Rightarrow$ ('aahi, 'uhi) HF option $\Rightarrow$ ('aahi, 'uhi) HF list $\Rightarrow$

('aahi, 'uhi) HF option $\Rightarrow$ ('aahi, 'uhi) HF list **where**

hfs-valid-prefix-generic ainfo uinfo pas pre fut nxt $\equiv$

hfs-valid-prefix ainfo uinfo pas fut

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

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*

<proof>

lemma *auth-seg2-elem*: $\llbracket (ainfo, hfs) \in \text{auth-seg2 } uinfo; hf \in \text{set } hfs \rrbracket$

$\Rightarrow \exists uinfo . hf\text{-valid } ainfo uinfo hfs hf \wedge \text{auth-restrict } ainfo uinfo hfs \wedge (ainfo, AHIS hfs) \in \text{auth-seg0}$

<proof>

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 $\Rightarrow$ UINFO $\Rightarrow$ ('aahi, 'uhi) HF list $\Rightarrow$ ('aahi, 'uhi) HF $\Rightarrow$ 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
  <proof>

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)
  <proof>

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
  <proof>
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*:

$\llbracket hf\text{-valid ainfo uinfo } l \ hf; \text{ terms-hf } hf \subseteq \text{analz } ik; \text{ no-oracle ainfo uinfo}; hf \in \text{set } l \rrbracket$
 $\implies \exists hfs . hf \in \text{set } hfs \wedge (\exists uinfo' . (ainfo, hfs) \in (\text{auth-seg2 } uinfo'))$

— 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*:

$\llbracket ASID (AHI \ hf) \notin \text{bad}; hf\text{-valid ainfo uinfo } l \ hf; \text{ terms-hf } hf \subseteq \text{synth } (\text{analz } ik);$
 $\text{no-oracle ainfo uinfo}; hf \in \text{set } l \rrbracket$
 $\implies \text{terms-hf } hf \subseteq \text{analz } ik$

— Each valid hop field contains the entire path.

and *COND-extr*:

$\llbracket hf\text{-valid ainfo uinfo } l \ hf \rrbracket \implies \text{extr } (HVF \ hf) = AHIS \ l$

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

and *COND-hf-valid-uinfo*:

$\llbracket hf\text{-valid ainfo uinfo } l \ hf; hf\text{-valid ainfo' uinfo' } l' \ hf \rrbracket$
 $\implies uinfo' = uinfo$

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 $\text{terms-pkt } m \subseteq \text{synth } (\text{analz } ik)$ **and**

$\text{future } m = \text{fut}$ **and** $AInfo \ m = \text{ainfo}$ **and** $\text{nxt} = \text{None}$ **and** $\text{no-oracle ainfo uinfo}$

shows pfragment ainfo

$(AHIS (hfs\text{-valid-prefix ainfo uinfo pas fut}))$

auth-seg0

$\langle \text{proof} \rangle$

lemma *upd-uinfo-pkt-id[simp]*: $\text{upd-uinfo-pkt } pkt = UInfo \ pkt$

$\langle \text{proof} \rangle$

print-locale *dataplane-2*

sublocale *dataplane-2* - - - - *hfs-valid-prefix-generic* - - - - - *no-oracle* - - *hf-valid-generic* *upd-uinfo*

$\langle \text{proof} \rangle$

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  $\times$  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  $\Rightarrow$  msgterm
   $\Rightarrow$  SCION-HF
   $\Rightarrow$  SCION-HF option  $\Rightarrow$  bool where
    hf-valid (Num ts) uinfo ( $\llbracket$  AHI = ahi, UHI = -, HVF = x  $\rrbracket$  (Some ( $\llbracket$  AHI = ahi2, UHI = -, HVF =
    x2  $\rrbracket$ ))  $\longleftrightarrow$ 
      ( $\exists$  upif downif upif2 downif2.
        x = Mac[macKey (ASID ahi)] (L [Num ts, upif, downif, upif2, downif2, x2])  $\wedge$ 
        ASIF (DownIF ahi) downif  $\wedge$  ASIF (UpIF ahi) upif  $\wedge$ 
        ASIF (DownIF ahi2) downif2  $\wedge$  ASIF (UpIF ahi2) upif2  $\wedge$  uinfo =  $\varepsilon$ )
  | hf-valid (Num ts) uinfo ( $\llbracket$  AHI = ahi, UHI = -, HVF = x  $\rrbracket$  None  $\longleftrightarrow$ 
    ( $\exists$  upif downif. x = Mac[macKey (ASID ahi)] (L [Num ts, upif, downif])  $\wedge$ 
      ASIF (DownIF ahi) downif  $\wedge$  ASIF (UpIF ahi) upif  $\wedge$  uinfo =  $\varepsilon$ )
  | hf-valid - - - = False

```

definition upd-uinfo :: msgterm $\Rightarrow$ SCION-HF $\Rightarrow$ msgterm **where**
 upd-uinfo uinfo hf $\equiv$ 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  $\Rightarrow$  ahi list where
  extr (Mac[macKey asid] (L [ts, upif, downif, upif2, downif2, x2]))
  = ( $\llbracket$  UpIF = term2if upif, DownIF = term2if downif, ASID = asid  $\rrbracket$  # extr x2)
  | extr (Mac[macKey asid] (L [ts, upif, downif]))
  = ( $\llbracket$  UpIF = term2if upif, DownIF = term2if downif, ASID = asid  $\rrbracket$ )
  | extr - = []

```

Extract the authenticated info field from a hop validation field.

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

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

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

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

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* $\equiv (\exists ts. \text{ainfo} = \text{Num } ts) \wedge (\text{uinfo} = \varepsilon)$

abbreviation *no-oracle* **where** *no-oracle* $\equiv (\lambda - .. \text{True})$

We now define useful properties of the above definition.

lemma *hf-valid-invert*:

hf-valid *tsn* *uinfo* *hf* *mo* $\longleftrightarrow$
 $((\exists \text{ahi} \text{ahi2} \text{ts} \text{upif} \text{downif} \text{asid} \text{x} \text{upif2} \text{downif2} \text{x2}.$
 $\text{hf} = \langle \text{AHI} = \text{ahi}, \text{UHI} = (), \text{HVF} = \text{x} \rangle \wedge$
 $\text{ASID } \text{ahi} = \text{asid} \wedge \text{ASIF } (\text{DownIF } \text{ahi}) \text{ downif} \wedge \text{ASIF } (\text{UpIF } \text{ahi}) \text{ upif} \wedge$
 $\text{mo} = \text{Some } \langle \text{AHI} = \text{ahi2}, \text{UHI} = (), \text{HVF} = \text{x2} \rangle \wedge$
 $\text{ASIF } (\text{DownIF } \text{ahi2}) \text{ downif2} \wedge \text{ASIF } (\text{UpIF } \text{ahi2}) \text{ upif2} \wedge$
 $\text{x} = \text{Mac}[\text{macKey } \text{asid}] (\text{L } [\text{tsn}, \text{upif}, \text{downif}, \text{upif2}, \text{downif2}, \text{x2}]) \wedge$
 $\text{tsn} = \text{Num } \text{ts} \wedge$
 $\text{uinfo} = \varepsilon)$
 $\vee (\exists \text{ahi} \text{ts} \text{upif} \text{downif} \text{asid} \text{x}.$
 $\text{hf} = \langle \text{AHI} = \text{ahi}, \text{UHI} = (), \text{HVF} = \text{x} \rangle \wedge$
 $\text{ASID } \text{ahi} = \text{asid} \wedge \text{ASIF } (\text{DownIF } \text{ahi}) \text{ downif} \wedge \text{ASIF } (\text{UpIF } \text{ahi}) \text{ upif} \wedge$
 $\text{mo} = \text{None} \wedge$
 $\text{x} = \text{Mac}[\text{macKey } \text{asid}] (\text{L } [\text{tsn}, \text{upif}, \text{downif}]) \wedge$
 $\text{tsn} = \text{Num } \text{ts} \wedge$
 $\text{uinfo} = \varepsilon)$
 $)$
 $\langle \text{proof} \rangle$

lemma *hf-valid-auth-restrict[dest]*: *hf-valid* *ainfo* *uinfo* *hf* *z* $\implies$ *auth-restrict* *ainfo* *uinfo* *l*
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

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
    <proof>

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  $\equiv$  {}
```

3.1.3 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
    <proof>
```

```
lemma auth-ainfo[dest]:  $\llbracket (ainfo, hfs) \in \text{auth-seg2 } uinfo \rrbracket \implies \exists ts . ainfo = \text{Num } ts$ 
    <proof>
```

```
lemma auth-uinfo[dest]:  $\llbracket (ainfo, hfs) \in \text{auth-seg2 } uinfo \rrbracket \implies uinfo = \varepsilon$ 
    <proof>
```

```
lemma upds-simp[simp]:  $TWu.upds \text{ upd-uinfo } uinfo \text{ hfs} = uinfo$ 
    <proof>
```

```
lemma upd-shifted-simp[simp]:  $TWu.upd-shifted \text{ upd-uinfo } uinfo \text{ hfs } \text{next} = uinfo$ 
    <proof>
```

```
lemma ik-hfs-form:  $t \in \text{parts } ik\text{-hfs} \implies \exists t' . t = \text{Hash } t'$ 
    <proof>
```

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

```
lemma parts-ik-hfs[simp]:  $\text{parts } ik\text{-hfs} = ik\text{-hfs}$ 
    <proof>
```

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*:

$$t \in ik\text{-}hfs \iff (\exists t' . t = Hash\ t') \wedge (\exists hf . t = HVF\ hf \\ \wedge (\exists hfs . hf \in set\ hfs \wedge (\exists ainfo . (ainfo, hfs) \in (auth\text{-}seg2\ \varepsilon) \\ \wedge (\exists\ next . hf\text{-}valid\ ainfo\ \varepsilon\ hf\ next)))) \text{ (is } ?lhs \iff ?rhs)$$

$\langle proof \rangle$

Properties of Intruder Knowledge

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

$\langle proof \rangle$

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

lemma *analz-parts-ik[simp]*: $analz\ ik = parts\ ik$

$\langle proof \rangle$

lemma *parts-ik[simp]*: $parts\ ik = ik$

$\langle proof \rangle$

lemma *key-ik-bad*: $Key\ (macK\ asid) \in ik \implies asid \in bad$

$\langle proof \rangle$

lemma *MAC-synth-helper*:

assumes $hf\text{-}valid\ ainfo\ uinfo\ m\ z\ HVF\ m = Mac[Key\ (macK\ asid)]\ j\ HVF\ m \in ik$

shows $\exists hfs . m \in set\ hfs \wedge (\exists uinfo' . (ainfo, hfs) \in auth\text{-}seg2\ uinfo')$

$\langle proof \rangle$

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**

$mac\text{-}format\ m\ asid \equiv \exists j . m = Mac[macKey\ asid]\ j$

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

lemma *MAC-synth*:

assumes $hf\text{-}valid\ ainfo\ uinfo\ m\ z\ HVF\ m \in synth\ ik\ mac\text{-}format\ (HVF\ m)\ asid$

$asid \notin bad\ checkInfo\ ainfo$

shows $\exists hfs . m \in set\ hfs \wedge (\exists uinfo' . (ainfo, hfs) \in auth\text{-}seg2\ uinfo')$

$\langle proof \rangle$

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

lemma *COND-honest-hf-analz*:

assumes $ASID\ (AHI\ hf) \notin bad\ hf\text{-}valid\ ainfo\ uinfo\ hf\ next\ terms\text{-}hf\ hf \subseteq synth\ (analz\ ik)$

$no\text{-}oracle\ ainfo\ uinfo$

shows $terms\text{-}hf\ hf \subseteq analz\ ik$

$\langle proof \rangle$

lemma *COND-terms-hf*:

assumes $hf\text{-}valid\ ainfo\ uinfo\ hf\ z$ **and** $terms\text{-}hf\ hf \subseteq analz\ ik$ **and** $no\text{-}oracle\ ainfo\ uinfo$

shows $\exists hfs . hf \in set\ hfs \wedge (\exists uinfo' . (ainfo, hfs) \in auth\text{-}seg2\ uinfo')$

$\langle \text{proof} \rangle$

lemma *COND-extr-prefix-path:*

$\llbracket \text{hfs-valid ainfo uinfo l next; next} = \text{None} \rrbracket \implies \text{prefix (extr-from-hd l) (AHIS l)}$

$\langle \text{proof} \rangle$

lemma *COND-path-prefix-extr:*

$\text{prefix (AHIS (hfs-valid-prefix ainfo uinfo l next))}$
 $\quad (\text{extr-from-hd l})$

$\langle \text{proof} \rangle$

lemma *COND-hf-valid-uinfo:*

$\llbracket \text{hf-valid ainfo uinfo hf next; hf-valid ainfo' uinfo' hf next} \rrbracket \implies \text{uinfo}' = \text{uinfo}$

$\langle \text{proof} \rangle$

lemma *COND-upd-uinfo-ik:*

$\llbracket \text{terms-uinfo uinfo} \subseteq \text{synth (analz ik); terms-hf hf} \subseteq \text{synth (analz ik)} \rrbracket$
 $\implies \text{terms-uinfo (upd-uinfo uinfo hf)} \subseteq \text{synth (analz ik)}$

$\langle \text{proof} \rangle$

lemma *COND-upd-uinfo-no-oracle:*

$\text{no-oracle ainfo uinfo} \implies \text{no-oracle ainfo (upd-uinfo uinfo fld)}$

$\langle \text{proof} \rangle$

lemma *COND-auth-restrict-upd:*

$\text{auth-restrict ainfo uinfo (x\#y\#hfs)}$
 $\implies \text{auth-restrict ainfo (upd-uinfo uinfo y) (y\#hfs)}$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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  $\times$  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  $\Rightarrow$  msgterm
   $\Rightarrow$  SCION-HF
   $\Rightarrow$  SCION-HF option  $\Rightarrow$  bool where
    hf-valid (Num ts) uinfo ( $\langle$  AHI = ahi, UHI = -, HVF = x  $\rangle$ ) (Some ( $\langle$  AHI = ahi2, UHI = -, HVF = x2  $\rangle$ ))  $\longleftrightarrow$ 
      ( $\exists$  upif downif. x = Mac[macKey (ASID ahi)] (L [Num ts, upif, downif, x2])  $\wedge$ 
        ASIF (DownIF ahi) downif  $\wedge$  ASIF (UpIF ahi) upif  $\wedge$  uinfo =  $\varepsilon$ )
  | hf-valid (Num ts) uinfo ( $\langle$  AHI = ahi, UHI = -, HVF = x  $\rangle$ ) None  $\longleftrightarrow$ 
      ( $\exists$  upif downif. x = Mac[macKey (ASID ahi)] (L [Num ts, upif, downif])  $\wedge$ 
        ASIF (DownIF ahi) downif  $\wedge$  ASIF (UpIF ahi) upif  $\wedge$  uinfo =  $\varepsilon$ )
  | hf-valid - - - = False

```

```

definition upd-uinfo :: msgterm  $\Rightarrow$  SCION-HF  $\Rightarrow$  msgterm where
  upd-uinfo uinfo hf  $\equiv$  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  $\Rightarrow$  ahi list where
  extr (Mac[macKey asid] (L [ts, upif, downif, x2]))
  = ( $\langle$  UpIF = term2if upif, DownIF = term2if downif, ASID = asid  $\rangle$  # extr x2)
  | extr (Mac[macKey asid] (L [ts, upif, downif]))
  = ( $\langle$  UpIF = term2if upif, DownIF = term2if downif, ASID = asid  $\rangle$ )
  | extr - = []

```

Extract the authenticated info field from a hop validation field.

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

abbreviation *term-ainfo* :: *msgterm* $\Rightarrow$ *msgterm* **where**
term-ainfo $\equiv$ *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* $\Rightarrow$ *msgterm* *set* **where**
terms-hf *hf* = {*HVF* *hf*}

abbreviation *terms-uinfo* :: *msgterm* $\Rightarrow$ *msgterm* *set* **where**
terms-uinfo *x* $\equiv$ {*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* $\equiv$ ($\exists$ *ts*. *ainfo* = *Num* *ts*) $\wedge$ (*uinfo* = ε)

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* *hf* *mo* $\longleftrightarrow$
($\exists$ *ahi* *ahi2* *ts* *upif* *downif* *asid* *x* *x2*.
hf = ($\langle$ *AHI* = *ahi*, *UHI* = (), *HVF* = *x* $\rangle$ $\wedge$
ASID *ahi* = *asid* $\wedge$ *ASIF* (*DownIF* *ahi*) *downif* $\wedge$ *ASIF* (*UpIF* *ahi*) *upif* $\wedge$
mo = *Some* ($\langle$ *AHI* = *ahi2*, *UHI* = (), *HVF* = *x2* $\rangle$ $\wedge$
x = *Mac*[*macKey* *asid*] (*L* [*tsn*, *upif*, *downif*, *x2*]) $\wedge$
tsn = *Num* *ts* $\wedge$
uinfo = ε)
 $\vee$ ($\exists$ *ahi* *ts* *upif* *downif* *asid* *x*.
hf = ($\langle$ *AHI* = *ahi*, *UHI* = (), *HVF* = *x* $\rangle$ $\wedge$
ASID *ahi* = *asid* $\wedge$ *ASIF* (*DownIF* *ahi*) *downif* $\wedge$ *ASIF* (*UpIF* *ahi*) *upif* $\wedge$
mo = *None* $\wedge$
x = *Mac*[*macKey* *asid*] (*L* [*tsn*, *upif*, *downif*]) $\wedge$
tsn = *Num* *ts* $\wedge$
uinfo = ε)
 $\rangle$
 $\langle$ proof $\rangle$

lemma *hf-valid-auth-restrict*[*dest*]: *hf-valid* *ainfo* *uinfo* *hf* *z* $\implies$ *auth-restrict* *ainfo* *uinfo* *l*
 $\langle$ proof $\rangle$

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*
 $\langle$ proof $\rangle$

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
    <proof>

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  $\equiv$  {}
```

3.3.3 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
    <proof>
```

```
lemma auth-ainfo[dest]:  $\llbracket (ainfo, hfs) \in \text{auth-seg2 } uinfo \rrbracket \implies \exists ts . ainfo = \text{Num } ts$ 
    <proof>
```

```
lemma auth-uinfo[dest]:  $\llbracket (ainfo, hfs) \in \text{auth-seg2 } uinfo \rrbracket \implies uinfo = \varepsilon$ 
    <proof>
```

```
lemma upds-simp[simp]:  $TWu.upds \text{ upd-uinfo } uinfo \text{ hfs} = uinfo$ 
    <proof>
```

```
lemma upd-shifted-simp[simp]:  $TWu.upd-shifted \text{ upd-uinfo } uinfo \text{ hfs } \text{next} = uinfo$ 
    <proof>
```

```
lemma ik-hfs-form:  $t \in \text{parts } ik\text{-hfs} \implies \exists t' . t = \text{Hash } t'$ 
    <proof>
```

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

```
lemma parts-ik-hfs[simp]:  $\text{parts } ik\text{-hfs} = ik\text{-hfs}$ 
    <proof>
```

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*:

$$t \in ik\text{-}hfs \iff (\exists t' . t = Hash\ t') \wedge (\exists hf . t = HVF\ hf \\ \wedge (\exists hfs. hf \in set\ hfs \wedge (\exists ainfo . (ainfo, hfs) \in (auth\text{-}seg2\ \varepsilon) \\ \wedge (\exists\ next. hf\text{-}valid\ ainfo\ \varepsilon\ hf\ next)))) \text{ (is } ?lhs \iff ?rhs)$$

$\langle proof \rangle$

Properties of Intruder Knowledge

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

$\langle proof \rangle$

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

lemma *analz-parts-ik[simp]*: $analz\ ik = parts\ ik$

$\langle proof \rangle$

lemma *parts-ik[simp]*: $parts\ ik = ik$

$\langle proof \rangle$

lemma *key-ik-bad*: $Key\ (macK\ asid) \in ik \implies asid \in bad$

$\langle proof \rangle$

lemma *MAC-synth-helper*:

assumes $hf\text{-}valid\ ainfo\ uinfo\ m\ z\ HVF\ m = Mac[Key\ (macK\ asid)]\ j\ HVF\ m \in ik$

shows $\exists hfs. m \in set\ hfs \wedge (\exists uinfo'. (ainfo, hfs) \in auth\text{-}seg2\ uinfo')$

$\langle proof \rangle$

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**

$mac\text{-}format\ m\ asid \equiv \exists j . m = Mac[macKey\ asid]\ j$

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

lemma *MAC-synth*:

assumes $hf\text{-}valid\ ainfo\ uinfo\ m\ z\ HVF\ m \in synth\ ik\ mac\text{-}format\ (HVF\ m)\ asid$

$asid \notin bad\ checkInfo\ ainfo$

shows $\exists hfs . m \in set\ hfs \wedge (\exists uinfo'. (ainfo, hfs) \in auth\text{-}seg2\ uinfo')$

$\langle proof \rangle$

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

lemma *COND-honest-hf-analz*:

assumes $ASID\ (AHI\ hf) \notin bad\ hf\text{-}valid\ ainfo\ uinfo\ hf\ next\ terms\text{-}hf\ hf \subseteq synth\ (analz\ ik)$

$no\text{-}oracle\ ainfo\ uinfo$

shows $terms\text{-}hf\ hf \subseteq analz\ ik$

$\langle proof \rangle$

lemma *COND-terms-hf*:

assumes $hf\text{-}valid\ ainfo\ uinfo\ hf\ z$ **and** $terms\text{-}hf\ hf \subseteq analz\ ik$ **and** $no\text{-}oracle\ ainfo\ uinfo$

shows $\exists hfs. hf \in set\ hfs \wedge (\exists uinfo'. (ainfo, hfs) \in auth\text{-}seg2\ uinfo')$

$\langle \text{proof} \rangle$

lemma *COND-extr-prefix-path:*

$\llbracket \text{hfs-valid ainfo uinfo l next; next} = \text{None} \rrbracket \implies \text{prefix (extr-from-hd l) (AHIS l)}$

$\langle \text{proof} \rangle$

lemma *COND-path-prefix-extr:*

$\text{prefix (AHIS (hfs-valid-prefix ainfo uinfo l next))}$
 (extr-from-hd l)

$\langle \text{proof} \rangle$

lemma *COND-hf-valid-uinfo:*

$\llbracket \text{hf-valid ainfo uinfo hf next; hf-valid ainfo' uinfo' hf next} \rrbracket \implies \text{uinfo}' = \text{uinfo}$

$\langle \text{proof} \rangle$

lemma *COND-upd-uinfo-ik:*

$\llbracket \text{terms-uinfo uinfo} \subseteq \text{synth (analz ik); terms-hf hf} \subseteq \text{synth (analz ik)} \rrbracket$
 $\implies \text{terms-uinfo (upd-uinfo uinfo hf)} \subseteq \text{synth (analz ik)}$

$\langle \text{proof} \rangle$

lemma *COND-upd-uinfo-no-oracle:* $\text{no-oracle ainfo uinfo} \implies \text{no-oracle ainfo (upd-uinfo-pkt m)}$

$\langle \text{proof} \rangle$

lemma *COND-auth-restrict-upd:*

$\text{auth-restrict ainfo uinfo (x\#y\#hfs)}$
 $\implies \text{auth-restrict ainfo (upd-uinfo uinfo y) (y\#hfs)}$

$\langle \text{proof} \rangle$

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

$\langle \text{proof} \rangle$

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  $\times$  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  $\Rightarrow$  UINFO
   $\Rightarrow$  EPIC-HF
   $\Rightarrow$  EPIC-HF option  $\Rightarrow$  bool where
    hf-valid (Num ts) tspkt ( $\langle$ AHI = ahi, UHI = uhi, HVF = x $\rangle$ ) (Some ( $\langle$ AHI = ahi2, UHI = uhi2, HVF = x2 $\rangle$ ))  $\longleftrightarrow$ 
      ( $\exists \sigma$  upif downif.  $\sigma$  = Mac[macKey (ASID ahi)] (L [Num ts, upif, downif, uhi2])  $\wedge$ 
        ASIF (DownIF ahi) downif  $\wedge$  ASIF (UpIF ahi) upif  $\wedge$  uhi = Hash  $\sigma$   $\wedge$  x = Mac[ $\sigma$ ]  $\langle$ Num
        ts, Num tspkt $\rangle$ )
    | hf-valid (Num ts) tspkt ( $\langle$ AHI = ahi, UHI = uhi, HVF = x $\rangle$ ) None  $\longleftrightarrow$ 
      ( $\exists \sigma$  upif downif.  $\sigma$  = Mac[macKey (ASID ahi)] (L [Num ts, upif, downif])  $\wedge$ 
        ASIF (DownIF ahi) downif  $\wedge$  ASIF (UpIF ahi) upif  $\wedge$  uhi = Hash  $\sigma$   $\wedge$  x = Mac[ $\sigma$ ]  $\langle$ Num
        ts, Num tspkt $\rangle$ )
    | hf-valid - - - = False

```

```

definition upd-uinfo :: nat  $\Rightarrow$  EPIC-HF  $\Rightarrow$  nat where

```

$upd-uinfo\ uinfo\ hf \equiv uinfo$

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  $\Rightarrow$  ahi list where
  extrUhi (Hash (Mac[macKey asid] (L [ts, upif, downif, uhi2])))
= ( $\Downarrow$  UpIF = term2if upif, DownIF = term2if downif, ASID = asid) # extrUhi uhi2
| extrUhi (Hash (Mac[macKey asid] (L [ts, upif, downif])))
= ( $\Downarrow$  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  $\Rightarrow$  ahi list where
  extr (Mac[ $\sigma$ ] -) = extrUhi (Hash  $\sigma$ )
| extr - = []
```

Extract the authenticated info field from a hop validation field.

```
fun extr-ainfo :: msgterm  $\Rightarrow$  msgterm where
  extr-ainfo (Mac[Mac[macKey asid] (L (Num ts # xs))] -) = Num ts
| extr-ainfo - =  $\varepsilon$ 
```

```
abbreviation term-ainfo :: msgterm  $\Rightarrow$  msgterm where
  term-ainfo  $\equiv id$ 
```

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  $\Rightarrow$  msgterm set where
  terms-hf hf = {HVF hf, UHI hf}
```

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

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  $\equiv (\exists ts. ainfo = Num\ ts)$ 
```

```
abbreviation no-oracle where no-oracle  $\equiv (\lambda - . True)$ 
```

We now define useful properties of the above definition.

lemma hf-valid-invert:

```
hf-valid tsn uinfo hf mo  $\longleftrightarrow$ 
  ( $\exists$  ahi ahi2  $\sigma$  ts upif downif asid x upif2 downif2 asid2 uhi uhi2 x2.
    hf = ( $\Downarrow$  AHI = ahi, UHI = uhi, HVF = x)  $\wedge$ 
    ASID ahi = asid  $\wedge$  ASIF (DownIF ahi) downif  $\wedge$  ASIF (UpIF ahi) upif  $\wedge$ 
```

$mo = \text{Some } (\lambda AHI = ahi2, UHI = uhi2, HVF = x2) \wedge$
 $ASID\ ahi2 = asid2 \wedge ASIF\ (\text{DownIF}\ ahi2)\ \text{downif2} \wedge ASIF\ (\text{UpIF}\ ahi2)\ \text{upif2} \wedge$
 $\sigma = \text{Mac}[\text{macKey}\ asid]\ (L\ [tsn, \text{upif}, \text{downif}, uhi2]) \wedge$
 $tsn = \text{Num}\ ts \wedge$
 $uhi = \text{Hash}\ \sigma \wedge$
 $x = \text{Mac}[\sigma]\ \langle tsn, \text{Num}\ uinfo \rangle$
 $\vee (\exists ahi\ \sigma\ ts\ \text{upif}\ \text{downif}\ asid\ uhi\ x.$
 $\quad hf = (\lambda AHI = ahi, UHI = uhi, HVF = x) \wedge$
 $\quad ASID\ ahi = asid \wedge ASIF\ (\text{DownIF}\ ahi)\ \text{downif} \wedge ASIF\ (\text{UpIF}\ ahi)\ \text{upif} \wedge$
 $\quad mo = \text{None} \wedge$
 $\quad \sigma = \text{Mac}[\text{macKey}\ asid]\ (L\ [tsn, \text{upif}, \text{downif}]) \wedge$
 $\quad tsn = \text{Num}\ ts \wedge$
 $\quad uhi = \text{Hash}\ \sigma \wedge$
 $\quad x = \text{Mac}[\sigma]\ \langle tsn, \text{Num}\ uinfo \rangle)$
 $\rangle$
 $\langle \text{proof} \rangle$

lemma *hf-valid-auth-restrict[dest]*: *hf-valid ainfo uinfo hf z $\implies$ auth-restrict ainfo uinfo l*
 $\langle \text{proof} \rangle$

lemma *auth-restrict-ainfo[dest]*: *auth-restrict ainfo uinfo l $\implies \exists ts. ainfo = \text{Num}\ ts$*
 $\langle \text{proof} \rangle$

lemma *info-hvf*:

assumes *hf-valid ainfo uinfo m z HVF m = Mac[σ] $\langle ainfo', \text{Num}\ uinfo' \rangle \vee hf\text{-valid ainfo' uinfo' m z'}$*
shows *ainfo = uinfo' ainfo' = ainfo*
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

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*

$ik\text{-add} \equiv \{ \sigma \mid ainfo\ uinfo\ l\ hf\ \sigma.$
 $(ainfo, l) \in \text{auth-seg2}\ uinfo \wedge hf \in \text{set}\ l \wedge HVF\ hf = \text{Mac}[\sigma]\ \langle ainfo, \text{Num}\ uinfo \rangle \}$

lemma *ik-addI*:

$\llbracket (ainfo, l) \in \text{auth-seg2}\ uinfo; hf \in \text{set}\ l; HVF\ hf = \text{Mac}[\sigma]\ \langle ainfo, \text{Num}\ uinfo \rangle \rrbracket \implies \sigma \in ik\text{-add}$
 $\langle \text{proof} \rangle$

lemma *ik-add-form*: $t \in ik\text{-}add \implies \exists \text{ asid } l . t = Mac[macKey \text{ asid}] l$
 <proof>

lemma *parts-ik-add[simp]*: $parts \text{ ik-add} = ik\text{-}add$
 <proof>

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
 <proof>

lemma *ik-hfs-form*: $t \in parts \text{ ik-hfs} \implies \exists t' . t = Hash \ t'$
 <proof>

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

lemma *parts-ik-hfs[simp]*: $parts \text{ ik-hfs} = ik\text{-hfs}$
 <proof>

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*:

$t \in ik\text{-hfs} \iff (\exists t' . t = Hash \ t') \wedge (\exists hf . (t = HVF \ hf \vee t = UHI \ hf) \wedge (\exists hfs. hf \in set \ hfs \wedge (\exists ainfo \ uinfo. (ainfo, hfs) \in auth\text{-}seg2 \ uinfo \wedge (\exists \text{ next}. hf\text{-}valid \ ainfo \ uinfo \ hf \text{ next})))) (\text{is } ?lhs \iff ?rhs)$

<proof>

Properties of Intruder Knowledge

lemma *auth-ainfo[dest]*: $\llbracket (ainfo, hfs) \in auth\text{-}seg2 \ uinfo \rrbracket \implies \exists ts . ainfo = Num \ ts$
 <proof>

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

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

lemma *analz-parts-ik[simp]*: $analz \text{ ik} = parts \text{ ik}$
 <proof>

lemma *parts-ik[simp]*: $parts \text{ ik} = ik$
 <proof>

lemma *key-ik-bad*: $Key (macK\ asid) \in ik \implies asid \in bad$
 <proof>

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* $\Rightarrow$ *EPIC-HF* $\Rightarrow$ *EPIC-HF* **where**
uinfo-change-hf new-uinfo hf =
 (case *HVF hf of Mac* $[\sigma] \langle ainfo, uinfo \rangle \Rightarrow hf(HVF := Mac[\sigma] \langle ainfo, Num\ new-uinfo \rangle) \mid - \Rightarrow hf$)

fun *uinfo-change* :: *UINFO* $\Rightarrow$ *EPIC-HF list* $\Rightarrow$ *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 next $\implies$ hfs-valid ainfo new-uinfo (uinfo-change new-uinfo l) next
 <proof>

lemma *uinfo-change-hf-AHI*: *AHI (uinfo-change-hf new-uinfo hf) = AHI hf*
 <proof>

lemma *uinfo-change-hf-AHIS[simp]*: *AHIS (map (uinfo-change-hf new-uinfo) l) = AHIS l*
 <proof>

lemma *uinfo-change-auth-seg2*:
assumes *hf-valid ainfo uinfo m z $\sigma = Mac[Key (macK\ asid)] j$*
HVF m = Mac $[\sigma] \langle ainfo, Num\ uinfo \rangle$ *$\sigma \in ik-add$*
shows $\exists hfs. m \in set\ hfs \wedge (\exists uinfo''. (ainfo, hfs) \in auth-seg2\ uinfo'')$
 <proof>

lemma *MAC-synth-helper*:
 $\llbracket hf-valid\ ainfo\ uinfo\ m\ z;$
HVF m = Mac $[\sigma] \langle ainfo, Num\ uinfo \rangle; \sigma = Mac[Key (macK\ asid)] j; \sigma \in ik \vee HVF\ m \in ik$
 $\implies \exists hfs. m \in set\ hfs \wedge (\exists uinfo'. (ainfo, hfs) \in auth-seg2\ uinfo')$
 <proof>

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

definition *mac-format* :: *msgterm* $\Rightarrow$ *as* $\Rightarrow$ *bool* **where**
mac-format m asid $\equiv \exists j\ ts\ uinfo. m = Mac[Mac[macKey\ asid] j] \langle Num\ ts, uinfo \rangle$

If a valid hop field is derivable by the attacker, but does not belong to the attacker, 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 $\in synth\ ik\ mac-format (HVF\ m)\ asid$*
asid $\notin bad$
shows $\exists hfs. m \in set\ hfs \wedge (\exists uinfo'. (ainfo, hfs) \in auth-seg2\ uinfo')$

$\langle proof \rangle$

3.4.4 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*

$\langle proof \rangle$

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*)

$\langle proof \rangle$

lemma *COND-extr-prefix-path*:

$\llbracket \text{hfs-valid ainfo uinfo l nxt; nxt} = \text{None} \rrbracket \implies \text{prefix (extr-from-hd l) (AHIS l)}$

$\langle proof \rangle$

lemma *COND-path-prefix-extr*:

prefix (AHIS (hfs-valid-prefix ainfo uinfo l nxt))
(extr-from-hd l)

$\langle proof \rangle$

lemma *COND-hf-valid-uinfo*:

$\llbracket \text{hf-valid ainfo uinfo hf nxt; hf-valid ainfo' uinfo' hf nxt} \rrbracket \implies \text{uinfo}' = \text{uinfo}$

$\langle proof \rangle$

lemma *COND-upd-uinfo-ik*:

$\llbracket \text{terms-uinfo uinfo} \subseteq \text{synth (analz ik); terms-hf hf} \subseteq \text{synth (analz ik)} \rrbracket$
 $\implies \text{terms-uinfo (upd-uinfo uinfo hf)} \subseteq \text{synth (analz ik)}$

$\langle proof \rangle$

lemma *COND-upd-uinfo-no-oracle*:

no-oracle ainfo uinfo $\implies$ *no-oracle ainfo (upd-uinfo uinfo fld)*

$\langle proof \rangle$

lemma *COND-auth-restrict-upd*:

auth-restrict ainfo uinfo (x#y#hfs)
 $\implies$ *auth-restrict ainfo (upd-uinfo uinfo y) (y#hfs)*

$\langle proof \rangle$

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

$\langle proof \rangle$

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  $\times$  ahi list) set +
    fixes no-oracle :: msgterm  $\Rightarrow$  UINFO  $\Rightarrow$  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  $\Rightarrow$  UINFO
   $\Rightarrow$  EPIC-HF
   $\Rightarrow$  EPIC-HF option  $\Rightarrow$  bool where
    hf-valid (Num ts) uinfo (AHI = ahi, UHI = uhi, HVF = x) (Some (AHI = ahi2, UHI = uhi2,
    HVF = x2))  $\longleftrightarrow$ 
      ( $\exists \sigma$  upif downif.  $\sigma$  = Mac[macKey (ASID ahi)] (L [Num ts, upif, downif, uhi2])  $\wedge$ 
        ASIF (DownIF ahi) downif  $\wedge$  ASIF (UpIF ahi) upif  $\wedge$  uhi = Hash  $\sigma$   $\wedge$  x = Mac[ $\sigma$ ] (Num
        ts, Num uinfo))
    | hf-valid (Num ts) uinfo (AHI = ahi, UHI = uhi, HVF = x) None  $\longleftrightarrow$ 
      ( $\exists \sigma$  upif downif.  $\sigma$  = Mac[macKey (ASID ahi)] (L [Num ts, upif, downif])  $\wedge$ 
        ASIF (DownIF ahi) downif  $\wedge$  ASIF (UpIF ahi) upif  $\wedge$  uhi = Hash  $\sigma$   $\wedge$  x = Mac[ $\sigma$ ] (Num
        ts, Num uinfo))
    | hf-valid - - - = False

```

definition $upd-uinfo :: nat \Rightarrow EPIC-HF \Rightarrow nat$ **where**
 $upd-uinfo\ uinfo\ hf \equiv uinfo$

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 $extrUhi :: msgterm \Rightarrow ahi\ list$ **where**
 $extrUhi\ (Hash\ (Mac[macKey\ asid]\ (L\ [ts,\ upif,\ downif,\ uhi2])))$
 $= \langle UpIF = term2if\ upif,\ DownIF = term2if\ downif,\ ASID = asid \rangle \# extrUhi\ uhi2$
 $| extrUhi\ (Hash\ (Mac[macKey\ asid]\ (L\ [ts,\ upif,\ downif])))$
 $= \langle UpIF = term2if\ upif,\ DownIF = term2if\ downif,\ ASID = asid \rangle$
 $| extrUhi\ - = []$

This function extracts from a hop validation field (HVF hf) the entire path.

fun $extr :: msgterm \Rightarrow ahi\ list$ **where**
 $extr\ (Mac[\sigma]\ -) = extrUhi\ (Hash\ \sigma)$
 $| extr\ - = []$

Extract the authenticated info field from a hop validation field.

fun $extr-ainfo :: msgterm \Rightarrow msgterm$ **where**
 $extr-ainfo\ (Mac[-]\ \langle Num\ ts,\ - \rangle) = Num\ ts$
 $| extr-ainfo\ - = \varepsilon$

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

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

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

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

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 $hf-valid-invert$:

$hf-valid\ tsn\ uinfo\ hf\ mo \longleftrightarrow$
 $(\exists\ ahi\ ahi2\ \sigma\ ts\ upif\ downif\ asid\ x\ upif2\ downif2\ asid2\ uhi\ uhi2\ x2.$
 $hf = \langle AHI = ahi,\ UHI = uhi,\ HVF = x \rangle \wedge$
 $ASID\ ahi = asid \wedge ASIF\ (DownIF\ ahi)\ downif \wedge ASIF\ (UpIF\ ahi)\ upif \wedge$

$mo = \text{Some } (\lambda AHI = ahi2, UHI = uhi2, HVF = x2) \wedge$
 $ASID\ ahi2 = asid2 \wedge ASIF\ (\text{DownIF}\ ahi2)\ \text{downif2} \wedge ASIF\ (\text{UpIF}\ ahi2)\ \text{upif2} \wedge$
 $\sigma = \text{Mac}[\text{macKey}\ asid]\ (L\ [tsn, \text{upif}, \text{downif}, uhi2]) \wedge$
 $tsn = \text{Num}\ ts \wedge$
 $uhi = \text{Hash}\ \sigma \wedge$
 $x = \text{Mac}[\sigma]\ \langle tsn, \text{Num}\ uinfo \rangle$
 $\vee (\exists ahi\ \sigma\ ts\ \text{upif}\ \text{downif}\ asid\ uhi\ x.$
 $\quad hf = (\lambda AHI = ahi, UHI = uhi, HVF = x) \wedge$
 $\quad ASID\ ahi = asid \wedge ASIF\ (\text{DownIF}\ ahi)\ \text{downif} \wedge ASIF\ (\text{UpIF}\ ahi)\ \text{upif} \wedge$
 $\quad mo = \text{None} \wedge$
 $\quad \sigma = \text{Mac}[\text{macKey}\ asid]\ (L\ [tsn, \text{upif}, \text{downif}]) \wedge$
 $\quad tsn = \text{Num}\ ts \wedge$
 $\quad uhi = \text{Hash}\ \sigma \wedge$
 $\quad x = \text{Mac}[\sigma]\ \langle tsn, \text{Num}\ uinfo \rangle)$
 $\rangle$
 $\langle \text{proof} \rangle$

lemma *hf-valid-auth-restrict[dest]: hf-valid ainfo uinfo hf z $\implies$ auth-restrict ainfo uinfo l*
 $\langle \text{proof} \rangle$

lemma *auth-restrict-ainfo[dest]: auth-restrict ainfo uinfo l $\implies \exists ts. ainfo = \text{Num}\ ts$*
 $\langle \text{proof} \rangle$

lemma *info-hvf:*

assumes *hf-valid ainfo uinfo m z HVF m = Mac[σ] ⟨ainfo', Num uinfo'⟩ $\vee$ hf-valid ainfo' uinfo' m z'*

shows *ainfo = uinfo' ainfo' = ainfo*

$\langle \text{proof} \rangle$

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*

$\langle \text{proof} \rangle$

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*

$ik-add \equiv \{ \sigma \mid ainfo\ uinfo\ l\ hf\ \sigma.$
 $\quad (ainfo::msgterm, l::(EPIC-HF\ list)) \in$
 $\quad ((local.auth-seg2\ uinfo)::(msgterm \times EPIC-HF\ list)\ set))$

$$\wedge hf \in \text{set } l \wedge HVF \ hf = Mac[\sigma] \ \langle ainfo, Num \ uinfo \rangle \}$$

lemma *ik-addI*:

$$\llbracket (ainfo, l) \in local.auth-seg2 \ uinfo; hf \in \text{set } l; HVF \ hf = Mac[\sigma] \ \langle ainfo, Num \ uinfo \rangle \rrbracket \implies \sigma \in ik-add \ \langle proof \rangle$$

lemma *ik-add-form*: $t \in local.ik-add \implies \exists \ asid \ l . t = Mac[macKey \ asid] \ l$

$\langle proof \rangle$

lemma *parts-ik-add[simp]*: $parts \ ik-add = ik-add$

$\langle proof \rangle$

This is the oracle output provided to the adversary. Only those hop validation fields and segment identifiers whose path origin (combination of *ainfo* *uinfo*) is not contained in *no-oracle* appears here.

definition *ik-oracle* :: *msgterm set* **where**

$$ik-oracle = \{ t \mid t \ ainfo \ hf \ l \ uinfo . hf \in \text{set } l \wedge hfs-valid-None \ ainfo \ uinfo \ l \wedge is-oracle \ ainfo \ uinfo \wedge (\forall \ uinfo' . (ainfo, l) \notin auth-seg2 \ uinfo') \wedge (t = HVF \ hf \vee t = UHI \ hf) \}$$

lemma *ik-oracle-parts-form*:

$$t \in ik-oracle \implies$$

$$(\exists \ asid \ l \ ainfo \ uinfo . t = Mac[Mac[macKey \ asid] \ l] \ \langle ainfo, uinfo \rangle) \vee$$

$$(\exists \ asid \ l . t = Hash \ (Mac[macKey \ asid] \ l))$$

$\langle proof \rangle$

lemma *parts-ik-oracle[simp]*: $parts \ ik-oracle = ik-oracle$

$\langle proof \rangle$

lemma *ik-oracle-simp*: $t \in ik-oracle \longleftrightarrow$

$$(\exists \ ainfo \ hf \ l \ uinfo . hf \in \text{set } l \wedge hfs-valid-None \ ainfo \ uinfo \ l \wedge is-oracle \ ainfo \ uinfo \wedge (\forall \ uinfo' . (ainfo, l) \notin auth-seg2 \ uinfo') \wedge (t = HVF \ hf \vee t = UHI \ hf))$$

$\langle proof \rangle$

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

$\langle proof \rangle$

lemma *ik-hfs-form*: $t \in parts \ ik-hfs \implies \exists \ t' . t = Hash \ t'$

$\langle proof \rangle$

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

lemma *parts-ik-hfs[simp]*: *parts ik-hfs = ik-hfs*
 ⟨*proof*⟩

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*:

$$t \in ik-hfs \longleftrightarrow (\exists t' . t = Hash\ t') \wedge (\exists hf . (t = HVF\ hf \vee t = UHI\ hf) \\ \wedge (\exists hfs . hf \in set\ hfs \wedge (\exists ainfo\ uinfo . (ainfo, hfs) \in auth-seg2\ uinfo \\ \wedge (\exists nxt . hf-valid\ ainfo\ uinfo\ hf\ nxt)))) \text{ (is } ?lhs \longleftrightarrow ?rhs)$$

 ⟨*proof*⟩

Properties of Intruder Knowledge

lemma *auth-ainfo[dest]*: $\llbracket (ainfo, hfs) \in auth-seg2\ uinfo \rrbracket \implies \exists ts . ainfo = Num\ ts$
 ⟨*proof*⟩

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*
 ⟨*proof*⟩

lemma *parts-ik[simp]*: *parts ik = ik*
 ⟨*proof*⟩

lemma *key-ik-bad*: $Key\ (macK\ asid) \in ik \implies asid \in bad$
 ⟨*proof*⟩

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] $\langle ainfo, uinfo \rangle \Rightarrow hf(HVF := Mac[\sigma]\ \langle ainfo, Num\ new-uinfo \rangle) \mid - \Rightarrow hf)$

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

lemma *uinfo-change-valid*:
 $hfs-valid\ ainfo\ uinfo\ l\ nxt \implies hfs-valid\ ainfo\ new-uinfo\ (uinfo-change\ new-uinfo\ l)\ nxt$
 ⟨*proof*⟩

lemma *uinfo-change-hf-AHI*: $AHI\ (uinfo-change-hf\ new-uinfo\ hf) = AHI\ hf$
 ⟨*proof*⟩

lemma *uinfo-change-hf-AHIS[simp]*: $AHIS\ (map\ (uinfo-change-hf\ new-uinfo)\ l) = AHIS\ l$
 ⟨*proof*⟩

lemma *uinfo-change-auth-seg2*:

assumes $hf\text{-valid } ainfo \ uinfo \ m \ z \ \sigma = Mac[Key \ (macK \ asid)] \ j$
 $HVF \ m = Mac[\sigma] \ \langle ainfo, Num \ uinfo \rangle \ \sigma \in ik\text{-add no-oracle } ainfo \ uinfo$
shows $\exists hfs. m \in set \ hfs \wedge (\exists uinfo''. (ainfo, hfs) \in auth\text{-seg2 } uinfo'')$
 $\langle proof \rangle$

lemma *MAC-synth-oracle*:

assumes $hf\text{-valid } ainfo \ uinfo \ m \ z \ HVF \ m \in ik\text{-oracle}$
shows $is\text{-oracle } ainfo \ uinfo$
 $\langle proof \rangle$

lemma *ik-oracle-is-oracle*:

$\llbracket Mac[\sigma] \ \langle ainfo, Num \ uinfo \rangle \in ik\text{-oracle} \rrbracket \implies is\text{-oracle } ainfo \ uinfo$
 $\langle proof \rangle$

lemma *MAC-synth-helper*:

$\llbracket hf\text{-valid } ainfo \ uinfo \ m \ z; no\text{-oracle } ainfo \ uinfo; HVF \ m = Mac[\sigma] \ \langle ainfo, Num \ uinfo \rangle; \sigma = Mac[Key \ (macK \ asid)] \ j; \sigma \in ik \vee HVF \ m \in ik \rrbracket$
 $\implies \exists hfs. m \in set \ hfs \wedge (\exists uinfo'. (ainfo, hfs) \in auth\text{-seg2 } uinfo')$
 $\langle proof \rangle$

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

definition $mac\text{-format} :: msgterm \Rightarrow as \Rightarrow bool$ **where**

$mac\text{-format } m \ asid \equiv \exists j \ ts \ uinfo. m = Mac[Mac[macKey \ asid] \ j] \ \langle Num \ ts, uinfo \rangle$

If a valid hop field is derivable by the attacker, but does not belong to the attacker, and is over a path origin that does not belong to an oracle query, then the hop field is already contained in the set of authorized segments.

lemma *MAC-synth*:

assumes $hf\text{-valid } ainfo \ uinfo \ m \ z \ HVF \ m \in synth \ ik \ mac\text{-format } (HVF \ m) \ asid$
 $asid \notin bad \ no\text{-oracle } ainfo \ uinfo$
shows $\exists hfs. m \in set \ hfs \wedge (\exists uinfo'. (ainfo, hfs) \in auth\text{-seg2 } uinfo')$
 $\langle proof \rangle$

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

lemma *COND-honest-hf-analz*:

assumes $ASID \ (AHI \ hf) \notin bad \ hf\text{-valid } ainfo \ uinfo \ hf \ next \ terms\text{-hf } hf \subseteq synth \ (analz \ ik)$
 $no\text{-oracle } ainfo \ uinfo$
shows $terms\text{-hf } hf \subseteq analz \ ik$
 $\langle proof \rangle$

lemma *COND-terms-hf*:

assumes $hf\text{-valid } ainfo \ uinfo \ hf \ z$ **and** $HVF \ hf \in ik$ **and** $no\text{-oracle } ainfo \ uinfo$
shows $\exists hfs. hf \in set \ hfs \wedge (\exists uinfo. (ainfo, hfs) \in auth\text{-seg2 } uinfo)$
 $\langle proof \rangle$

lemma *COND-extr-prefix-path*:

$\llbracket hfs\text{-valid } ainfo \ uinfo \ l \ next; next = None \rrbracket \implies prefix \ (extr\text{-from-hd } l) \ (AHIS \ l)$
 $\langle proof \rangle$

lemma *COND-path-prefix-extr:*

prefix (AHIS (hfs-valid-prefix ainfo uinfo l nxt))
(extr-from-hd l)

<proof>

lemma *COND-hf-valid-uinfo:*

$\llbracket hf\text{-valid } ainfo \text{ uinfo hf } nxt; hf\text{-valid } ainfo' \text{ uinfo}' hf \text{ } nxt \rrbracket \implies uinfo' = uinfo$

<proof>

lemma *COND-upd-uinfo-ik:*

$\llbracket terms\text{-uinfo } uinfo \subseteq synth \text{ (analz ik)}; terms\text{-hf hf} \subseteq synth \text{ (analz ik)} \rrbracket$
 $\implies terms\text{-uinfo (upd-uinfo uinfo hf)} \subseteq synth \text{ (analz ik)}$

<proof>

lemma *COND-upd-uinfo-no-oracle:*

$no\text{-oracle } ainfo \text{ uinfo} \implies no\text{-oracle } ainfo \text{ (upd-uinfo uinfo fld)}$

<proof>

lemma *COND-auth-restrict-upd:*

$auth\text{-restrict } ainfo \text{ uinfo (x\#y\#hfs)}$
 $\implies auth\text{-restrict } ainfo \text{ (upd-uinfo uinfo y) (y\#hfs)}$

<proof>

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

<proof>

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  $\equiv$  3
abbreviation nB :: as where nB  $\equiv$  4
abbreviation nC :: as where nC  $\equiv$  5
abbreviation nD :: as where nD  $\equiv$  6
abbreviation nE :: as where nE  $\equiv$  7
abbreviation nF :: as where nF  $\equiv$  8
abbreviation nG :: as where nG  $\equiv$  9
```

```
abbreviation bad :: as set where bad  $\equiv$  {nF}
```

We assume a complete graph, in which interfaces contain the name of the adjacent AS

```
fun tgtas :: as  $\Rightarrow$  ifs  $\Rightarrow$  as option where
  tgtas a i = Some i
fun tgtif :: as  $\Rightarrow$  ifs  $\Rightarrow$  ifs option where
  tgtif a i = Some a
```

3.6.1 Left segment

```
abbreviation hiAl :: ahi where hiAl  $\equiv$  ( $\Downarrow$  UpIF = None, DownIF = Some nB, ASID = nA)
abbreviation hiBl :: ahi where hiBl  $\equiv$  ( $\Downarrow$  UpIF = Some nA, DownIF = Some nD, ASID = nB)
abbreviation hiDl :: ahi where hiDl  $\equiv$  ( $\Downarrow$  UpIF = Some nB, DownIF = Some nE, ASID = nD)
abbreviation hiEl :: ahi where hiEl  $\equiv$  ( $\Downarrow$  UpIF = Some nD, DownIF = Some nF, ASID = nE)
abbreviation hiFl :: ahi where hiFl  $\equiv$  ( $\Downarrow$  UpIF = Some nE, DownIF = None, ASID = nF)
```

3.6.2 Right segment

```
abbreviation hiAr :: ahi where hiAr  $\equiv$  ( $\Downarrow$  UpIF = None, DownIF = Some nB, ASID = nA)
abbreviation hiBr :: ahi where hiBr  $\equiv$  ( $\Downarrow$  UpIF = Some nA, DownIF = Some nD, ASID = nB)
abbreviation hiDr :: ahi where hiDr  $\equiv$  ( $\Downarrow$  UpIF = Some nB, DownIF = Some nE, ASID = nD)
abbreviation hiEr :: ahi where hiEr  $\equiv$  ( $\Downarrow$  UpIF = Some nD, DownIF = Some nG, ASID = nE)
abbreviation hiGr :: ahi where hiGr  $\equiv$  ( $\Downarrow$  UpIF = Some nE, DownIF = None, ASID = nG)
```

```
abbreviation hfF-attr-E :: ahi set where hfF-attr-E  $\equiv$  {hi . ASID hi = nF  $\wedge$  UpIF hi = Some nE}
abbreviation hfF-attr :: ahi set where hfF-attr  $\equiv$  {hi . ASID hi = nF}
```

```
abbreviation lefthpath :: ahi list where
  lefthpath  $\equiv$  [hiFl, hiEl, hiDl, hiBl, hiAl]
abbreviation righthpath :: ahi list where
  righthpath  $\equiv$  [hiGr, hiEr, hiDr, hiBr, hiAr]
abbreviation rightsegment where rightsegment  $\equiv$  (Num 0, righthpath)
```

abbreviation *leftpath-wormholed* :: *ahi list set* **where**

leftpath-wormholed $\equiv$
 $\{ xs@[hf, hiEl, hiDl, hiBl, hiAl] \mid hf \in hfF\text{-}attr\text{-}E \wedge set\ xs \subseteq hfF\text{-}attr \}$

definition *leftsegment-wormholed* :: (*msgterm* $\times$ *ahi list*) *set* **where**

leftsegment-wormholed = $\{ (Num\ 0, leftpath) \mid leftpath . leftpath \in leftpath\text{-}wormholed \}$

definition *attr-segment* :: (*msgterm* $\times$ *ahi list*) *set* **where**

attr-segment = $\{ (ainfo, path) \mid ainfo\ path . set\ path \subseteq hfF\text{-}attr \}$

definition *auth-seg0* :: (*msgterm* $\times$ *ahi list*) *set* **where**

auth-seg0 = *leftsegment-wormholed* $\cup \{rightsegment\} \cup attr\text{-}segment$

lemma *tgtasif-inv*:

$\llbracket tgtas\ u\ i = Some\ v;\ tgtif\ u\ i = Some\ j \rrbracket \implies tgtas\ v\ j = Some\ u$

$\llbracket tgtas\ u\ i = Some\ v; \llbracket tgtif\ u\ i = Some\ j \rrbracket \implies tgtif\ v\ j = Some\ i$

<proof>

locale *no-assumptions-left*

begin

sublocale *d0*: *network-model bad auth-seg0 tgtas tgtif*

<proof>

lemma *attr-ifs-valid*: $\llbracket ASID\ y = nF; set\ ys \subseteq hfF\text{-}attr \rrbracket \implies d0.ifs\text{-}valid\ (Some\ y)\ ys\ next$

<proof>

lemma *attr-ifs-valid'*: $\llbracket set\ ys \subseteq hfF\text{-}attr; pre = None \rrbracket \implies d0.ifs\text{-}valid\ pre\ ys\ next$

<proof>

lemma *leftpath-ifs-valid*: $\llbracket pre = None; ASID\ hf = nF; UpIF\ hf = Some\ nE; set\ xs \subseteq hfF\text{-}attr \rrbracket$

$\implies d0.ifs\text{-}valid\ pre\ (xs\ @\ [hf, hiEl, hiDl, hiBl, hiAl])\ next$

<proof>

lemma *ASM-if-valid*: $\llbracket (info, l) \in auth\text{-}seg0; pre = None \rrbracket \implies d0.ifs\text{-}valid\ pre\ l\ next$

<proof>

lemma *rooted-app[simp]*: $d0.rooted\ (xs@y\#\ ys) \longleftrightarrow d0.rooted\ (y\#\ ys)$

<proof>

lemma *ASM-rooted*: $(info, l) \in auth\text{-}seg0 \implies d0.rooted\ l$

<proof>

lemma *ASM-terminated*: $(info, l) \in auth\text{-}seg0 \implies d0.terminated\ l$

<proof>

lemma *ASM-empty*: $(info, []) \in auth\text{-}seg0$

<proof>

lemma *ASM-singleton*: $\llbracket ASID\ hf \in bad \rrbracket \implies (info, [hf]) \in auth\text{-}seg0$

<proof>

lemma *ASM-extension*:

$\llbracket (info, hf2 \# ys) \in auth-seg0; ASID\ hf2 \in bad; ASID\ hf1 \in bad \rrbracket$
 $\implies (info, hf1 \# hf2 \# ys) \in auth-seg0$
 $\langle proof \rangle$

lemma *ASM-modify*: $\llbracket (info, hf \# ys) \in auth-seg0; ASID\ hf = a;$

$ASID\ hf' = a; UpIF\ hf' = UpIF\ hf; a \in bad \rrbracket \implies (info, hf' \# ys) \in auth-seg0$

$\langle proof \rangle$

lemma *rightpath-no-nF*: $\llbracket ASID\ hf = nF; zs @ hf \# ys = rightpath \rrbracket \implies False$

$\langle proof \rangle$

lemma *ASM-cutoff-leftpath*:

$\llbracket ASID\ hf = nF;$

$\forall hfa. UpIF\ hfa = Some\ nE \longrightarrow ASID\ hfa = nF \longrightarrow (\forall xs. hf \# ys = xs @ [hfa, hiEl, hiDr, hiBr,$
 $hiAr] \longrightarrow$

$\neg set\ xs \subseteq hfF-attr; x \in set\ ys; info = Num\ 0;$

$zs @ hf \# ys = xs @ [hfa, hiEl, hiDr, hiBr, hiAr]; ASID\ hfa = nF; UpIF\ hfa = Some\ nE; set$
 $xs \subseteq hfF-attr \rrbracket$

$\implies ASID\ x = nF$

$\langle proof \rangle$

lemma *ASM-cutoff*: $\llbracket (info, zs @ hf \# ys) \in auth-seg0; ASID\ hf \in bad \rrbracket \implies (info, hf \# ys) \in auth-seg0$

$\langle proof \rangle$

sublocale *network-assums-direct-instance*: *network-assums-direct bad tgtas tgtif auth-seg0*

$\langle proof \rangle$

definition *no-oracle* :: *msgterm* $\Rightarrow$ *nat* $\Rightarrow$ *bool* **where**

no-oracle ainfo uinfo = *True*

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

$\langle proof \rangle$

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* $\equiv$ *Num 0*

abbreviation *uinfo* :: *nat* **where** *uinfo* $\equiv$ *1*

abbreviation σA **where** $\sigma A \equiv Mac[macKey\ nA]\ (L\ [ainfo, \varepsilon, AS\ nB])$

abbreviation σB **where** $\sigma B \equiv Mac[macKey\ nB]\ (L\ [ainfo, AS\ nA, AS\ nD, Hash\ \sigma A])$

abbreviation σD **where** $\sigma D \equiv Mac[macKey\ nD]\ (L\ [ainfo, AS\ nB, AS\ nE, Hash\ \sigma B])$

abbreviation σE **where** $\sigma E \equiv Mac[macKey\ nE]\ (L\ [ainfo, AS\ nD, AS\ nF, Hash\ \sigma D])$

abbreviation σF **where** $\sigma F \equiv Mac[macKey\ nF]\ (L\ [ainfo, AS\ nE, \varepsilon, Hash\ \sigma E])$

definition *hfAl* **where** *hfAl* $\equiv \langle AHI = hiAl, UHI = Hash\ \sigma A, HVF = Mac[\sigma A] \langle ainfo, Num\ uinfo \rangle \rangle$
definition *hfBl* **where** *hfBl* $\equiv \langle AHI = hiBl, UHI = Hash\ \sigma B, HVF = Mac[\sigma B] \langle ainfo, Num\ uinfo \rangle \rangle$
definition *hfDl* **where** *hfDl* $\equiv \langle AHI = hiDl, UHI = Hash\ \sigma D, HVF = Mac[\sigma D] \langle ainfo, Num\ uinfo \rangle \rangle$
definition *hfEl* **where** *hfEl* $\equiv \langle AHI = hiEl, UHI = Hash\ \sigma E, HVF = Mac[\sigma E] \langle ainfo, Num\ uinfo \rangle \rangle$
definition *hfFl* **where** *hfFl* $\equiv \langle AHI = hiFl, UHI = Hash\ \sigma F, HVF = Mac[\sigma F] \langle ainfo, Num\ uinfo \rangle \rangle$

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

lemma *e1.hf-valid ainfo uinfo hfAl None*
 $\langle proof \rangle$

lemma *e1.hf-valid ainfo uinfo hfBl (Some hfAl)*
 $\langle proof \rangle$

lemma *e1.hf-valid ainfo uinfo hfFl (Some hfEl)*
 $\langle proof \rangle$

abbreviation *forwardingpath* **where**
forwardingpath $\equiv [hfFl, hfEl, hfDl, hfBl, hfAl]$

definition *pkt0* **where** *pkt0* $\equiv \langle$
 AInfo = *ainfo*,
 UInfo = *uinfo*,
 past = [],
 future = *forwardingpath*,
 history = []
 $\rangle$

definition *pkt1* **where** *pkt1* $\equiv \langle$
 AInfo = *ainfo*,
 UInfo = *uinfo*,
 past = [*hfFl*],
 future = [*hfEl*, *hfDl*, *hfBl*, *hfAl*],
 history = [*hiFl*]
 $\rangle$

definition *pkt2* **where** *pkt2* $\equiv \langle$
 AInfo = *ainfo*,
 UInfo = *uinfo*,
 past = [*hfEl*, *hfFl*],
 future = [*hfDl*, *hfBl*, *hfAl*],
 history = [*hiEl*, *hiFl*]
 $\rangle$

definition *pkt3* **where** *pkt3* $\equiv \langle$
 AInfo = *ainfo*,
 UInfo = *uinfo*,
 past = [*hfDl*, *hfEl*, *hfFl*],
 future = [*hfBl*, *hfAl*],
 history = [*hiDl*, *hiEl*, *hiFl*]
 $\rangle$

definition *pkt4* **where** *pkt4* $\equiv \langle$
 AInfo = *ainfo*,
 UInfo = *uinfo*,
 past = [*hfBl*, *hfDl*, *hfEl*, *hfFl*],
 future = [*hfAl*],
 history = [*hiBl*, *hiDl*, *hiEl*, *hiFl*]
 $\rangle$

$\rangle$
definition $pkt5$ **where** $pkt5 \equiv ()$
 $AInfo = ainfo,$
 $UInfo = uinfo,$
 $past = [hfAl, hfBl, hfDl, hfEl, hfFl],$
 $future = [],$
 $history = [hiAl, hiBl, hiDl, hiEl, hiFl]$
 $\rangle$

definition $s0$ **where** $s0 \equiv e1.dp2-init$

definition $s1$ **where** $s1 \equiv s0 \langle loc2 := (loc2\ s0)(nF := \{pkt0\}) \rangle$

definition $s2$ **where**

$s2 \equiv s1 \langle chan2 := (chan2\ s1)((nF, nE, nE, nF) := chan2\ s1\ (nF, nE, nE, nF) \cup \{pkt1\}) \rangle$

definition $s3$ **where** $s3 \equiv s2 \langle loc2 := (loc2\ s2)(nE := \{pkt1\}) \rangle$

definition $s4$ **where**

$s4 \equiv s3 \langle chan2 := (chan2\ s3)((nE, nD, nD, nE) := chan2\ s3\ (nE, nD, nD, nE) \cup \{pkt2\}) \rangle$

definition $s5$ **where** $s5 \equiv s4 \langle loc2 := (loc2\ s4)(nD := \{pkt2\}) \rangle$

definition $s6$ **where**

$s6 \equiv s5 \langle chan2 := (chan2\ s5)((nD, nB, nB, nD) := chan2\ s5\ (nD, nB, nB, nD) \cup \{pkt3\}) \rangle$

definition $s7$ **where** $s7 \equiv s6 \langle loc2 := (loc2\ s6)(nB := \{pkt3\}) \rangle$

definition $s8$ **where**

$s8 \equiv s7 \langle chan2 := (chan2\ s7)((nB, nA, nA, nB) := chan2\ s7\ (nB, nA, nA, nB) \cup \{pkt4\}) \rangle$

definition $s9$ **where** $s9 \equiv s8 \langle loc2 := (loc2\ s8)(nA := \{pkt4\}) \rangle$

definition $s10$ **where** $s10 \equiv s9 \langle loc2 := (loc2\ s9)(nA := \{pkt4, pkt5\}) \rangle$

lemmas $forwarding-states =$

$s0-def\ s1-def\ s2-def\ s3-def\ s4-def\ s5-def\ s6-def\ s7-def\ s8-def\ s9-def\ s10-def$

lemma $forwardingpath-valid: e1.hfs-valid-None\ ainfo\ uinfo\ forwardingpath$
 $\langle proof \rangle$

lemma $forwardingpath-auth: pfragment\ ainfo\ forwardingpath\ (e1.auth-seg2\ uinfo)$
 $\langle proof \rangle$

lemma $reach-s0: reach\ e1.dp2\ s0\ \langle proof \rangle$

lemma $s0-s1: e1.dp2: s0 -evt-dispatch-int2\ nF\ pkt0 \rightarrow s1$
 $\langle proof \rangle$

lemma $s1-s2: e1.dp2: s1 -evt-send2\ nF\ nE\ pkt0 \rightarrow s2$
 $\langle proof \rangle$

lemma $s2-s3: e1.dp2: s2 -evt-recv2\ nE\ nF\ pkt1 \rightarrow s3$
 $\langle proof \rangle$

lemma $s3-s4: e1.dp2: s3 -evt-send2\ nE\ nD\ pkt1 \rightarrow s4$
 $\langle proof \rangle$

lemma $s4-s5: e1.dp2: s4 -evt-recv2\ nD\ nE\ pkt2 \rightarrow s5$
 $\langle proof \rangle$

lemma $s5-s6: e1.dp2: s5 -evt-send2\ nD\ nB\ pkt2 \rightarrow s6$

$\langle proof \rangle$

lemma $s6-s7$: $e1.dp2$: $s6 - evt-recv2\ nB\ nD\ pkt3 \rightarrow s7$
 $\langle proof \rangle$

lemma $s7-s8$: $e1.dp2$: $s7 - evt-send2\ nB\ nA\ pkt3 \rightarrow s8$
 $\langle proof \rangle$

lemma $s8-s9$: $e1.dp2$: $s8 - evt-recv2\ nA\ nB\ pkt4 \rightarrow s9$
 $\langle proof \rangle$

lemma $s9-s10$: $e1.dp2$: $s9 - evt-deliver2\ nA\ pkt4 \rightarrow s10$
 $\langle proof \rangle$

The state in which the packet is received is reachable

lemma *executability*: $reach\ e1.dp2\ s10$
 $\langle proof \rangle$

Attacker event executability

We also show that the attacker event can be executed.

definition $pkt-attr$ **where** $pkt-attr \equiv ()$
 $AInfo = ainfo,$
 $UInfo = uinfo,$
 $past = [],$
 $future = [hfEl],$
 $history = []$
 $\rangle$

definition $s-attr$ **where**

$s-attr \equiv s0()chan2 := (chan2\ s0)((nF, nE, nE, nF) := chan2\ s0\ (nF, nE, nE, nF) \cup \{pkt-attr\})()$

lemma $ik-hfs-in-ik$: $t \in e1.ik-hfs \implies t \in synth\ (analz\ (e1.ik-dyn\ s))$
 $\langle proof \rangle$

lemma $hvf-e-auth$: $HVF\ hfEl \in e1.ik-hfs$
 $\langle proof \rangle$

lemma $uhi-e-auth$: $UHI\ hfEl \in e1.ik-hfs$
 $\langle proof \rangle$

The attacker can also execute her event.

lemma *attr-executability*: $reach\ e1.dp2\ s-attr$
 $\langle proof \rangle$

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  $\times$  ahi list) set +
    fixes no-oracle :: msgterm  $\Rightarrow$  UINFO  $\Rightarrow$  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 $\Rightarrow$ msgterm **where** *source-extract* = *undefined*

definition *K-i* :: as $\Rightarrow$ msgterm $\Rightarrow$ msgterm **where**
K-i asid uinfo = $\langle$ AS asid, *source-extract uinfo* $\rangle$

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  $\Rightarrow$  UINFO
   $\Rightarrow$  EPIC-HF
   $\Rightarrow$  EPIC-HF option  $\Rightarrow$  bool where

```

```

  hf-valid (Num ts) uinfo (AHI = ahi, UHI = uhi, HVF = x) (Some (AHI = ahi2, UHI = uhi2,
HVF = x2))  $\longleftrightarrow$ 
  ( $\exists \sigma$  upif downif.  $\sigma = \text{Mac}[\text{macKey } (ASID \text{ ahi})] (L [\text{Num } ts, \text{upif}, \text{downif}, \text{uhi2}]) \wedge$ 
    ASIF (DownIF ahi) downif  $\wedge$  ASIF (UpIF ahi) upif  $\wedge$  uhi = Hash  $\sigma \wedge$ 
    x = Mac[K-i (ASID ahi) (Num uinfo)] (Num ts, Num uinfo,  $\sigma$ ))
| hf-valid (Num ts) uinfo (AHI = ahi, UHI = uhi, HVF = x) None  $\longleftrightarrow$ 
  ( $\exists \sigma$  upif downif.  $\sigma = \text{Mac}[\text{macKey } (ASID \text{ ahi})] (L [\text{Num } ts, \text{upif}, \text{downif}]) \wedge$ 
    ASIF (DownIF ahi) downif  $\wedge$  ASIF (UpIF ahi) upif  $\wedge$  uhi = Hash  $\sigma \wedge$ 
    x = Mac[K-i (ASID ahi) (Num uinfo)] (Num ts, Num uinfo,  $\sigma$ ))
| hf-valid - - - = False

```

abbreviation *upd-uinfo* :: nat $\Rightarrow$ EPIC-HF $\Rightarrow$ nat **where**
upd-uinfo uinfo hf $\equiv$ 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  $\Rightarrow$  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  $\Rightarrow$  ahi list where
  extr (Mac[-] (<-, -,  $\sigma$ )) = extrUhi (Hash  $\sigma$ )
| extr - = []

```

Extract the authenticated info field from a hop validation field.

```

fun extr-ainfo :: msgterm  $\Rightarrow$  msgterm where
  extr-ainfo (Mac[-] (Num ts, -, -)) = Num ts
| extr-ainfo - =  $\varepsilon$ 

```

abbreviation *term-ainfo* :: msgterm $\Rightarrow$ msgterm **where**
term-ainfo $\equiv$ 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  $\Rightarrow$  msgterm set where
  terms-hf hf = {HVF hf, UHI hf}

```

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

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**

$$\text{auth-restrict ainfo uinfo } l \equiv (\exists ts. \text{ ainfo} = \text{Num } ts)$$

We now define useful properties of the above definition.

lemma *hf-valid-invert*:

$$\text{hf-valid tsn uinfo hf } mo \longleftrightarrow$$

$$\begin{aligned} & ((\exists \text{ ahi ahi2 } \sigma \text{ ts upif downif asid } x \text{ upif2 downif2 asid2 uhi uhi2 } x2. \\ & \quad \text{hf} = (\text{AHI} = \text{ahi}, \text{UHI} = \text{uhi}, \text{HVF} = x) \wedge \\ & \quad \text{ASID } \text{ahi} = \text{asid} \wedge \text{ASIF } (\text{DownIF } \text{ahi}) \text{ downif} \wedge \text{ASIF } (\text{UpIF } \text{ahi}) \text{ upif} \wedge \\ & \quad \text{mo} = \text{Some } (\text{AHI} = \text{ahi2}, \text{UHI} = \text{uhi2}, \text{HVF} = x2) \wedge \\ & \quad \text{ASID } \text{ahi2} = \text{asid2} \wedge \text{ASIF } (\text{DownIF } \text{ahi2}) \text{ downif2} \wedge \text{ASIF } (\text{UpIF } \text{ahi2}) \text{ upif2} \wedge \\ & \quad \sigma = \text{Mac}[\text{macKey asid}] (L [\text{tsn}, \text{upif}, \text{downif}, \text{uhi2}]) \wedge \\ & \quad \text{tsn} = \text{Num } ts \wedge \\ & \quad \text{uhi} = \text{Hash } \sigma \wedge \\ & \quad x = \text{Mac}[K-i (\text{ASID } \text{ahi}) (\text{Num } \text{uinfo})] (\text{tsn}, \text{Num } \text{uinfo}, \sigma)) \\ \vee & ((\exists \text{ ahi } \sigma \text{ ts upif downif asid uhi } x. \\ & \quad \text{hf} = (\text{AHI} = \text{ahi}, \text{UHI} = \text{uhi}, \text{HVF} = x) \wedge \\ & \quad \text{ASID } \text{ahi} = \text{asid} \wedge \text{ASIF } (\text{DownIF } \text{ahi}) \text{ downif} \wedge \text{ASIF } (\text{UpIF } \text{ahi}) \text{ upif} \wedge \\ & \quad \text{mo} = \text{None} \wedge \\ & \quad \sigma = \text{Mac}[\text{macKey asid}] (L [\text{tsn}, \text{upif}, \text{downif}]) \wedge \\ & \quad \text{tsn} = \text{Num } ts \wedge \\ & \quad \text{uhi} = \text{Hash } \sigma \wedge \\ & \quad x = \text{Mac}[K-i (\text{ASID } \text{ahi}) (\text{Num } \text{uinfo})] (\text{tsn}, \text{Num } \text{uinfo}, \sigma)) \\ &) \\ & \langle \text{proof} \rangle \end{aligned}$$

lemma *hf-valid-auth-restrict[dest]*: $\text{hf-valid ainfo uinfo hf } z \implies \text{auth-restrict ainfo uinfo } l$
 $\langle \text{proof} \rangle$

lemma *auth-restrict-ainfo[dest]*: $\text{auth-restrict ainfo uinfo } l \implies \exists ts. \text{ ainfo} = \text{Num } ts$
 $\langle \text{proof} \rangle$

lemma *info-hvf*:

assumes $\text{hf-valid ainfo uinfo } m \text{ z } \text{HVF } m = \text{Mac}[k-i] \langle \text{ainfo}', \text{Num } \text{uinfo}', \sigma \rangle \vee \text{hf-valid ainfo}' \text{ uinfo}'$
 $m \text{ z}'$
shows $\text{uinfo} = \text{uinfo}' \text{ ainfo}' = \text{ainfo}$
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

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

declare $TWu.holds\text{-}set\text{-}list[dest]$
declare $TWu.holds\text{-}takeW\text{-}is\text{-}identity[simp]$
declare $parts\text{-}singleton[dest]$

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 $ik\text{-}hfs$), but to the underlying hop authenticators that are used to create them.

definition $ik\text{-}add :: msgterm\ set$ **where**
 $ik\text{-}add \equiv \{ \sigma \mid ainfo\ uinfo\ l\ hf\ \sigma\ k\text{-}i.$
 $(ainfo, l) \in auth\text{-}seg2\ uinfo$
 $\wedge hf \in set\ l \wedge HVF\ hf = Mac[k\text{-}i]\ \langle ainfo, Num\ uinfo, \sigma \rangle \}$

lemma $ik\text{-}addI$:

$\llbracket (ainfo, l) \in auth\text{-}seg2\ uinfo; hf \in set\ l; HVF\ hf = Mac[k\text{-}i]\ \langle ainfo, Num\ uinfo, \sigma \rangle \rrbracket \implies \sigma \in ik\text{-}add$
 $\langle proof \rangle$

lemma $ik\text{-}add\text{-}form$: $t \in ik\text{-}add \implies \exists\ asid\ l . t = Mac[macKey\ asid]\ l$

$\langle proof \rangle$

lemma $parts\text{-}ik\text{-}add[simp]$: $parts\ ik\text{-}add = ik\text{-}add$

$\langle proof \rangle$

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 $no\text{-}oracle$ appears here.

definition $ik\text{-}oracle :: msgterm\ set$ **where**
 $ik\text{-}oracle = \{ t \mid t\ ainfo\ hf\ l\ uinfo . hf \in set\ l \wedge hfs\text{-}valid\text{-}None\ ainfo\ uinfo\ l \wedge$
 $is\text{-}oracle\ ainfo\ uinfo \wedge (ainfo, l) \notin auth\text{-}seg2\ uinfo \wedge (t = HVF\ hf \vee t = UHI\ hf) \}$

lemma $ik\text{-}oracle\text{-}parts\text{-}form$:

$t \in ik\text{-}oracle \implies$

$(\exists\ asid\ l\ ainfo\ uinfo\ k\text{-}i . t = Mac[k\text{-}i]\ \langle ainfo, Num\ uinfo, Mac[macKey\ asid]\ l \rangle) \vee$
 $(\exists\ asid\ l . t = Hash\ (Mac[macKey\ asid]\ l))$
 $\langle proof \rangle$

lemma $parts\text{-}ik\text{-}oracle[simp]$: $parts\ ik\text{-}oracle = ik\text{-}oracle$

$\langle proof \rangle$

lemma $ik\text{-}oracle\text{-}simp$: $t \in ik\text{-}oracle \longleftrightarrow$

$(\exists\ ainfo\ hf\ l\ uinfo . hf \in set\ l \wedge hfs\text{-}valid\text{-}None\ ainfo\ uinfo\ l \wedge is\text{-}oracle\ ainfo\ uinfo$
 $\wedge (ainfo, l) \notin auth\text{-}seg2\ uinfo \wedge (t = HVF\ hf \vee t = UHI\ hf))$

$\langle proof \rangle$

3.7.3 Properties of the intruder knowledge, including $ik\text{-}add$ and $ik\text{-}oracle$

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

<proof>

lemma *ik-hfs-form*: $t \in \text{parts } ik\text{-hfs} \implies \exists t' . t = \text{Hash } t'$

<proof>

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

lemma *parts-ik-hfs*[*simp*]: $\text{parts } ik\text{-hfs} = ik\text{-hfs}$

<proof>

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*:

$t \in ik\text{-hfs} \longleftrightarrow (\exists t' . t = \text{Hash } t') \wedge (\exists hf . (t = \text{HVF } hf \vee t = \text{UHI } hf) \wedge (\exists hfs . hf \in \text{set } hfs \wedge (\exists ainfo \ uinfo . (ainfo, hfs) \in \text{auth-seg2 } uinfo \wedge (\exists \text{next} . hf\text{-valid } ainfo \ uinfo \ hf \text{next})))) (\text{is } ?lhs \longleftrightarrow ?rhs)$

<proof>

Properties of Intruder Knowledge

lemma *auth-ainfo*[*dest*]: $\llbracket (ainfo, hfs) \in \text{auth-seg2 } uinfo \rrbracket \implies \exists ts . ainfo = \text{Num } ts$

<proof>

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

lemma *analz-parts-ik*[*simp*]: $\text{analz } ik = \text{parts } ik$

<proof>

lemma *parts-ik*[*simp*]: $\text{parts } ik = ik$

<proof>

lemma *key-ik-bad*: $\text{Key } (\text{macK } asid) \in ik \implies asid \in \text{bad}$

<proof>

Hop authenticators are agnostic to uinfo field

fun *K-i-upd* :: $\text{msgterm} \Rightarrow \text{msgterm} \Rightarrow \text{msgterm}$ **where**

K-i-upd $\langle AS \ asid, - \rangle \ uinfo' = \langle AS \ asid, \text{source-extract } uinfo \rangle$

| *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* :: $UINFO \Rightarrow EPIC\text{-}HF \Rightarrow EPIC\text{-}HF$ **where**

uinfo-change-hf *new-uinfo* *hf* =

(*case* *HVF* *hf* *of* *Mac*[*k-i*] $\langle ainfo, \text{Num } uinfo, \sigma \rangle$

$\Rightarrow hf(\text{HVF} := \text{Mac}[K\text{-i-upd } k\text{-i } (\text{Num } \text{new-uinfo})] \langle ainfo, \text{Num } \text{new-uinfo}, \sigma \rangle) \mid - \Rightarrow hf)$

fun *uinfo-change* :: *UINFO* $\Rightarrow$ *EPIC-HF list* $\Rightarrow$ *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 next $\implies$ *hfs-valid ainfo new-uinfo (uinfo-change new-uinfo l) next*
 $\langle \text{proof} \rangle$

lemma *uinfo-change-hf-AHI*: *AHI (uinfo-change-hf new-uinfo hf)* = *AHI hf*
 $\langle \text{proof} \rangle$

lemma *uinfo-change-hf-AHIS[simp]*: *AHIS (map (uinfo-change-hf new-uinfo) l)* = *AHIS l*
 $\langle \text{proof} \rangle$

lemma *uinfo-change-auth-seg2*:

assumes *hf-valid ainfo uinfo m z* $\sigma = \text{Mac}[\text{Key } (\text{macK } \text{asid})] j$
HVF m = *Mac*[*k-i*] $\langle \text{ainfo}, \text{uinfo}', \sigma \rangle$ $\sigma \in \text{ik-add no-oracle ainfo uinfo}$
shows $\exists \text{hfs}. m \in \text{set hfs} \wedge (\exists \text{uinfo}''. (\text{ainfo}, \text{hfs}) \in \text{auth-seg2 uinfo}'')$
 $\langle \text{proof} \rangle$

lemma *MAC-synth-oracle*:

assumes *hf-valid ainfo uinfo m z HVF m* $\in \text{ik-oracle}$
shows *is-oracle ainfo uinfo*
 $\langle \text{proof} \rangle$

lemma *ik-oracle-is-oracle*:

$\llbracket \text{Mac}[\sigma] \langle \text{ainfo}, \text{Num uinfo} \rangle \in \text{ik-oracle} \rrbracket \implies \text{is-oracle ainfo uinfo}$
 $\langle \text{proof} \rangle$

lemma *MAC-synth-helper*:

$\llbracket \text{hf-valid ainfo uinfo m z}; \text{no-oracle ainfo uinfo};$
HVF m = *Mac*[*k-i*] $\langle \text{ainfo}, \text{Num uinfo}, \sigma \rangle$; $\sigma = \text{Mac}[\text{Key } (\text{macK } \text{asid})] j$; $\sigma \in \text{ik} \vee \text{HVF } m \in \text{ik} \rrbracket$
 $\implies \exists \text{hfs}. m \in \text{set hfs} \wedge (\exists \text{uinfo}'. (\text{ainfo}, \text{hfs}) \in \text{auth-seg2 uinfo}')$
 $\langle \text{proof} \rangle$

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

definition *mac-format* :: *msgterm* $\Rightarrow$ *as* $\Rightarrow$ *bool* **where**

mac-format m asid $\equiv \exists j \text{ ts uinfo k-i}. m = \text{Mac}[k-i] \langle \text{Num ts}, \text{uinfo}, \text{Mac}[\text{macKey asid}] j \rangle$

If a valid hop field is derivable by the attacker, but does not belong to the attacker, and is over a path origin that does not belong to an oracle query, then the hop field is already contained in the set of authorized segments.

lemma *MAC-synth*:

assumes *hf-valid ainfo uinfo m z HVF m* $\in \text{synth ik mac-format (HVF m) asid}$
 $\text{asid} \notin \text{bad checkInfo ainfo no-oracle ainfo uinfo}$
shows $\exists \text{hfs}. m \in \text{set hfs} \wedge (\exists \text{uinfo}'. (\text{ainfo}, \text{hfs}) \in \text{auth-seg2 uinfo}')$
 $\langle \text{proof} \rangle$

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

lemma *COND-honest-hf-analz*:

assumes $ASID (AHI\ hf) \notin bad\ hf\text{-}valid\ ainfo\ uinfo\ hf\ next\ terms\text{-}hf\ hf \subseteq synth\ (analz\ ik)$
 $no\text{-}oracle\ ainfo\ uinfo$
shows $terms\text{-}hf\ hf \subseteq analz\ ik$
 $\langle proof \rangle$

lemma *COND-terms-hf*:
assumes $hf\text{-}valid\ ainfo\ uinfo\ hf\ z$ **and** $HVF\ hf \in ik$ **and** $no\text{-}oracle\ ainfo\ uinfo$
shows $\exists hfs. hf \in set\ hfs \wedge (\exists uinfo. (ainfo, hfs) \in auth\text{-}seg2\ uinfo)$
 $\langle proof \rangle$

lemma *COND-extr-prefix-path*:
 $\llbracket hf\text{-}valid\ ainfo\ uinfo\ l\ next; next = None \rrbracket \implies prefix\ (extr\text{-}from\text{-}hd\ l)\ (AHIS\ l)$
 $\langle proof \rangle$

lemma *COND-path-prefix-extr*:
 $prefix\ (AHIS\ (hfs\text{-}valid\text{-}prefix\ ainfo\ uinfo\ l\ next))$
 $(extr\text{-}from\text{-}hd\ l)$
 $\langle proof \rangle$

lemma *COND-hf-valid-uinfo*:
 $\llbracket hf\text{-}valid\ ainfo\ uinfo\ hf\ next; hf\text{-}valid\ ainfo'\ uinfo'\ hf\ next \rrbracket \implies uinfo' = uinfo$
 $\langle proof \rangle$

lemma *COND-upd-uinfo-ik*:
 $\llbracket terms\text{-}uinfo\ uinfo \subseteq synth\ (analz\ ik); terms\text{-}hf\ hf \subseteq synth\ (analz\ ik) \rrbracket$
 $\implies terms\text{-}uinfo\ (upd\text{-}uinfo\ uinfo\ hf) \subseteq synth\ (analz\ ik)$
 $\langle proof \rangle$

lemma *COND-upd-uinfo-no-oracle*:
 $no\text{-}oracle\ ainfo\ uinfo \implies no\text{-}oracle\ ainfo\ (upd\text{-}uinfo\ uinfo\ fld)$
 $\langle proof \rangle$

lemma *COND-auth-restrict-upd*:
 $auth\text{-}restrict\ ainfo\ uinfo\ (x\#y\#hfs)$
 $\implies auth\text{-}restrict\ ainfo\ (upd\text{-}uinfo\ uinfo\ y)\ (y\#hfs)$
 $\langle proof \rangle$

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*

$upd\text{-}uinfo\ ik\text{-}add$
 $ik\text{-}oracle\ no\text{-}oracle$
 $\langle proof \rangle$

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 $\Rightarrow$ 'a fset $\Rightarrow$ 'a fset **where**

xor xs ys = (*xs* $\cup$ *ys*) $\mid\mid$ (*xs* $\cap$ *ys*)

lemma *xor-singleton*:

xor xs $\{| z |\}$ = (*if* *z* $\in$ *xs* *then* *xs* $\mid\mid$ $\{| z |\}$ *else* *fininsert z xs*)

xor $\{| z |\}$ *xs* = (*if* *z* $\in$ *xs* *then* *xs* $\mid\mid$ $\{| z |\}$ *else* *fininsert z xs*)

$\langle$ *proof* $\rangle$

declare *fininsertCI*[*rule del*]

declare *fininsertCI*[*intro*]

lemma *xor-assoc*: *xor* (*xor xs ys*) *zs* = *xor xs* (*xor ys zs*)

$\langle$ *proof* $\rangle$

lemma *xor-commut*: *xor xs ys* = *xor ys xs*

$\langle$ *proof* $\rangle$

lemma *xor-self-inv*: $\llbracket \text{xor } xs \text{ } ys = zs; xs = ys \rrbracket \Longrightarrow zs = \{\|\}$

$\langle$ *proof* $\rangle$

lemma *xor-self-inv'*: *xor xs xs* = $\{\|\}$

$\langle$ *proof* $\rangle$

lemma *xor-self-inv''*[*dest!*]: *xor xs ys* = $\{\|\}$ $\Longrightarrow xs = ys$

$\langle$ *proof* $\rangle$

lemma *xor-identity1*[*simp*]: *xor xs* $\{\|\}$ = *xs*

$\langle$ *proof* $\rangle$

lemma *xor-identity2*[*simp*]: *xor* $\{\|\}$ *xs* = *xs*

$\langle$ *proof* $\rangle$

lemma *xor-in*: *z* $\in$ *xs* $\Longrightarrow z \notin$ (*xor xs* $\{| z |\}$)

$\langle$ *proof* $\rangle$

lemma *xor-out*: *z* $\notin$ *xs* $\Longrightarrow z \in$ (*xor xs* $\{| z |\}$)

$\langle$ *proof* $\rangle$

lemma *xor-elem1*[*dest*]: $\llbracket x \in \text{fset } (\text{xor } X \text{ } Y); x \notin X \rrbracket \Longrightarrow x \in Y$

$\langle proof \rangle$

lemma *xor-elim2*[*dest*]: $\llbracket x \in fset \ (xor \ X \ Y); x \notin Y \rrbracket \implies x \in X$
 $\langle proof \rangle$

lemma *xor-finsert-self*: $xor \ (finsert \ x \ xs) \ \{|x|\} = xs - \{|x|\}$
 $\langle proof \rangle$

3.8.2 Lemmas referring to XOR and msgterm

lemma *FS-contains-elim*:
 assumes $elem = f \ (FS \ zs-s) \ zs-s = xor \ zs-b \ \{|elem|\} \wedge x. \ size \ (f \ x) > size \ x$
 shows $elem \in fset \ zs-b$
 $\langle proof \rangle$

lemma *FS-is-finsert-elim*:
 assumes $elem = f \ (FS \ zs-s) \ zs-s = xor \ zs-b \ \{|elem|\} \wedge x. \ size \ (f \ x) > size \ x$
 shows $zs-b = finsert \ elem \ zs-s$
 $\langle proof \rangle$

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 $xs = ys$
 $\langle proof \rangle$

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

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

lemma *fset-size-le*:
 assumes $x \in fset \ xs$
 shows $size \ x < Suc \ (\sum x \in fset \ xs. \ Suc \ (size \ x))$
 $\langle proof \rangle$

We can show that xor is a commutative function.

locale *abstract-xor*
begin
 sublocale *comp-fun-commute xor*
 $\langle proof \rangle$
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  $\times$  ahi list) set
begin

sublocale comp-fun-commute xor
  <proof>
```

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  $\Rightarrow$  msgterm fset
   $\Rightarrow$  SCION-HF
   $\Rightarrow$  SCION-HF option  $\Rightarrow$  bool where
  hf-valid (Num ts) uinfo (| AHI = ahi, UHI = -, HVF = x |) nxt  $\longleftrightarrow$ 
    ( $\exists$  upif downif.  $x = \text{Mac}[\text{macKey } (ASID \ ahi)] \ (L \ [\text{Num } ts, \text{upif}, \text{downif}, \text{FS } uinfo]) \wedge$ 
      ASIF (DownIF ahi) downif  $\wedge$  ASIF (UpIF ahi) upif)
  | hf-valid - - - = False
```

Updating the uinfo field involves XOR in 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  $\Rightarrow$  SCION-HF  $\Rightarrow$  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:
  [|  $x = \text{Mac}[\text{macKey } asid'a] \ (L \ [ts, \text{upif}'a, \text{downif}'a, \text{FS } segid'])$ ;
    $fcard \ segid' = fcard \ (xor \ segid \ \{|x|\})$ ;  $x \in segid$  |]
```

```

    ⇒ (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)
  ⟨proof⟩

```

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 - = []
  ⟨proof⟩
termination
  ⟨proof⟩

```

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)
    ∧ (case l of [] ⇒ (uinfo = {||}) |
      - ⇒ 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 ⇒ msgterm set where

```

$terms\text{-}hf\ hf = \{HVF\ hf\}$

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

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

abbreviation $no\text{-}oracle :: 'a\ info \Rightarrow msgterm\ fset \Rightarrow bool$ **where** $no\text{-}oracle \equiv (\lambda\ -. \ True)$

Properties following from definitions

We now define useful properties of the above definition.

lemma *hf-valid-invert*:

$hf\text{-}valid\ tsn\ uinfo\ hf\ next \longleftrightarrow$
 $((\exists\ ahi\ ts\ upif\ downif\ asid\ x.$
 $\quad hf = \langle AHI = ahi, UHI = (), HVF = x \rangle \wedge$
 $\quad ASID\ ahi = asid \wedge ASIF\ (DownIF\ ahi)\ downif \wedge ASIF\ (UpIF\ ahi)\ upif \wedge$
 $\quad x = Mac[macKey\ asid]\ (L\ [tsn, upif, downif, FS\ uinfo]) \wedge$
 $\quad tsn = Num\ ts)$
 $)$
 $\langle proof \rangle$

lemma *info-hvf*:

assumes $hf\text{-}valid\ ainfo\ uinfo\ m\ z\ hf\text{-}valid\ ainfo'\ uinfo'\ m'\ z'\ HVF\ m = HVF\ m'$
shows $ainfo' = ainfo\ m' = m$
 $\langle proof \rangle$

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*
 $terms\text{-}hf\ terms\text{-}uinfo\ upd\text{-}uinfo\ no\text{-}oracle$

$\langle proof \rangle$

declare *TWu.holds-set-list*[*dest*]

declare *TWu.holds-takeW-is-identity*[*simp*]

declare *parts-singleton*[*dest*]

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

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

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-uinfo
term-uinfo

terms-hf upd-uinfo ik-add ik-oracle

<proof>

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 $\implies$ hf-valid ainfo uinfo hf nxt'*

<proof>

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

lemma *auth-properties*:

assumes *hf $\in$ set hfs hfs-valid ainfo uinfo hfs nxt auth-restrict ainfo uinfo hfs*

t = HVF hf

shows *($\exists t' . t = \text{Hash } t'$)*

$\wedge (\exists \text{uinfo}' . \text{auth-restrict ainfo uinfo}' \text{ hfs})$

$\wedge (\exists \text{nxt} . \text{hf-valid ainfo uinfo}' \text{ hf } \text{nxt})$

<proof>

lemma *ik-hfs-form*: *t $\in$ parts ik-hfs $\implies \exists t' . t = \text{Hash } t'$*

<proof>

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

lemma *parts-ik-hfs[simp]*: *parts ik-hfs = ik-hfs*

<proof>

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 $\longleftrightarrow (\exists t' . t = \text{Hash } t') \wedge (\exists \text{hf} . t = \text{HVF hf}$

$\wedge (\exists \text{hfs uinfo} . \text{hf} \in \text{set hfs} \wedge (\exists \text{ainfo} . (\text{ainfo}, \text{hfs}) \in (\text{auth-seg2 uinfo})$

*$\wedge (\exists \text{nxt uinfo}' . \text{hf-valid ainfo uinfo}' \text{ hf } \text{nxt})))$ (**is** ?lhs $\longleftrightarrow$?rhs)*

<proof>

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)

$\implies$ auth-restrict ainfo (upd-uinfo uinfo y) (y#hfs)

<proof>

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 *$\exists \text{hf} . t = \text{HVF hf} \wedge \text{hf} \in \text{set hfs}$*

<proof>

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

lemma *ik-uinfo-in-ik-hfs*: $t \in ik\text{-}uinfo \implies t \in ik\text{-}hfs$
 $\langle proof \rangle$

Properties of Intruder Knowledge

lemma *auth-ainfo[dest]*: $\llbracket (ainfo, hfs) \in (auth\text{-}seg2\ uinfo) \rrbracket \implies \exists ts . ainfo = Num\ ts$
 $\langle proof \rangle$

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

lemma *ik-simpler*:
 $ik = ik\text{-}hfs$
 $\cup \{term\text{-}ainfo\ ainfo \mid ainfo\ hfs\ uinfo. (ainfo, hfs) \in (auth\text{-}seg2\ uinfo)\}$
 $\cup Key'(macK\ bad)$
 $\langle proof \rangle$

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

lemma *analz-parts-ik[simp]*: $analz\ ik = parts\ ik$
 $\langle proof \rangle$

lemma *parts-ik[simp]*: $parts\ ik = ik$
 $\langle proof \rangle$

lemma *key-ik-bad*: $Key\ (macK\ asid) \in ik \implies asid \in bad$
 $\langle proof \rangle$

lemma *MAC-synth-helper*:
assumes *hf-valid ainfo uinfo m z HVF m = Mac[Key (macK asid)] j HVF m ∈ ik*
shows $\exists hfs. m \in set\ hfs \wedge (\exists uinfo'. (ainfo, hfs) \in auth\text{-}seg2\ uinfo')$
 $\langle proof \rangle$

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**
 $mac\text{-}format\ m\ asid \equiv \exists j . m = Mac[macKey\ asid]\ j$

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

lemma *MAC-synth*:
assumes *hf-valid ainfo uinfo m z HVF m ∈ synth ik mac-format (HVF m) asid asid ∉ bad*
shows $\exists hfs. m \in set\ hfs \wedge$
 $(\exists uinfo'. (ainfo, hfs) \in auth\text{-}seg2\ uinfo')$
 $\langle proof \rangle$

3.9.4 Lemmas helping with conditions relating to extract

Resolve the definite descriptor operator THE.

lemma *THE-nextmac*:

assumes $hvf = \text{Mac}[\text{macKey } askey] (L [\text{Num } ts, upif, downif, FS (xor \text{info } \{|hvf|\})])$
shows $(THE \text{nextmac}. \exists asid' upif' downif'. \\ \text{nextmac} = \text{Mac}[\text{macKey } asid'] (L [\text{Num } ts, upif', downif', FS (xor \text{info } \{|\text{nextmac}|\})])) \\ = hvf$
 $\langle proof \rangle$

lemma *hf-valid-uinfo*:

assumes $hf\text{-valid } ainfo (upd\text{-uinfo } uinfo \ y) \ y \text{ next } hvfy = HVF \ y$
shows $hvfy \in fset \ uinfo$
 $\langle proof \rangle$

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 $hf\text{-valid } ainfo \ uinfo \ x \text{ next } hf\text{-valid } ainfo (upd\text{-uinfo } uinfo \ y) \ y \text{ next}'$
shows $extr (HVF \ x) = AHI \ x \ \# \ extr (HVF \ y)$
 $\langle proof \rangle$

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

lemma *COND-honest-hf-analz*:

assumes $ASID (AHI \ hf) \notin bad \ hf\text{-valid } ainfo \ uinfo \ hf \text{ next } terms\text{-hf } hf \subseteq synth (analz \ ik)$
 $no\text{-oracle } ainfo \ uinfo$
shows $terms\text{-hf } hf \subseteq analz \ ik$
 $\langle proof \rangle$

lemma *COND-terms-hf*:

assumes $hf\text{-valid } ainfo \ uinfo \ hf \text{ next } terms\text{-hf } hf \subseteq analz \ ik$
 $no\text{-oracle } ainfo \ uinfo$
shows $\exists hfs. hf \in set \ hfs \wedge (\exists uinfo'. (ainfo, hfs) \in (auth\text{-seg2 } uinfo'))$
 $\langle proof \rangle$

lemmas $COND\text{-auth-restrict-upd} = auth\text{-restrict-upd}$

lemma *COND-extr-prefix-path*:

$\llbracket hfs\text{-valid } ainfo \ uinfo \ l \text{ next}; auth\text{-restrict } ainfo \ uinfo \ l \rrbracket \implies prefix (extr\text{-from-hd } l) (AHIS \ l)$
 $\langle proof \rangle$

lemma *COND-path-prefix-extr*:

$prefix (AHIS (hfs\text{-valid-prefix } ainfo \ uinfo \ l \text{ next}))$
 $(extr\text{-from-hd } l)$
 $\langle proof \rangle$

lemma *COND-hf-valid-uinfo*:

$\llbracket hf\text{-valid } ainfo \ uinfo \ hf \text{ next}; hf\text{-valid } ainfo' \ uinfo' \ hf \text{ next} \rrbracket \implies uinfo' = uinfo$
 $\langle proof \rangle$

lemma *COND-upd-uinfo-ik*:

$\llbracket terms\text{-uinfo } uinfo \subseteq synth (analz \ ik); terms\text{-hf } hf \subseteq synth (analz \ ik) \rrbracket$
 $\implies terms\text{-uinfo } (upd\text{-uinfo } uinfo \ hf) \subseteq synth (analz \ ik)$
 $\langle proof \rangle$

lemma *COND-upd-ainfo-no-oracle*:
no-oracle ainfo uinfo $\implies$ no-oracle ainfo (upd-ainfo uinfo fld)
<proof>

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-ainfo ik-add
ik-oracle no-oracle
<proof>

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*:
 $\llbracket normalized\ (FS\ (upd-ainfo\ info\ y));\ normalized\ (FS\ \{|HVF\ y\ |}) \rrbracket$
 $\implies normalized\ (FS\ info)$
<proof>

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

lemma *auth-ainfo-normalized*:
 $\llbracket hfs-valid\ ainfo\ uinfo\ hfs\ nxt;\ auth-restrict\ ainfo\ uinfo\ hfs \rrbracket \implies normalized\ (FS\ uinfo)$
<proof>

lemma *auth-normalized-hf*:
assumes *auth-restrict ainfo uinfo (pre @ hf # post)*
hfs-valid ainfo (upds-ainfo-shifted uinfo pre hf) (hf # post) nxt
upds-ainfo-shifted uinfo pre hf = hf-ainfo
shows *normalized (HVF hf)*
<proof>

lemma *auth-normalized*:
 $\llbracket hf \in set\ hfs;\ hfs-valid\ ainfo\ uinfo\ hfs\ nxt;\ auth-restrict\ ainfo\ uinfo\ hfs \rrbracket$
 $\implies normalized\ (HVF\ hf)$
<proof>

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*: $t \in synth\ (analz\ ik) \implies normalized\ t$
<proof>

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* $\times$ *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* $\Rightarrow$ *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* $\implies$ *ahi2* = *ahi1*

<proof>

fun *hf2term* :: *nat* *ahi-scheme* $\Rightarrow$ *msgterm* **where**

hf2term (*UpIF* = *upif*, *DownIF* = *downif*, *ASID* = *asid*, ... = *tag*)
= *L* [*if2term upif*, *if2term downif*, *Num asid*, *Num tag*]

fun *term2hf* :: *msgterm* $\Rightarrow$ *nat* *ahi-scheme* **where**

$term2hf (L [upif, downif, Num asid, Num tag])$
 $= \langle UpIF = term2if upif, DownIF = term2if downif, ASID = asid, \dots = tag \rangle$

lemma $term2hf-hf2term[simp]: term2hf (hf2term hf) = hf \langle proof \rangle$

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 $fullpath :: ICING-HF list \Rightarrow msgterm$ **where**
 $fullpath hfs = L (map (hf2term o AHI) hfs)$

definition $maccontents$ **where**
 $maccontents ahi hfs PoC-i-expire$
 $= \langle Mac[sntag ahi] \langle fullpath hfs, Num PoC-i-expire \rangle, \langle Num 0, Hash (fullpath hfs) \rangle \rangle$

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 $hf-valid :: msgterm \Rightarrow msgterm$
 $\Rightarrow ICING-HF list$
 $\Rightarrow ICING-HF$
 $\Rightarrow bool$ **where**
 $hf-valid (Num PoC-i-expire) uinfo hfs \langle AHI = ahi, UHI = uhi, HVF = A-i \rangle \longleftrightarrow$
 $uhi = () \wedge uinfo = \varepsilon \wedge A-i = Hash (maccontents ahi hfs PoC-i-expire)$
 $| hf-valid - - - = False$

We can extract the entire path (past and future) from the hvf field.

fun $extr :: msgterm \Rightarrow nat ahi-scheme list$ **where**
 $extr (Mac[Mac[-] \langle L fullpathhfs, Num PoC-i-expire \rangle] -)$
 $= map term2hf fullpathhfs$
 $| extr - = []$

Extract the authenticated info field from a hop validation field.

fun $extr-ainfo :: msgterm \Rightarrow msgterm$ **where**
 $extr-ainfo (Mac[-] (L (Num ts \# xs))) = Num ts$
 $| extr-ainfo - = \varepsilon$

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

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 $terms-hf :: ICING-HF \Rightarrow msgterm set$ **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* $\equiv (\lambda _ _ . \text{True})$

We now define useful properties of the above definition.

lemma *hf-valid-invert*:

hf-valid tsn uinfo hfs hf $\longleftrightarrow$
 $(\exists \text{ PoC-i-expire ahi A-i } . \text{tsn} = \text{Num PoC-i-expire} \wedge \text{ahi} = \text{AHI hf} \wedge$
 $\text{UHI hf} = () \wedge \text{uinfo} = \varepsilon \wedge$
 $\text{HVF hf} = \text{A-i} \wedge$
 $\text{A-i} = \text{Hash (maccontents ahi hfs PoC-i-expire)})$
 $\langle \text{proof} \rangle$

lemma *hf-valid-auth-restrict[dest]*: *hf-valid ainfo uinfo hfs hf* $\implies$ *auth-restrict ainfo uinfo l*
 $\langle \text{proof} \rangle$

lemma *auth-restrict-ainfo[dest]*: *auth-restrict ainfo uinfo l* $\implies \exists \text{ts. ainfo} = \text{Num ts}$
 $\langle \text{proof} \rangle$

lemma *auth-restrict-uinfo[dest]*: *auth-restrict ainfo uinfo l* $\implies \text{uinfo} = \varepsilon$
 $\langle \text{proof} \rangle$

lemma *info-hvf*:

assumes *hf-valid ainfo uinfo hfs m hf-valid ainfo' uinfo' hfs' m'*
 $\text{HVF } m = \text{HVF } m' \text{ } m \in \text{set hfs } m' \in \text{set hfs'}$
shows *ainfo' = ainfo m' = m*
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

declare *parts-singleton[dest]*

definition *ik-add* :: *msgterm set* **where**

ik-add $\equiv \{ \text{PoC} \mid \text{ainfo } l \text{ uinfo hf PoC pkthash.}$
 $(\text{ainfo}, l) \in \text{auth-seg2 uinfo}$
 $\wedge \text{hf} \in \text{set } l \wedge \text{HVF hf} = \text{Mac}[\text{PoC}] \text{ pkthash} \}$

lemma *ik-addI*:

$\llbracket (\text{ainfo}, l) \in \text{local.auth-seg2 uinfo; hf} \in \text{set } l; \text{HVF hf} = \text{Mac}[\text{PoC}] \text{ pkthash} \rrbracket \implies \text{PoC} \in \text{ik-add}$
 $\langle \text{proof} \rangle$

lemma *ik-add-form*:

$t \in \text{ik-add} \implies \exists \text{ asid upif downif tag } l . t = \text{Mac}[\langle \text{macKey asid, if2term upif, if2term downif, Num tag} \rangle] l$
 $\langle \text{proof} \rangle$

lemma *elem-eq*: $\llbracket x \in xs; x = y; xs = ys \rrbracket \implies y \in ys$
 $\langle proof \rangle$

lemma *valid-hf-eq*:
 $\llbracket HVF\ hf = Mac[Mac[sntag\ (AHI\ hf)]\ \langle fullpath\ hfs,\ ainfo' \rangle]\ \langle Num\ 0,\ Hash\ (fullpath\ hfs) \rangle;$
 $HVF\ hf' = Mac[Mac[sntag\ (AHI\ hf)]\ \langle fullpath\ hfs,\ ainfo' \rangle]\ pkthash;$
 $(ainfo', l) \in auth-seg2\ uinfo; hf' \in set\ l \rrbracket$
 $\implies hf = hf'$
 $\langle proof \rangle$

lemma *parts-ik-add[simp]*: $parts\ ik-add = ik-add$
 $\langle proof \rangle$

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

lemma *uinfo-empty[dest]*: $(ainfo, hfs) \in auth-seg2\ uinfo \implies uinfo = \varepsilon$
 $\langle proof \rangle$

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
 $\langle proof \rangle$

lemma *ik-hfs-form*: $t \in parts\ ik-hfs \implies \exists\ t' . t = Hash\ t'$
 $\langle proof \rangle$

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

lemma *parts-ik-hfs[simp]*: $parts\ ik-hfs = ik-hfs$
 $\langle proof \rangle$

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*:

$t \in ik-hfs \iff (\exists\ t' . t = Hash\ t') \wedge (\exists\ hf . t = HVF\ hf$
 $\wedge (\exists\ hfs. hf \in set\ hfs \wedge (\exists\ ainfo\ uinfo. (ainfo, hfs) \in auth-seg2\ uinfo$
 $\wedge hf-valid\ ainfo\ uinfo\ hfs\ hf)))\ (is\ ?lhs \iff ?rhs)$

$\langle proof \rangle$

lemma *ik-uinfo-empty[simp]*: $ik-uinfo = \{\varepsilon\}$
 $\langle proof \rangle$

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

Properties of Intruder Knowledge

lemma *auth-ainfo[dest]*: $\llbracket (ainfo, hfs) \in auth-seg2\ uinfo \rrbracket \implies \exists\ ts . ainfo = Num\ ts$
 $\langle proof \rangle$

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

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

lemma *analz-parts-ik[simp]*: $analz\ ik = parts\ ik$
 $\langle proof \rangle$

lemma *parts-ik[simp]*: $parts\ ik = ik$
 $\langle proof \rangle$

lemma *sntag-synth-bad*: $sntag\ ahi \in synth\ ik \implies ASID\ ahi \in bad$
 $\langle proof \rangle$

lemma *HF-eq*:
 $\llbracket AHI\ hf' = AHI\ hf; UHI\ hf' = UHI\ hf; HVF\ hf' = HVF\ hf \rrbracket \implies hf' = (hf::('x, 'y)HF)$
 $\langle proof \rangle$

3.10.4 Direct proof goals for interpretation of *dataplane-3-undirected*

lemma *COND-honest-hf-analz*:
assumes $ASID\ (AHI\ hf) \notin bad$ $hf\text{-}valid\ ainfo\ uinfo\ hfs\ hf\ terms\text{-}hf\ hf \subseteq synth\ (analz\ ik)$
 $no\text{-}oracle\ ainfo\ uinfo\ hf \in set\ hfs$
shows $terms\text{-}hf\ hf \subseteq analz\ ik$
 $\langle proof \rangle$

lemma *COND-terms-hf*:
assumes $hf\text{-}valid\ ainfo\ uinfo\ hfs\ hf$ **and** $HVF\ hf \in ik$ **and** $no\text{-}oracle\ ainfo\ uinfo$ **and** $hf \in set\ hfs$
shows $\exists\ hfs. hf \in set\ hfs \wedge (\exists\ uinfo' . (ainfo, hfs) \in auth-seg2\ uinfo')$
 $\langle proof \rangle$

lemma *COND-extr*:
 $\llbracket hf\text{-}valid\ ainfo\ uinfo\ l\ hf \rrbracket \implies extr\ (HVF\ hf) = AHIS\ l$
 $\langle proof \rangle$

lemma *COND-hf-valid-uinfo*:
 $\llbracket hf\text{-}valid\ ainfo\ uinfo\ l\ hf; hf\text{-}valid\ ainfo'\ uinfo'\ l'\ hf \rrbracket$
 $\implies uinfo' = uinfo$
 $\langle proof \rangle$

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
 $\langle proof \rangle$

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  $\times$  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  $\Rightarrow$  msgterm where
  sntag ( $\langle UpIF = upif, DownIF = downif, ASID = asid \rangle$ ) =  $\langle macKey\ asid, \langle if2term\ upif, if2term\ downif \rangle \rangle$ 

```

```

lemma sntag-eq: sntag ahi2 = sntag ahi1  $\implies$  ahi2 = ahi1
   $\langle proof \rangle$ 

```

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  $\Rightarrow$  msgterm
   $\Rightarrow$  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-i-expire) # (map (hf2term o AHI) hfs))) ∧ uinfo = ε
| 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-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 ⇔
  (∃ ts ahi. tsn = Num ts ∧ ahi = AHI hf ∧
   UHI hf = () ∧
   HVF hf = Mac[sntag ahi] (L ((Num ts) # (map (hf2term o AHI) hfs))) ∧ uinfo = ε)
  ⟨proof⟩

```

```

lemma hf-valid-auth-restrict[dest]: hf-valid ainfo uinfo hfs hf ⇒ auth-restrict ainfo uinfo l
  ⟨proof⟩

```

```

lemma auth-restrict-ainfo[dest]: auth-restrict ainfo uinfo l ⇒ ∃ ts. ainfo = Num ts
  ⟨proof⟩

```

```

lemma auth-restrict-uinfo[dest]: auth-restrict ainfo uinfo l ⇒ uinfo = ε
  ⟨proof⟩

```

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*
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

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*
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

lemma *ik-hfs-form: t ∈ parts ik-hfs ⇒ ∃ t' . t = Hash t'*
 $\langle \text{proof} \rangle$

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

lemma *parts-ik-hfs[simp]: parts ik-hfs = ik-hfs*
 $\langle \text{proof} \rangle$

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*:
 $t \in \text{ik-hfs} \longleftrightarrow (\exists t' . t = \text{Hash } t') \wedge (\exists hf . t = \text{HVF } hf$
 $\wedge (\exists hfs . hf \in \text{set } hfs \wedge (\exists ainfo uinfo . (ainfo, hfs) \in \text{auth-seg2 } uinfo$
 $\wedge hf\text{-valid } ainfo uinfo hfs hf))) \text{ (is ?lhs } \longleftrightarrow \text{ ?rhs)}$
 $\langle \text{proof} \rangle$

lemma *ik-uinfo-empty*[simp]: $ik-uinfo = \{\varepsilon\}$
 $\langle proof \rangle$
declare *ik-uinfo-def*[simp del]

Properties of Intruder Knowledge

lemma *auth-ainfo*[dest]: $\llbracket (ainfo, hfs) \in auth-seg2\ uinfo \rrbracket \implies \exists\ ts.\ ainfo = Num\ ts$
 $\langle proof \rangle$

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

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

lemma *analz-parts-ik*[simp]: $analz\ ik = parts\ ik$
 $\langle proof \rangle$

lemma *parts-ik*[simp]: $parts\ ik = ik$
 $\langle proof \rangle$

lemma *sntag-synth-bad*: $sntag\ ahi \in synth\ ik \implies ASID\ ahi \in bad$
 $\langle proof \rangle$

3.11.4 Direct proof goals for interpretation of *dataplane-3-undirected*

lemma *COND-honest-hf-analz*:
assumes $ASID\ (AHI\ hf) \notin bad$ $hf-valid\ ainfo\ uinfo\ hfs\ hf\ terms-hf\ hf \subseteq synth\ (analz\ ik)$
 $no-oracle\ ainfo\ uinfo\ hf \in set\ hfs$
shows $terms-hf\ hf \subseteq analz\ ik$
 $\langle proof \rangle$

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 $\exists\ hfs.\ hf \in set\ hfs \wedge (\exists\ uinfo'. (ainfo, hfs) \in auth-seg2\ uinfo')$
 $\langle proof \rangle$

lemma *COND-extr*:
 $\llbracket hf-valid\ ainfo\ uinfo\ l\ hf \rrbracket \implies extr\ (HVF\ hf) = AHIS\ l$
 $\langle proof \rangle$

lemma *COND-hf-valid-uinfo*:
 $\llbracket hf-valid\ ainfo\ uinfo\ l\ hf; hf-valid\ ainfo'\ uinfo'\ l'\ hf \rrbracket$
 $\implies uinfo' = uinfo$
 $\langle proof \rangle$

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*
 $\langle proof \rangle$

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* $\times$ *ahi list*) *set*

+ **assumes** *auth-seg0-no-dups*:

$\llbracket (ainfo, hfs) \in auth-seg0; hf \in set\ hfs; hf' \in set\ hfs; ASID\ hf' = ASID\ hf \rrbracket \implies hf' = hf$

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* $\Rightarrow$ *msgterm* **where**

sntag ($\langle UpIF = upif, DownIF = downif, ASID = asid \rangle$) = *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* $\Rightarrow$ *msgterm*

$\Rightarrow$ *ICING-HF list*

$\Rightarrow$ *ICING-HF*

$\Rightarrow$ *bool* **where**

hf-valid (*Num PoC-i-expire*) *uinfo hfs* ($\langle AHI = ahi, UHI = uhi, HVF = x \rangle \longleftrightarrow uhi = () \wedge$

$x = \text{Mac}[\text{sntag } \text{ahi}] (L ((\text{Num } \text{PoC-i-expire}) \# (\text{map } (\text{hf2term } o \text{ AHI}) \text{ hfs}))) \wedge \text{uinfo} = \varepsilon$
 $| \text{hf-valid} - - - = \text{False}$

We can extract the entire path (past and future) from the hvf field.

fun *extr* :: *msgterm* $\Rightarrow$ *ahi list* **where**
extr (*Mac*[-] (*L hfs*))
 $= \text{map } \text{term2hf } (\text{tl } \text{hfs})$
 $| \text{extr } - = []$

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*
 $| \text{extr-ainfo } - = \varepsilon$

abbreviation *term-ainfo* :: *msgterm* $\Rightarrow$ *msgterm* **where**
term-ainfo $\equiv \text{id}$

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 $\equiv (\exists \text{ts. ainfo} = \text{Num ts}) \wedge (\text{uinfo} = \varepsilon)$

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* $\Rightarrow$ *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* $\equiv (\lambda - -. \text{True})$

We now define useful properties of the above definition.

lemma *hf-valid-invert*:
 $\text{hf-valid tsn uinfo hfs hf} \longleftrightarrow$
 $(\exists \text{ts ahi. tsn} = \text{Num ts} \wedge \text{ahi} = \text{AHI hf} \wedge$
 $\text{UHI hf} = () \wedge$
 $\text{HVF hf} = \text{Mac}[\text{sntag ahi}] (L ((\text{Num ts}) \# (\text{map } (\text{hf2term } o \text{ AHI}) \text{ hfs}))) \wedge \text{uinfo} = \varepsilon)$
 $\langle \text{proof} \rangle$

lemma *hf-valid-auth-restrict[dest]*: $\text{hf-valid ainfo uinfo hfs hf} \implies \text{auth-restrict ainfo uinfo l}$
 $\langle \text{proof} \rangle$

lemma *auth-restrict-ainfo[dest]*: $\text{auth-restrict ainfo uinfo l} \implies \exists \text{ts. ainfo} = \text{Num ts}$
 $\langle \text{proof} \rangle$

lemma *auth-restrict-uinfo[dest]*: $\text{auth-restrict ainfo uinfo l} \implies \text{uinfo} = \varepsilon$
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

declare *parts-singleton*[*dest*]

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

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

lemma *uinfo-empty*[*dest*]: $(\text{ainfo}, \text{hfs}) \in \text{auth-seg2 } \text{uinfo} \implies \text{uinfo} = \varepsilon$
 $\langle \text{proof} \rangle$

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
 $\langle \text{proof} \rangle$

lemma *ik-hfs-form*: $t \in \text{parts } \text{ik-hfs} \implies \exists t' . t = \text{Hash } t'$
 $\langle \text{proof} \rangle$

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

lemma *parts-ik-hfs*[*simp*]: $\text{parts } \text{ik-hfs} = \text{ik-hfs}$
 $\langle \text{proof} \rangle$

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*:
 $t \in \text{ik-hfs} \iff (\exists t' . t = \text{Hash } t') \wedge (\exists \text{hf} . t = \text{HVF } \text{hf}$
 $\wedge (\exists \text{hfs } \text{uinfo} . \text{hf} \in \text{set } \text{hfs} \wedge (\exists \text{ainfo} . (\text{ainfo}, \text{hfs}) \in \text{auth-seg2 } \text{uinfo}$
 $\wedge \text{hf-valid } \text{ainfo } \text{uinfo } \text{hfs } \text{hf}))) \text{ (is ?lhs } \iff \text{ ?rhs)}$
 $\langle \text{proof} \rangle$

lemma *ik-uinfo-empty*[*simp*]: $\text{ik-uinfo} = \{\varepsilon\}$
 $\langle \text{proof} \rangle$

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

Properties of Intruder Knowledge

lemma *auth-ainfo*[*dest*]: $\llbracket (\text{ainfo}, \text{hfs}) \in \text{auth-seg2 } \text{uinfo} \rrbracket \implies \exists \text{ts} . \text{ainfo} = \text{Num } \text{ts}$
 $\langle \text{proof} \rangle$

lemma *Num-ik*[*intro*]: $\text{Num } \text{ts} \in \text{ik}$
 $\langle \text{proof} \rangle$

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*
 <proof>

lemma *parts-ik[simp]*: *parts ik = ik*
 <proof>

lemma *sntag-synth-bad*: *sntag ahi ∈ synth ik ⇒ ASID ahi ∈ bad*
 <proof>

lemma *back-subst-set-member*: $\llbracket hf' \in \text{set } hfs; hf' = hf \rrbracket \Rightarrow hf \in \text{set } hfs$ <proof>

lemma *sntag-asid*: *sntag hf = sntag hf' ⇒ ASID hf' = ASID hf* <proof>

lemma *map-hf2term-eq*: *map (λx. hf2term (AHI x)) hfs = map (λx. hf2term (AHI x)) hfs'*
 $\Rightarrow AHIS\ hfs' = AHIS\ hfs$ <proof>

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>

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 $\exists hfs. hf \in \text{set } hfs \wedge (\exists uinfo'. (ainfo, hfs) \in \text{auth-seg2 } uinfo')$

<proof>

lemma *COND-extr*:

$\llbracket hf\text{-valid } ainfo\ uinfo\ l\ hf \rrbracket \Rightarrow \text{extr } (HVF\ hf) = AHIS\ l$

<proof>

lemma *COND-hf-valid-uinfo*:

$\llbracket hf\text{-valid } ainfo\ uinfo\ l\ hf; hf\text{-valid } ainfo'\ uinfo'\ l'\ hf \rrbracket$
 $\Rightarrow uinfo' = uinfo$

<proof>

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

<proof>

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
```